Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Nicolas Frederic Rung
BACHELOR THESIS

Improving the suggestion system of a cloud-based web application

Submitted on 3.11.2025

Supervisors: Prof. Dr. Dirk Riehle, M.B.A.
Dr. Andreas Kaufmann
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

Mühlhausen, 31.10.2025

License

_____

Mühlhausen, 31.10.2025

Abstract

QDAcity is a collaborative cloud-based web application for qualitative data analysis. Being a collaborative application, making suggestions can be very useful tool for many users. For this QDAcity offers a suggestion system, where instead of directly editing making changes, a user can suggest making changes. Up until now the suggestion system was very rudimentary, only offering suggestions for codes, a concept used in qualitative data analysis.

This thesis implements new functionality for this suggestion system, expanding the types of suggestions a user can make, validating that those suggestions are always possible and improving some central functions of the suggestion system. This thesis presents the final state of the implemented functionalities, while only describing steps in the development process if relevant to the explanation. At the end the project's success is evaluated and some ideas for future work are mentioned.

Contents

Contents

List of figures

1 Introduction

One of the big strengths of cloud-based web applications is the ability for collaboration and to share data between coworkers. A common way to enable the user to work in such ways is the ability to make suggestions to coworkers and discuss them. A web application that offers such a suggestion feature is QDAcity[1].

QDAcity is a cloud-based web application for qualitative data analysis, which focuses on collaborative workflows. A good user experience for making suggestions, which this thesis contributes, is therefore of particular value.

Qualitative data analysis[2] is a process which researchers use to gain insight into non-numerical data. Data is first gathered, then organized and finally interpreted to uncover patterns and themes.

A researcher starts out with some data the researcher wants to analyze. Text documents, PDF documents, interview transcripts etc. before moving on to one of the key processes in the analysis, called coding.

The coding process starts out with the creation of so called “codes”. These codes represent the theoretical structures the researcher is interested in. For example, in linguistics studies where a researcher is interested in patterns of how people express politeness, different codes could be used for each linguistic strategy being analyzed. Codes can also have descriptions and labels to further define when they should be used or what they represent. All codes, their descriptions and labels are stored in a so called “codebook”, which represents a foundational common understanding shared by all researchers in the project. QDAcity supports the creation of such codebooks with all the aforementioned features.

A code can now be applied to a document, which is called a “coding”. Codings mark the location or text to which the code was applied to. The researcher can then evaluate statistics for his codings, how often they were used and whether they overlap and many more, to gain insight into the data.

QDAcity also offers a lot of other features to aid qualitative data analysis but they aren’t related to this thesis.

1.1 Motivation

At the beginning of this thesis the suggestions feature in QDAcity was very rudimentary. Suggestions were limited to creating, deleting or relocating a code, or editing a code’s properties. The whole UI and UX were also very cut down. To start off an extra step was necessary to activate suggestions as a whole for the project. This led to the feature being hidden away from most users who didn’t seek out this specific functionality. Also entering suggestion mode worked by clicking a link that said, “Activate suggestion mode”, which was not very nice to look at and didn’t integrate with QDAcity’s overall design. (See Figure 1 Old Coding editor. Suggestion mode off vs on)

The goal of this thesis was adding the core analysis workflow of the coding editor as suggestions and to polish up the UI to make it feel more refined. This was done in collaboration with a supervisor from QDAcity. A lot of the requirements came from him since there was no user feedback to go off of.

[1] https://qdacity.com/
[2] https://getthematic.com/insights/qualitative-data-analysis#what-is-qualitative-data-analysis

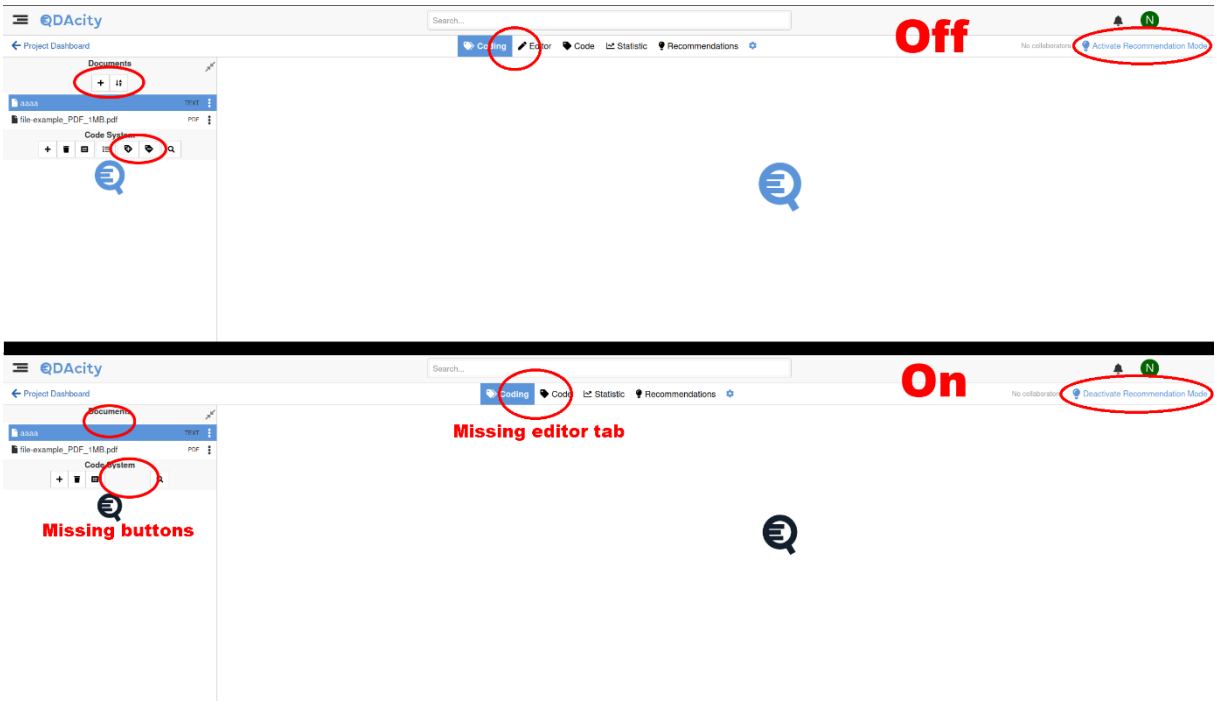

Figure 1 Old Coding editor. Suggestion mode off vs on

1.2 Thesis Structure

This thesis qualifies as an engineering thesis[3] where the student "performs and presents traditional engineering work". In the following I lay out the general structure of the thesis and state the topic of each chapter.

Chapter 1 starts out with a brief explanation of qualitive data analysis and presents the current state of the suggestion system

Chapter 2 defines the features to be implemented and their specific requirements.

Chapter 3 lays out the existing architecture of both QDAcity as a whole and the suggestion system.

Chapter 4 dives into the design and implementation of the features and requirements defined in chapter 2.

Chapter 5 evaluates the success of the project based on the requirements and notes some shortcomings or features that didn't make it into this iteration.

Finally, chapter 6 draws conclusions from the project and its success and points out future work that needs to or could be done.

[3] https://oss.cs.fau.de/theses/structure-content/

2 Requirements

2.1 Comparisons and inspiration

Before working on new features, it is good practice to look at the current state of the art and make comparisons between them to get inspiration for your own features. This gives you insights into what you should focus on, what might or might not be important for your application, and what things to look out for when developing your feature.

The two main applications that were used for inspiration were Google Docs[4] and Adobe Reader[5].

Google Docs is a cloud-based web application for editing and collaborating on text documents. They offer a suggestion system where you can edit, write or delete text and have it be a suggestion rather than directly editing the text. Suggestions are shown on side of the document where they can be commented on and rejected or accepted. (Figure 3)

Adobe Reader is a PDF Viewer that lets you view PDFs, fill forms, sign them and much more. Suggestions in Adobe Reader take the form of comments. Due to Adobe Reader not supporting a lot of direct editing features, comments are mainly used for information exchange in collaborative environments. They aren't necessarily tied to any action and serve more as a communication channel between people exchanging documents. (Figure 2)

The main takeaway I got from Adobe Reader was the sidebar. Having a dedicated comments sidebar is a really nice way to keep it out of the way for users that don't want to engage with the comments system. It also allows you to read a document without having the comments always on your screen.

Google docs does this in a similar way, although the distinction between sidebar and main document is not as clear. Also, the sidebar is enabled by default and to disable it you must click "view -> comments -> Hide comments", which are a lot of actions to do a quite simple task. They offer different modes of the sidebar though, like a smaller mode where the suggestions are small icons and not as intrusive. The sidebar in Google Docs is also very responsive. It follows your scrolling on the page and snaps the right suggestion next to the suggested text when you click on either the text or the suggestion itself. Overall, it felt really good to use and was my main inspiration for the implementation of the QDAcity version of such a sidebar.

[4] https://workspace.google.com/intl/de/products/docs/
[5] https://get.adobe.com/de/reader/

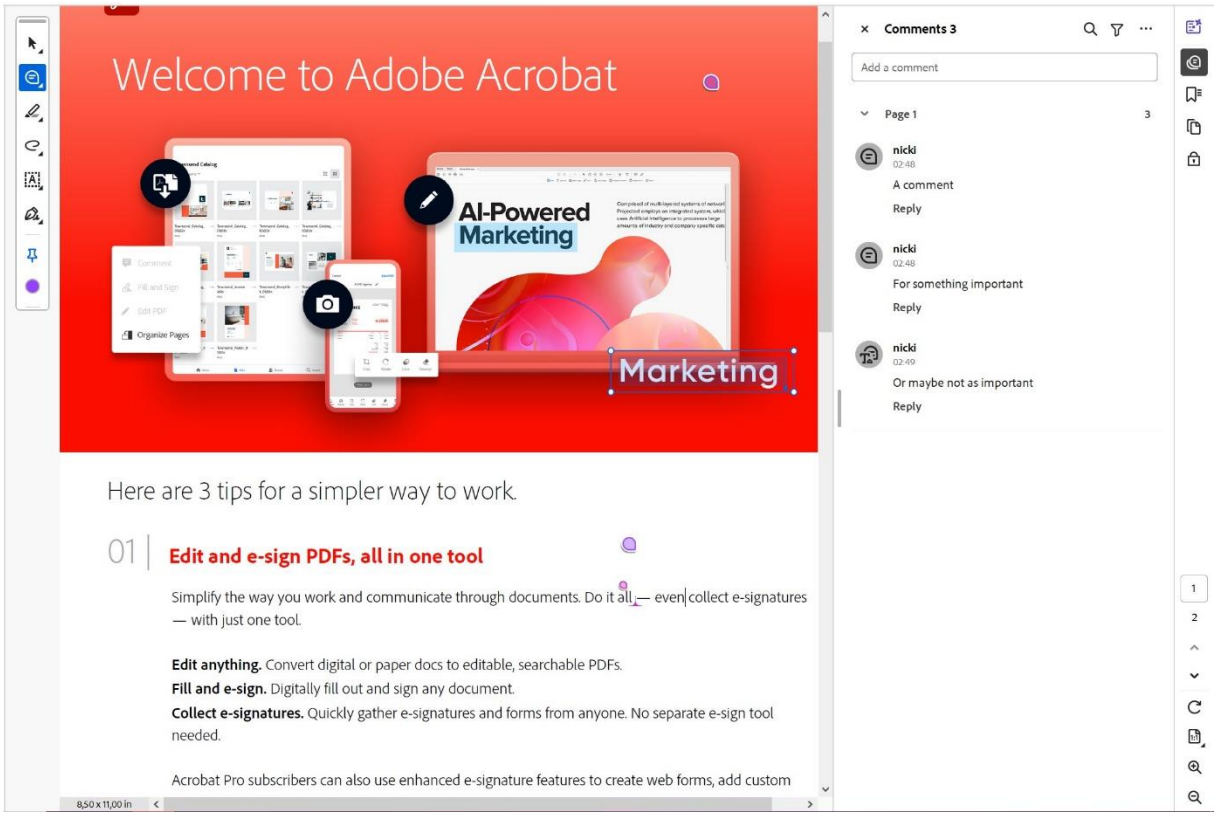

Figure 2 AdobeReader, Document with comments

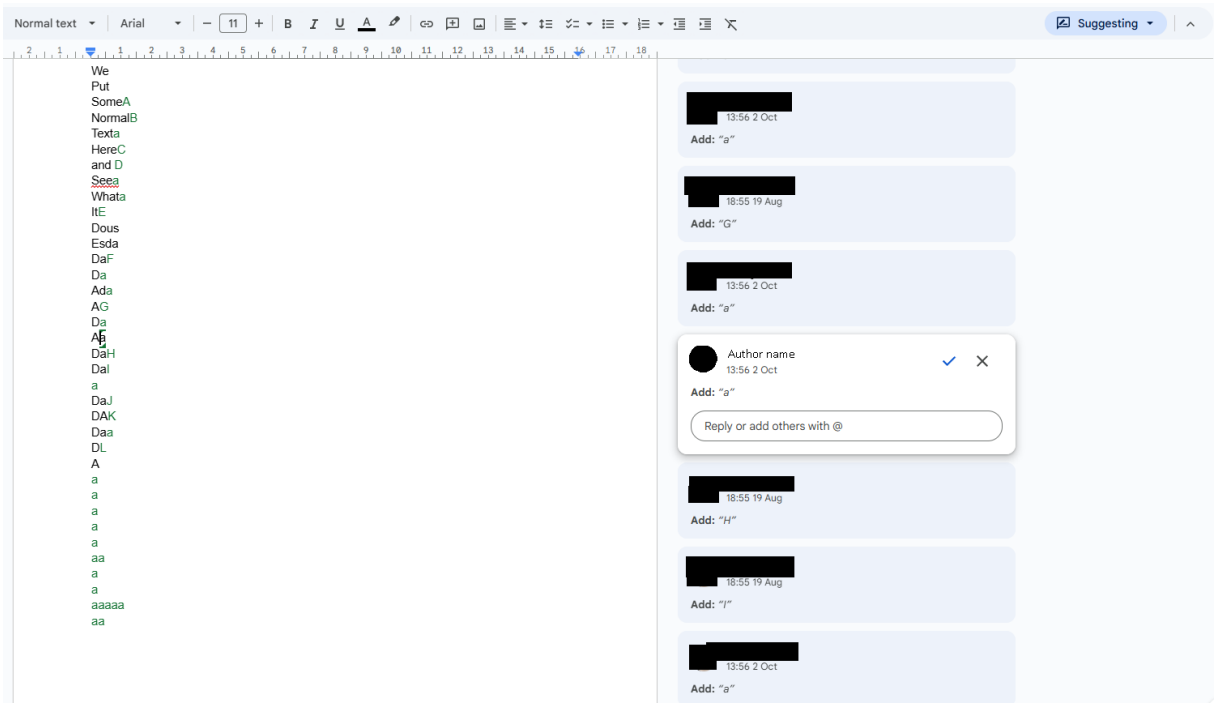

Figure 3 GoogleDocs, Document with suggestions

2.2 Defining goals

The goal of this thesis was to improve the suggestion mode. In a meeting with a supervisor from QDAcity, ideas were shared and discussed, and it came down to these general feature goals this thesis should implement:

- Document and Folder suggestions
- Coding suggestions
- A sidebar for working with coding suggestions
- Changing the steps needed to enter Suggestion mode

With all of these in place suggestion mode should become a default feature user can work with.

2.3 Requirements

This chapter defines the functional and non-functional requirements that emerged from the general goals defined in chapter 2.2. They are used later in chapters 5 and 6 to draw conclusions about the success of the project. These requirements follow the MoSCoW[6] syntax for natural language definition of requirements.

Requirements are split up into “Must have” – “Should have” – “Could have” and “Won’t have” groups. “Must have” requirements are mandatory, the system won’t be functional without them and not fulfilling them would mean the project has failed.

“Should have” and “Could have” requirements can be differentiated by the impact they have on the user experience. “Should have” requirements are important to the user’s experience, working without them would lead to a significantly worse experience so they should be prioritized.

“Could have” requirements improve the user experience, but it’s not as detrimental to leave them out.

“Won’t have” requirements are not developed in this current iteration. They are used for documenting known requirements and help keep the development focused and scoped correctly.

2.3.1 Functional Requirements

Functional requirements are requirements that are directly related to functions and services the system provides. They can also describe certain reactions the system should take to impulses or what it should not do. In the following these requirements are split into groups depending on what feature they apply to:

Code suggestions:

FRQ-1 The system must be able to create suggestions for adding codes

FRQ-2 The system must be able to create suggestions removing codes

FRQ-3 The system could be able to create suggestions for moving codes

FRQ-4 The system could be able to create suggestions for editing a codes properties

[6] https://www.agilebusiness.org/dsdm-project-framework/moscow-prioririsation.html

FRQ-5 The user could be prompted to make an initial comment when creating code suggestions

Coding suggestions:

FRQ-6 The system must be able to create suggestions for adding codings

FRQ-7 The system must be able to create suggestions for removing codings

FRQ-8 The user should not be prompted to comment on coding suggestions when creating them

Coding suggestion sidebar:

FRQ-9 The system must provide a coding suggestion sidebar which shows all coding suggestions for the current document

FRQ-10 The suggestion sidebar must provide a way for the user to correlate a suggestion with a coding

FRQ-11 Suggestions inside the coding suggestion sidebar should be interactable, meaning the user should be able to accept, reject and comment on them

FRQ-12 The system could also provide a smaller version of the coding suggestion sidebar, that only shows small icons for suggestions which are not interactable

Document suggestions:

FRQ-13 The system must be able to create suggestions for adding documents

FRQ-14 The system must be able to create suggestions for removing documents

FRQ-15 The system could be able to create suggestions for moving documents

FRQ-16 The user could be prompted to make an initial comment when creating document suggestions

Document folder suggestions:

FRQ-17 The system must be able to create suggestions for adding document folders

FRQ-18 The system must be able to create suggestions for deleting document folders

FRQ-19 The system could be able to create suggestions for moving document folders

FRQ-20 The user could be prompted to make an initial comment when creating document folder suggestions

Suggestion creation:

FRQ-21 Creating an add-type suggestion must create both the object it suggests and a suggestion for that object

FRQ-22 Creating a remove-type suggestion must not remove the object before the suggestion is accepted

FRQ-23 Creating a relocate-type suggestion must not relocate the object before the suggestion is accepted

Accepting and rejecting suggestions:

FRQ-24 Upon accepting an add-type suggestion, the visualization should be updated to show the object is no longer a suggestion

FRQ-25 Upon rejecting an add-type suggestion, the suggested object should be removed from the system

FRQ-26 Upon accepting a remove-type suggestion, the suggested object should be removed from the system

FRQ-27 Upon rejecting a remove-type suggestion, the visualization should be updated to show the removal of the object is no longer suggested

FRQ-28 Upon accepting a relocate-type suggestion, the object should be relocated, and visualization should be updated

FRQ-29 Upon rejecting a relocate-type suggestion, the object should stay at its original position and visualization should be updated

Suggestion validation:

FRQ-30 The system should validate that object a suggestion is targeting still exists

FRQ-31 The user should only be able to accept valid suggestions

FRQ-32 The system should always provide a valid state between suggestions and non-suggestion objects

Reviewing process:

FRQ-33 When reviewing suggestions the user should be able to see what the suggestion does

FRQ-34 Accepted and rejected suggestions should be documented to give the user a history of what has been done

FRQ-35 Documented suggestions should keep their comments to paint a fuller picture of why decisions were made

FRQ-36 The system won't provide the ability to search suggestions by text included in their action or comments

Other editor suggestions:

FRQ-37 The system could be able to create suggestions for editing text documents

FRQ-38 The system won't be able to create suggestions for the codemap editor

FRQ-39 The system won't offer suggestion features outside the coding editor

2.3.2 Non-Functional Requirements

Non-functional requirements include all other requirements such as constraints on the system, the development environment or ones imposed by standards. Since non-functional requirements are usually not directly tied to certain functionality, the split is done over their priority.

Must have:

NFRQ-1 Add-type suggestions must be identifiable by a consistent green color

NFRQ-2 Remove-type suggestions must be identifiable by a consistent red color

NFRQ-3 Suggestion mode must adhere to the QDAcity color scheme

Should have:

NFRQ-4 The user should be able to review suggestions inside the coding editor within one action

NFRQ-5 The naming scheme should be consistent within the codebase and user-interface

NFRQ-6 The visual differentiation of suggested and not suggested items should be possible for colorblind users

NFRQ-7 Responsive design of components should be respected. Suggestion mode specific components should remain usable with all of QDAcity's supported screen sizes.

Could have:

NFRQ-8 New codebase functionality could provide concise error logging directly stating the problem

NFRQ-9 Memoization could be used to improve performance and reduce re-renders

NFRQ-10 Memoization could be used to reduce API calls and improve performance

Won't have:

No non-functional won't have requirements were defined.

3 Architecture

3.1 QDAcity

QDAcity is a cloud-based web application consisting of a backend, frontend and CES service as well as a service for automated transcriptions, which is not relevant to this thesis and therefore omitted from the overview.

- **Backend:** The backend is built in Java with the spring boot[7] framework upon Google's App Engine Framework. It works with Google Datastore[8] as a database, Google Cloud Storage[9] to store files and documents, and many more APIs. Also notable is that custom API calls are also routed through the Backend.
- **Frontend:** The frontend is built in JavaScript and TypeScript using the React[10] library. There is an ongoing effort to move JavaScript files over to TypeScript due to its benefits like code completion and type checking. At the beginning of this thesis the suggestion system was written purely in JavaScript but some parts of it have been moved over to TypeScript. The frontend is also where most of the engineering work was done.
- **CES:** The "Collaborative Editing Service" is what enables the users to collaborate on documents at the same time. It handles synchronization of all clients and data via yJS[11], a CRDT[12] system for building collaborative applications. The yDoc, a part of yJS's system stored inside cloud storage, is also the place where the suggestions are stored. Hocuspocus[13] is used as a Webserver to handle client connections and uses the Redis extension[14] to scale horizontally. Apart from yJS the details of Hocuspocus and Redis are not relevant to this thesis.

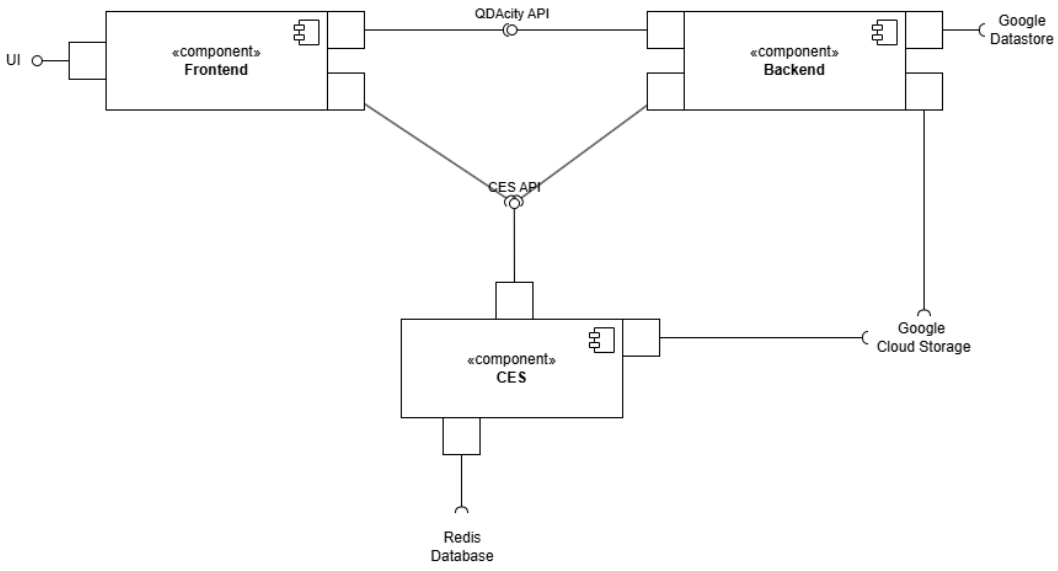

Figure 4 QDAcity architecture

[7] https://docs.spring.io/spring-boot/index.html
[8] https://cloud.google.com/products/datastore
[9] https://cloud.google.com/storage
[10] https://react.dev/
[11] https://yjs.dev/
[12] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type
[13] https://tiptap.dev/docs/hocuspocus/introduction
[14] https://tiptap.dev/docs/hocuspocus/server/extensions/redis

Work in QDAcity is always done within the scope of a project. A user can invite coworkers to the project, configure certain settings, create project revisions and much more. Inside a project is the coding editor, where you can upload documents, create codes inside the codebook and add codings to documents. The coding editor itself offers different editor views e.g. for editing text documents directly, evaluating statistics or reviewing suggestions. Almost all work was done in this coding editor so the other parts of QDAcity are only be mentioned should they come up.

Figure 5 Coding Editor with annotations

Specific systems inside the coding editor that were worked on are explained in the design and implementation section giving a brief overview and describing relevant parts of them.

3.2 Frontend

Since most of the engineering work was done in the frontend, some important features of the React library will be laid out. I will also go into some of the systems that were implemented in QDAcity's frontend that are relevant to this thesis.

A core principle of React are components: Individual pieces of code that combined form the application. A component is just a function that returns some JSX markup, a special kind of React markup code, that can then be rendered by React. Reacts calls this component function each time it renders it.

React also offers a lot of inbuild functionality for these components. These usually come in the form of "Hooks", special functions provided by the library.

Since components are just functions, data inside them is not inherently persistent between function calls. For this React offers functionality called State[15]. State is created with the useState hook which creates the starting value for some variable and a setter function usually named setX. For example, an array of suggestions as state would involve the "suggestions" array, and a "setSuggestions" function. In React, State is preserved between renders and allows components to hold data in between individual function calls. Managing state is an important part of working with React because each time state changes a re-render is done.

React causes re-renders based on an event system. There are different events that can cause a component to re-render but when it does, it re-renders both itself and its children. Depending on what each component does to render itself, this can have a big impact on performance. React offers a plethora of other functions to help with re-rendering, storing data between renders and other performance improvements but those will also be talked about if they come up.

In React, information always flows down, from parent component to child components. This information takes the form of function parameters of each component called "props". In certain cases, this can cause something called "prop-drilling" though, where some variable is passed down through a high number of components to arrive at its desired component. To avoid this React offers what is called Context[16]. Context offers a component the ability to access data from a non-direct parent component. Context is provided by a "Provider" and then consumed by a "Consumer", with the provider being a parent component and the consumer being any child component wrapped inside the Provider.

Some hooks also offer an optional dependency array parameter. For example, useEffect is a hook that runs each time a value in its dependency array changes and with it you can create event driven execution flows. Different hooks may react differently to changes in their dependency array.

React provides many more Hooks for all kinds of use cases, but these will be talked about should they come up as well.

[15] https://react.dev/learn/managing-state

[16] https://react.dev/learn/passing-data-deeply-with-context

3.3 Suggestion system

QDAcity makes use of the features provided by React. Context functionality is used in the form of several Context Providers at a top level, offering all components the information they need. One of these is the Suggestions Provider, used for managing suggestions in a project. It offers many functions to interact with suggestions and offers an array of open suggestions to its consumers.

Suggestions themselves are JavaScript Objects that contain many fields. Of note are:

- Id: used for identification of the suggestion
- Status: whether the suggestion is open, has been accepted or rejected
- Comments: an array of comments on that suggestion written by users
- Action: another object containing fields:
    - TargetType: The type of Object this suggestion targets
    - Type: The action type of this suggestion
    - TargetId: The id of the object it targets. This can be used together with the TargetType to identify the Target Object
    - Arguments: A JavaScript Object containing arguments the suggestion provider will need to properly execute the action
    - ContextData: Arbitrary data used for giving the suggestion context. As an example, the name of a document to be deleted
    - Status: The status of the action, whether it's valid or invalid

When creating a new suggestion, the component creating the suggestion starts off with the action, its TargetType and Type, its TargetId and ContextData and any arguments tied to its specific case. The suggestion provider then attaches other general data to it, like its creation date, author, etc. and uses a Factory design to create an ActionController depending on the action type of the suggestion. An ActionController implements 3 main functions:

- setupDependencies, called when a suggestion is set up (created)
- removeDependencies, called when a suggestion is removed (rejected)
- apply, called when a suggestion is accepted

It also has a hasDependencies function which determines whether or not the dependency functions should be called. The hasDependencies function just returns true or false and acts more like a getter for a Boolean value. Each suggestion type has its own ActionController to define what happens when it is created, accepted or rejected. Since we are working with just JavaScript Objects the abstraction process is done through differentiating on the action's TargetType and Type.

For example, an add coding suggestion would have TargetType: CODING and Type: ADD that would lead the ActionControllerFactory to create an AddCoding ActionController.

This all means that to add a new type of suggestion all a developer needs to do is start off with a suggestion action, specify the Type, TargetType and any arguments needed, create the appropriate ActionController and the path to create it inside the factory and he is done.

Suggestions are stored inside a yMap[17], a yJS data structure. This yMap is fetched from the projects yDoc and a local State variable is created for the suggestions array. Whenever the yMap changes, by a collaborator or the user himself making changes to the suggestions, this process would repeat updating the local suggestions and causing a re-render for any components depending on it. When a suggestion is created it is also inserted into the yMap via an Upserter, a system developed by QDAcity to abstract away from working directly with yJS. This upsert causes an update in the yMap for collaborators and keeps everyone in sync.

Visualization for suggestions is mostly done via color. QDAcity defines a theme.js file containing all the colors used throughout the application. For suggestions “accept” is currently green and “reject” is red. In addition to colors suggested objects also have a small light bulb icon next to them. This icon is used at multiple places to signify suggestion mode and is clickable to directly reveal all suggestions for that object.

Another component used throughout the suggestion system is the SuggestionDialogProvider component. It’s the entry point for using modals[18] to create or review suggestions. The clickable suggestions icon also uses the openReview function of the Dialog Provider to show the suggestion review. Besides openReview the Dialog Provider offers the openCreateSuggestion function. It allows you to pass in your suggestion action, an onSubmit function, an onClose function, and define input fields as well as add comments. The input fields are an array of fields you can define, which will then be passed into the onSubmit function together with the comments. An input field is an object with a name and a label. The name is what the variables name will be, and the label is what the modal will use to name the input field. This way you can take any function for your suggestion and pass it through this modal to add extra fields and let the user comment on the suggestion he is creating. This modal was already used for the creation of codes to let the user enter the name of the code to be created. In this thesis it will be used again for the document folder suggestions, where the reason it is used is also further explained.

The SuggestionReview component is the user’s interface for a certain suggestion. It displays the author, creation date and up and down vote buttons of the suggestion. Right under those is the SuggestionAction, a custom message that is built depending on the suggestion’ action type. It uses the actions context data to build a message for displaying what this suggestion does. As an example, for a suggestion to add a code called “Good screen”, the action message would be “Add code Good screen”. Under the SuggestionAction the comments are displayed together with an input field for new comments. At last, the SuggestionReviewToolbar is placed at the bottom, containing the accept, reject, delete and optional button to toggle whether or not comments are shown. The Suggestion Review takes in a default showComments prop. If comments are shown by default no comments button is displayed.

In addition to the modal used by the SuggestionDialogProvider, the SuggestionsEditor can also be used to review suggestions. The SuggestionsEditor is a dedicated editor tab for reviewing and interacting with suggestions. It renders a list of all suggestions, which, when clicked, reveal a SuggestionReview component. Suggestions inside the editor are also split by their status, open, accepted or rejected. Since accepted and rejected suggestions are also shown here, the SuggestionsEditor can also be used for documentation purposes.

[17] https://docs.yjs.dev/api/shared-types/y.map
[18] https://en.wikipedia.org/wiki/Modal_window

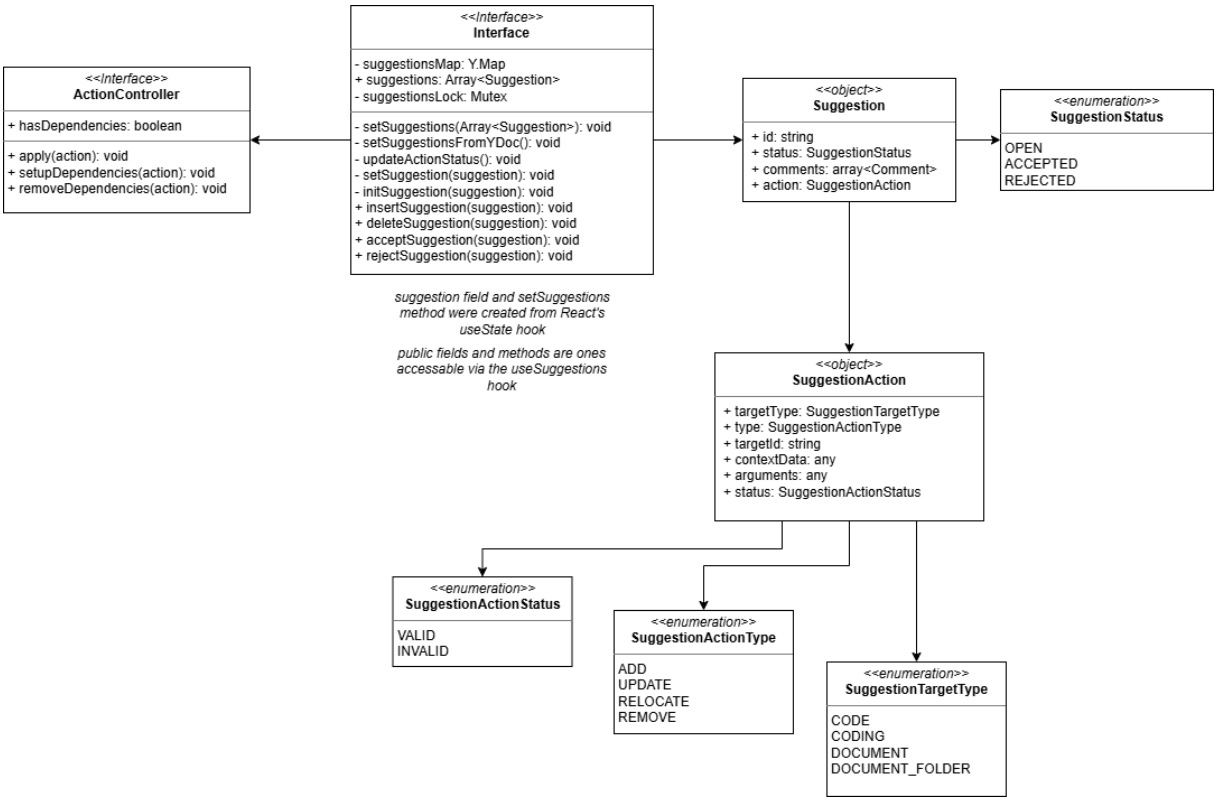

Figure 6 Suggestion system

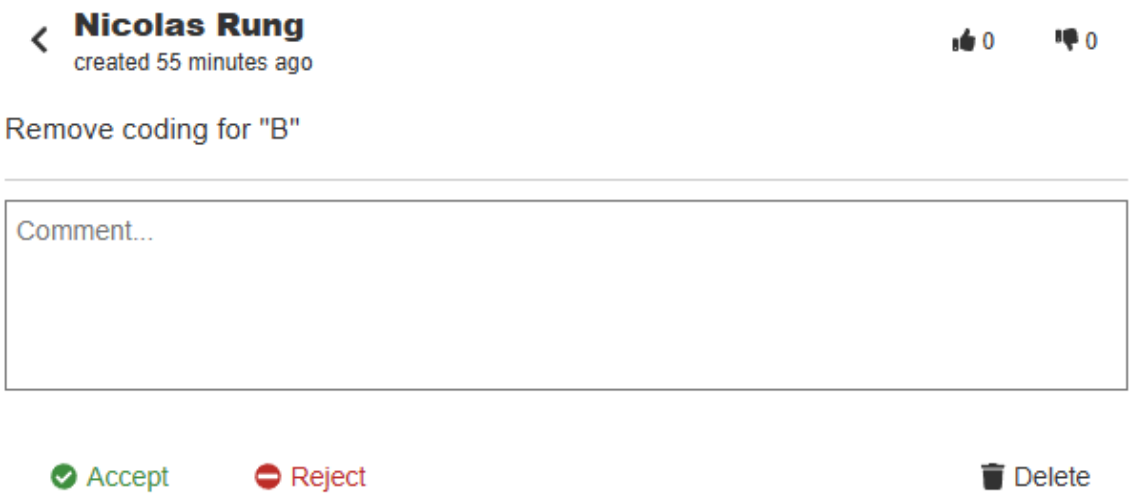

Figure 7 SuggestionReview component

4 Design and Implementation

This chapter covers the design and implementation of new features for the suggestion system. The actual engineering work was not done component by component, but rather as needed. If a new feature needed systematic changes to function, the system changes were made at that time. Still, for the sake of readability and providing a structured overview over the changes, it is now divided into 4 main sections:

- System improvements
- Coding Suggestions
- Document and Folder Suggestions
- Coding Suggestion Sidebar

System improvements will cover general improvements made to the suggestion system, specifically the suggestions provider, that were not directly linked to a single feature.

The other sections will cover the work done on the specific feature and any system changes that happened in correlation to them. Suggestions for codes were already present before this thesis and no changes were made to them, so they aren't listed here.

Each section will describe the final state of the feature and will only talk about stages in the development process if they are relevant to illustrate major problems that occurred or why certain design decisions were made.

Since most of the features to be implemented are completely new, there cannot be a section about their state before this thesis. An exception to this is the System Improvements section, which will also describe the state of the suggestion provider at the start of the thesis to further highlight the reasons for the changes made.

4.1 System improvements

4.1.1 Validation of suggestions

As FRQ-30 stated, suggestions should be validated. The reason for this is that only suggestions that would lead to a correct state should be able to be accepted. This way you couldn't have elements that are suggestions with no suggestion for them, or suggestions with no element.

At the beginning of this thesis a system for this was already in place. A function to validate suggestions, called the setActionStatus function. It was adequately named as it sets the suggestion action's status property to a value of the ActionStatus enumeration, VALID or INVALID, depending on checks that it performed. At first this function only checked ADD, REMOVE and RELOCATE suggestions for codes, but as more features got added to suggestion mode it was expanded to handle coding, document and document folder suggestions.

It now works by using switch statements over the actions Type and TargetType to decide on the specific checks it should do. Usually, these checks involve checking whether the object pointed at by TargetId still exists.

Inside the SuggestionProvider, a useEffect is used to detect changes in any of the suggestion dependencies, like codes, codings, documents etc.. Should a change occur, the setActionStatus function is called to validate the suggestions.

Before this thesis, the useEffect worked by using a single setSuggestions call to update the suggestions with a new array consisting of all suggestions after they went through setActionStatus. This also caused a re-render since the suggestion's state was updated using setSuggestions. setActionStatus directly modifies the suggestion object it gets, which does not cause a re-render, so the call to setSuggestions was necessary. This also meant that an update to the suggestions themselves could not cause a revalidation, because if it did it would trigger the useEffect again which would cause an infinite loop.

The suggestion's action status only existed in the local state at this time, not in the yDoc suggestion elements. This version of the useEffect would incorrectly validate suggestions later during development, which happened due to a Stale-closure error described later in 4.1.3.

4.1.2 Fetching of suggestions

Fetching suggestions in the SuggestionProvider was already briefly explained in chapter 3.3 but will now be explained in a bit more detail to later highlight the problem it had.

Suggestions are stored inside the suggestions yMap. A useEffect hook is used in combination with an observeDeep[19] yJS function to call the asynchronous setSuggestionsFromYDoc function, each time the suggestionMap changes. This means that each time the component is mounted or unmounted it registers the observeDeep function and if the suggestions yMap is changed, the setSuggestionsFromYDoc function will be called. The setSuggestionsFromYDoc function itself works as follows:

First it creates a suggestion array out of each suggestion in the yMap. It then makes a call to the backend to fetch UserInfos from the SuggestionsEndpoint, which includes information about the author, comments and the last update of a suggestion. This information is then mapped back into the suggestions array, setting the author of the suggestion and any comments written for it, as well as the seen flag, whether or not a certain user has seen the suggestion. Finally, it calls setSuggestions to update the suggestions state.

4.1.3 Stale-closure problems

The problem with validating was that sometimes suggestions would be evaluated as invalid when they should have been valid. This happened because all the dependency objects, like codes, codings etc., were not correct when the setActionStatus function was called. While race conditions were the first thing to come to mind, JavaScript is inherently single threaded, and await statements only pause the current execution when a yield statement is used. The actual reason for the absence of the dependencies was a state closure problem. Any function that uses callbacks like useEffect or observeDeep creates a closure of the function it wants to call when it is triggered. A closure is a combination of the function and references to its environment it uses. A simple way to explain this is with the useCallback hook from React.

useCallback can be used to create a callback function which is recreated on changes in its dependency array. When the function is created, it saves copies of all the references it needs, meaning if the function used some local state array it would save a reference to that array. Should the array now change but the useCallback has no dependency on it, the function would still be using the old reference it had saved which has the old state. This was exactly what was happening with the setSuggestionsFromYDoc and the useEffect. setSuggestionsFromYDoc was created by the observeDeep call and this version of the function only had references to the codes, codings, etc. at that time, which were just empty on the first creation. Since this observeDeep never updated, unless you left the coding editor as a whole, the references were

[19] https://docs.yjs.dev/api/shared-types/y.map#observing-changes-y.mapevent

also never updated. As a result, a setActionStatus call always used the old references and returned invalid. useEffect works in the same way but since it had dependencies on the codes, etc. it updated its references whenever they changed. That was why the useEffect always worked correctly but the setSuggestionsFromYDoc didn't.

The fix for this issue now works like this: setActionStatus uses a useCallback hook and recreates itself when any of the dependencies change. The useEffect and observeDeep now also recreate themselves when setActionStatus changes, taking care of their dependencies. The action status is now also upserted directly into the yDoc.

4.1.4 Rename and migration

Before this thesis the whole suggestion system was not called suggestion but rather recommendation system. The naming scheme was a bit misleading though since in a collaborative environment one would make suggestions rather than recommendations. Because of this a renaming had to be done.

To keep the naming scheme coherent throughout the user and developer experience, all functions, variables and classes had to be renamed. Also, all folders, files and imports had to be adjusted. The biggest engineering task of this renaming was the need for a migration though. For projects that used the suggestion system before the renaming, all old suggestions and database fields needed to be migrated as well. This migration can be split into two parts, the datastore migration and the yDoc migration.

In general migrations in the QDAcity run in the backend. The migration endpoint triggers a migration job, which then creates a bunch of new task jobs that are pushed to a low priority queue to not slow down production.

The datastore migration involved the project object's "recommendationServiceEnabled" and "recommendationEditorEnabled" fields. These two Booleans inside the Project's datamodel defined whether the suggestion mode was enabled at all and whether the SuggestionsEditor was enabled. The recommendationEditorEnabled was simply renamed to suggestionEditorEnabled and its value was copied over. Since the suggestion mode should be on by default after this thesis, the migration always set the recommendationServiceEnabled, or now suggestionServiceEnabled, field to true. The button to active/deactivate this feature had also been removed from a project's settings menu, since in the future it should become a payment tier's bonus feature.

Since the migration happened in the backend, but the backend has no access to yJS or the yDoc though, a call to the CES service was necessary. The CES SuggestionsEndpoint would be called with the projectType and Id to be migrated.

The yDoc migration was fairly simple too. It involved creating a copy of all the suggestion objects stored in the old "recommendations" map and then upserting them into a new "suggestions" map. The "recommendation" field on the codings also had to be renamed to "suggestion". This was done by simply getting the value from the yMap, copying the old value to a new key called "suggestion" and removing the old "recommendation" key.

4.2 Coding Suggestions

4.2.1 Editor Plugins

QDAcity uses a Plugin design for the different types of documents inside the coding editor. For each supported document type a corresponding plugin is selected. As of writing this thesis, only the PDFEditorPlugin and the TextEditorPlugin exist. The Plugins allow the other part of the coding editor to work independent of what document editor is used. In the case of codings the PDFEditor supports both Text and Area Codings while the TextEditor only supports Text codings.

The process of creating a normal coding involves first having some area or text selected. This is handled by different functions inside the corresponding Editor. When the addCoding button inside the codebook is clicked, a so called codingBatch object is requested from the EditorPlugin via the addCodingForSelection function. QDAcity supports batch operations for deleting and creating codings, meaning a codingBatch is an array of codings to be applied. With that codingBatch the sendCodingBatch function of the CodingProvider is called which differentiates on the batchItems BatchProcessType between CREATE, UPDATE and DELETE operations. Depending on that it upserts the coding into the codings yMap.

During this thesis I refactored different parts of this process, modifying only small parts of it to allow suggestions to create, update or deleting codings as needed.

4.2.2 Generic coding suggestion

This section describes the process of creating a coding suggestion. This process is the same, independent of which editor plugin was used. The following sections 4.2.3 and 4.2.4 will then go into detail about the differences between them.

The first step needed to create coding suggestions was to add the buttons to create a coding suggestion. Suggestion mode swaps out the entire codebook toolbar with its own components, so addCodingSuggestionButton and removeCodingSuggestionButton components were created. These components already create the first part of the suggestion, the action, with a Type, TargetType, TargetId and ContextData.

In the following I will describe the process of creating an addCoding suggestion, but a removeCoding suggestions works in almost the exact same way by using the remove-type counterparts of the add-type functions. Figure 8 Diagram showing the creation of an addCoding suggestionshows a diagram of this process.

The button’s onClick function calls the addCodingForSuggestion function from the PluginRegistry and adds the codingBatch to the action arguments. After this the component calls the addCodingSuggestion function which was given to it via the onSubmit prop from the Codesystem component. The Codesystem component also houses the addCodeSuggestion function, so it seemed fitting to put the addCodingSuggestion function in it as well.

addCodingSuggestion first checks if a codingBatch is present, since a suggestion with no codings shouldn’t exist. It then applies the correct styling to the coding in the form of a suggestion field which is a value of the SuggestionActionType enumeration. Finally, it calls the insertSuggestion function of the SuggestionProvider, which creates the suggestion from the action, calls the appropriate setupDependencies function and upserts it to the yMap.

The codings themselves needed to be updated to allow for different styling of their coding brackets. The brackets of suggested codings are a dotted instead of a straight line and are colored depending on if they are an ADD or REMOVE suggestion. This was done using CSS and the styled library, which is used all throughout QDAcity. The color was again defined by the Theme.js file.

The AddCoding's ActionController's setupDependencies function calls the sendCodingBatch to ensure a coding is only created if the suggestion was also created. The apply function simply updates the coding, setting the suggestion field to null to update the visualization. The removeDependencies function handles rejection and changes the codingBatch type to BatchProcessType.DELETE. This then removes the coding when sendCodingBatch is called.

For the RemoveCoding's ActionController the functions are a bit different. The BatchProcessType already is DELETE so the setupDependencies only changes the visualization. The apply function calls sendCodingBatch to remove the coding while removeDependencies sets the visualization back to normal.

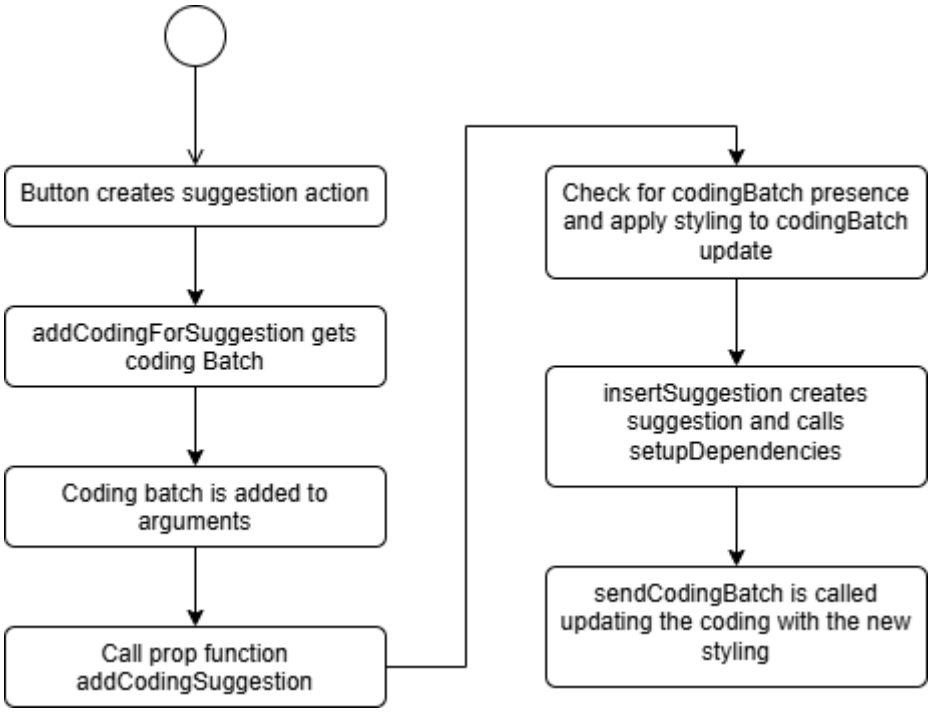

Figure 8 Diagram showing the creation of an addCoding suggestion

4.2.3 PDF Editor codings

For the PDF editor the generic process worked pretty well. The difference of Area or Text codings was entirely handled by the PDFEditorPlugin and all necessary functions needed when accepting or rejecting coding suggestions could be called from within the ActionController. No further changes were needed.

4.2.4 Text Editor codings

The generic process didn't work for Text Editor coding suggestions though, due to some differences in how they function.

QDAcity offers a built-in text editor for editing text documents directly inside the application. For this they use the Slate Text Editor[20] framework in combination with the slate-yjs[21] package, as editing is also synched for collaboration. This is important as text editor codings are attached to so called positions inside the editor. Using the positions codings will move

[20] https://github.com/ianstormtaylor/slate
[21] https://github.com/BitPhinix/slate-yjs

automatically when editing the text in the document and also be removed should the text they apply to be removed.

This was a problem for the suggestion system as the editor object could only be accessed if the TextEditor component was mounted, meaning if you were on the suggestions editor, the component was not mounted, and the program couldn't access the editor and update or store positions.

Also storing the positions was done immediately in the add/removeCodingForSelection function, which also placed positions into the editor before the coding was really created. The problem this led to was that creating a remove Coding suggestion would get the coding batch for the suggestion and immediately remove the points, which removed the coding before the suggestions was accepted.

To resolve this problem the add/removeCodingForSelection function was refactored. The process of creating codings in the text editor was split into two separate functions, the old add/removeCodingForSelection function and a new applySlateTextCodingPoints function. As the name implies it handles the Slate Editors positions for codings. It is called on automatically by sendCodingBatch so a developer wouldn't have to remember to manually call it each time they want to create codings. It ignores non TextEditor codings and handles all the BatchProcessTypes, CREATE, UPDATE, DELETE.

To circumvent the problem that it could only be called while the TextEditor component was mounted, a new variable was added to copy the editor object and yDoc whenever they were created. The editor positions are then applied through these copies. A downside of this approach is that you cannot accept a text coding suggestion if you haven't loaded the Text Editor at least once.

With this refactoring done both PDF Editor and Text Editor codings worked with the new suggestion system. One last thing to mention is that in earlier versions of this feature, each time you wanted to create a coding suggestion it prompted you with a modal to add an initial comment and confirm the coding suggestion. This was later changed to just immediately create the suggestion as it was very disruptive to the coding workflow and adding a comment could always be done at a later point in time.

4.3 Document and Folder Suggestions

Documents in QDAcity are stored in the backend of the application. A document is uploaded to the backend on creation and fetched from it when opening it. The DocumentView is the component housing all documents. It renders the document toolbar, which contains all the buttons concerning documents, and all documents and document folders.

Documents are rendered by the document component. To make documents draggable inside the DocumentView and rearrange them, a DragDocument wrapper is used around them. This wrapper uses the React-DND[22] package to function.

Important components for suggestions include the FileSelectorModal, which handles creating new documents. In it you can either select a document from your device to upload or create a new empty text document. Also important are the document options, a small submenu for each document housing the delete and edit buttons.

4.3.1 Document suggestions

As the creation process is handled solely by the FileSelectorModal, where the document is created in the confirm action of it, the suggestion creation also needed to be inside of it. The onSubmit function of the FileSelectorModal is a complex function using the different EditorPlugin's upload function to handle the different file types. In an effort to keep codebase additions generic and not having suggestion specific code scattered throughout the codebase, a general submissionAction was added. It is a prop function you can give the FileSelectorModal, which is called with the uploaded document as a parameter. For the document suggestions this function was the addDocumentSuggestion function, which creates the appropriate suggestion action and calls the SuggestionProvider to create the suggestion. For this to work the FileSelectorModal needed to be refactored though to return the uploaded document.

Deleting a document is handled by the deleteDocument function of the DocumentsProvider. Adding suggestion functionality was fairly easy as it required no additional refactoring, just replacing the function that was called from the delete button. The deleteDocumentSuggestion function works in the same way as its counterpart, creating a suggestion action and calling insertSuggestion.

The ActionControllers for documents then work as follows. For addDocument suggestions the only action needed is deleting the document should the suggestion be rejected. For removeDocument suggestions the apply function deletes the document. Visualization is handled differently from coding suggestions. As documents are stored in the backend, adding a suggestion field to the object would require adding it as a database field. In an effort to keep the database objects as clean as possible and since this field is only important for the frontend it wasn't added. Instead, a useEffect was created in the SuggestionProvider to check for changes in the documents or suggestions and apply a suggestion field to documents, only in the frontend. That field is then used for rendering the appropriate color.

The withSuggestions function, which is used to render the suggestion icon for codes, directly ties into the CodeHOC[23] and couldn't be used for documents. Instead, a new generic wrapper was implemented, the SuggestionsWrapper. It takes in a renderObject and any children components. It then checks the renderObject for an id field that it uses to determine the suggestion for that object. The children components are then rendered first with the suggestion icon behind them.

[22] https://react-dnd.github.io/react-dnd/about
[23] https://legacy.reactjs.org/docs/higher-order-components.html

4.3.2 Document Folder suggestions

Folders in QDAcity are handled in a very similar way to documents. Inside the codebase they are called document groups and are identified by an isGroup field. The DocumentView simply differentiates on isGroup and calls renderDocument or renderDocumentGroup accordingly. Both functions render the document component, with the only difference being which prop functions are used. With that being said creating suggestions for documentGroups is quite different from documents though.

DocumentGroups are created using the DocumentsProvider's addDocumentGroup and deleteDocumentGroup methods. Inside the DocumentView a new addNewDocumentGroupSuggestion function was created. It, like its document and coding counterparts, also creates the suggestion action but also does something different. Normally when creating a documentGroup, after creation the name is editable, allowing the user to write the desired name and confirm to save the group. Inside the codebase this was done in a two-step process where first, the documentGroup was created, then an update call was made to change its name. Since the suggestion needs ContextData, specifically the name of the group, to display in its ActionMessage, the final name was needed before the group was created. For this purpose, the openCreateSuggestion function is used. The modal asks for the folder's name and which is then used in the addDocumentGroup function. Since it does a backend call to create the documentGroup, the function also waits for success of the group's creation before creating the suggestion. Also important was saving the documents previous parentId, since rejecting the suggestion, and therefore deleting the document group, needed to place all the documents back to where they belong.

In edit mode, deleting a documentGroup prompts you for the action you want to take with the documents inside the group. Either you delete them all or you relocate them to another group or the root node. When creating a removeDocumentGroup suggestion this prompt is also shown and the selection is added to the suggestion's arguments. For rejecting createDocumentGroup suggestions this functionality unfortunately didn't make it into the finished version. Currently it always moves the contents back to the root node.

4.4 Coding Suggestion Sidebar

The SuggestionAction of coding suggestions only states what code was used to add a coding, not which coding in the document it actually is. To give the user the ability to make this correlation, a coding suggestion sidebar was implemented.

The sidebar would be next to the document inside the coding editor and show the suggestion review for each coding that is a suggestion. Since this would be the only place where you can actually correlate a coding suggestion with its coding, you should be able to directly interact with the suggestion, accepting, rejecting and writing comments for them. For this the SuggestionReview component could be reused and just rendered at the correct position next to the coding. Here a lot of the inspiration from 2.1 will come to show.

4.4.1 Sidebar layout

The first step was to create the sidebar at all. The sidebar was supposed to sit on the right side of the document, to not interfere with the document list and codebook. Up until now the coding editor didn't support a right sidebar, so a small refactoring of the layout of the coding editor was in need. Css[24] grid-template-areas are used to define the general layout of the coding editor while the size of each area is then set by certain conditions. For example, if the suggestion sidebar is open, the sidebarRight area would take up a certain amount of space in which the CodingEditorSuggestionSidebar component would be rendered.

4.4.2 The sidebar component

The sidebar works as follows:

First the necessary data is fetched. The suggestions from the SuggestionProvider and the codingBracketData via the PluginRegistry's getCodingBracketData function, which was also created during this thesis since the codingBracketData is stored as state by each editor themselves. codingBracketData itself is an array of CodingBracketDataObjects elements (Figure 10). The suggestions and codingBracketData are then filtered to only include coding suggestions and their relevant BracketData. A container div called "ReviewCard" was introduced which houses the SuggestionReview component. The ReviewCards have an absolute placement position, which ideally should be exactly the same height as the codingBracket, and are rendered one after the other using a renderCodingSuggestion helper function.

4.4.3 ReviewCard placement strategy

The biggest difficulty with the sidebar was placing all the ReviewCards inside the Sidebar container, on the same height as their codings, but without overlaps. Obviously, since codings can easily overlap but the ReviewCards cannot, just simply placing them all at their desired position doesn't work. Because of this some algorithm had to be constructed to place them as close as possible to their desired position without overlapping any of them.

This problem can also be generalized like this:
Given a container with arbitrary height, place objects of arbitrary height at specific positions in the container, without overlapping any of them.

This generalization made the variables that needed to be defined clear.

[24] https://www.w3.org/Style/CSS/Overview.en.html

First, the ReviewCard container's height. Since the sidebar should scroll together with the codings it should be at least the same height as the coding brackets container. This was achieved via a querySelection of the DOM element with id="codingBracketContainer" and taking its offsetHeight. A HTMLElement's OffsetHeight[25] is the height the element takes up once rendered, including padding and borders, as an integer. If there were more ReviewCards than would fit into this space, the sidebar would need to be bigger though. So initially it is set to the same height as the codingBracketContainer, but after calculating the last ReviewCard's position, if that card would be outside the container, the container is enlarged. This can work since the positions of all ReviewCards are calculated in a preprocessing step.

A similar approach was taken for the ReviewCards themselves. When rendering them, a Ref[26] is attached to them to track their height once rendered, which is again represented by its offsetHeight. Just setting a fixed height for either the container or the ReviewCard wouldn't have worked since resizing the window would change its height, violating the responsive design requirement. Since the ReviewCard can also be expanded, to show the comments for the suggestion, its height is not always the same. Because of this, each ReviewCard has its own Ref that tracks its individual height. A downside of this approach is that the correct height would only be available in the next render call but no other way to get a DOM element's height once rendered was found.

Finally, the last step needed was finding a ReviewCard's ideal position, which should be as close as possible to the height of the coding bracket. Important data structures for this calculation are seen in Figure 10. The placement height of a codingBracket is defined by its Top[27] property, same for the ReviewCard. First was finding the coding brackets middle, via its offsetTop and height properties, which are both part of the CodingBracketDataObject. Taking the offsetTop plus the height divided by two is the middle of the coding bracket. Then using the coding bracket's middle and adding or subtracting half the ReviewCards height you can calculate the edges of the ReviewCard. The coding bracket container and the ReviewCard container both having the same starting position is a useful simplification for this calculation.

To determine if two ReviewCards are overlapping, first the position and halfHeight of each ReviewCard is stored inside a placedPositions array when it is placed. Then, when placing a new card, the edges of each already placed ReviewCard are computed from their halfHeight and placement position. The height values that lie inside the placed card are exactly the intersection between values above the placed lower edge and under the placed upper edge. So, to find out if a certain new edge is inside an already placed card, you could check if it's part of this intersection. For the algorithm it's important to note that the top property is the distance from the top of the container and it grows towards the bottom. The function looks like this:

```
function checkIfInsidePlaced(position: number): boolean {
    return position < placedLowerEdge && position > placedUpperEdge;
}
```

If the new upper edge is inside the placed one, it's an upper edge overlap and is shoved down. If the new lower edge is inside, it's a lower edge overlap and is shoved up. How much to move the position up can be calculated by the difference between the new edge and the overlap edge. Since a full overlap would mean the edges would be exactly the same, smaller instead of smaller or equal comparison was used.

In the final implementation all coding brackets are sorted by their height. This means that the only overlap that can occur is one where the current card's top edge is higher than the placed card's lower edge, so an overlap that is shoved down. This sorting simplification is necessary

[25] https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/offsetHeight
[26] https://react.dev/reference/react/useRef
[27] https://developer.mozilla.org/en-US/docs/Web/CSS/top

as it takes care of a lot of issues with placing the ReviewCards like infinite shoving back and forth trying to fix recursive overlaps. Without sorting, depending on the order of coding brackets, a coding could be shoved up to fix one overlap which would then cause another that is shoved back down, causing an infinite loop. There was an attempt to solve this issue using a backtracking algorithm, but that led to a plethora of new problems, edge cases and increased complexity and was ultimately abandoned in favor of the current solution. With sorting, ReviewCards are not as close to their desired position as with shoving them up and down, but this was an acceptable compromise since adding the functionality to select and highlight them would ease the problem this created for the user.

Some other edge cases also need to be respected. When a coding is very close to the top or bottom of the document and the ReviewCard would not fit into the container, either the container or placement position need to be adjusted. For the bottom case this is already handled by making the ReviewCard container bigger. This enlargement towards the bottom doesn't interfere with the placement of the ReviewCards above it, since the positions of the coding bracket sidebar and suggestion sidebar are still the same above it. For the top the sidebar's header needs to be respected though. The header also has a Ref attached to it and using its height a ReviewCard is shoved down should they overlap. This approach works since the header is always at the top of the container so using its height you can calculate how far down the card should be shoved.

Figure 9 Finished Coding Suggestion Sidebar

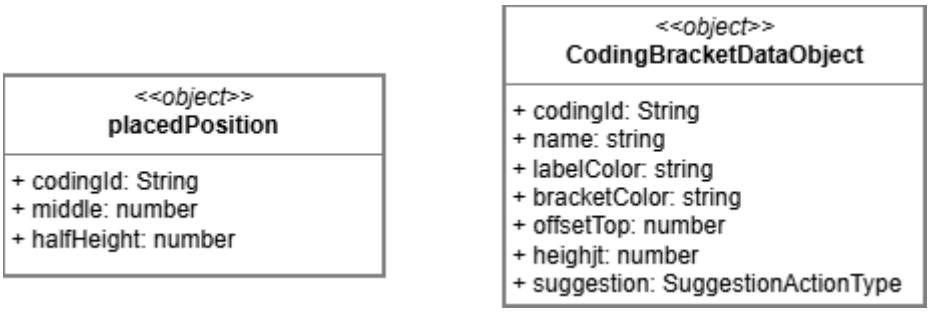

Figure 10 Definitions for placedPosition and CodingBracketDataObject

4.4.4 User experience

Engineering work was also done to make the sidebar feel very responsive and good to use.

Most notably the functionality that ReviewCards should be moved next to the correct coding once highlighted. You could already highlight a coding when clicking on it which showed the area, for pdf area codings, or the text, for text codings, which the coding was applied to. This highlighting was done via the setManuallyActivatedCoding function which was now also moved to the PluginRegistry. This way clicking either the coding itself or the ReviewCard would highlight and move the sidebar. Highlighting a coding and ReviewCard also causes a slight color shift in the background color of the ReviewCard. When a coding is highlighted via the setManuallyActivatedCoding function the activatedCodingId inside the CodingEditorProvider is set. Since all plugin editors automatically call the highlight function inside the sidebar should a coding be highlighted in any way, a useEffect is used to call a function in the suggestion sidebar each time activatedCodingId changes. This posed a big problem though. Since information only flows down, how can the parent now call a function inside the child object. Declaring the function in the CodingEditorProvider and handing it down to the sidebar component was not possible since the CodingEditorProvider had no knowledge of ReviewCards or the placedPositions. For this React offers a solution though, imperative handles[28]. The suggestion sidebar attaches its setReviewCardActive function to a Ref, passed down via the CodingEditor Context, which is then called from the CodingEditorProvider's useEffect. The setReviewCardActive function itself needs to calculate the correct scrollTop for the suggestion sidebar so that the right review is next to the coding. This involves converting the brackets center position to the viewports center position, then using that value to calculate the new scrollTop for the sidebar while respecting edge cases where the bracket to highlight is on the edge of the sidebar. Smooth scrolling of the sidebar was handled via the HTMLElement's scrollTo method.

Responsive design was also respected to make the sidebar as useable as possible for all screen sizes. The SuggestionReviewToolbar's css layout was changed to properly stack its buttons should they not fit into their container. Other parts of the suggestion review were changed in similar manners to not overlap and remain functional should they have insufficient space.

The Suggestion Sidebar should scroll with the document and coding bracket sidebar. The main document and the coding bracket container exist in a ResizeableTwoColumnsLayout component. The coding bracket container takes up the left column while the main document takes up the right. Scrolling is then handled by one singular event listener so scrolling is completely smooth. Simply getting the scroll container and using a useEffect to set the sidebars scrollTop to the bracket containers scrollTop was very choppy and made scrolling feel bad to use. Instead, an event listener is used for scrolling which copies the documents scrollTop and updates the sidebar's scroll accordingly, which made scrolling a lot smoother.

[28] https://react.dev/reference/react/useImperativeHandle

This sidebar scrolling updates the sidebar ref constantly though, which causes the component to re-render a lot.

The final change made was a new way to enter suggestion mode. The original link was removed and a new editModeButton component was added. The button opens a drop-down menu where the user can select which edit mode he wants to work in, currently either suggestion or edit. The component was also made in a way that a developer can add new modes with relative ease.

5 Evaluation

This section aims to evaluate the project's success, based on the requirements defined in 2.3 . The requirements are also listed in the same way as in section 2.3 and follow the same groupings.

For evaluating the requirements, the fulfillment status is quickly stated, followed by a, mostly, short explanation of how it was fulfilled, or why it wasn't.

After evaluating the requirements, section 5.3 will evaluate the success of the project as a whole, using the priority of each requirement, defined by the MoSCoW notation, as a base.

5.1 Functional requirements

Code suggestions:

FRQ-1 - **FRQ-5** were already fulfilled before this thesis. No changes were made to their systems, and they still function as before.

Coding suggestions:

Section 4.2 covers the implementation of coding suggestions.

FRQ-6 and **FRQ-7** are fulfilled. The system now has the ability to create suggestions for both adding and removing codings.

FRQ-8 is fulfilled as well. The end of section 4.2.4 states that the commenting prompt was removed for coding suggestions to not disrupt the workflow.

Coding suggestion sidebar:

Section 4.4 covers the implementation of the coding suggestion sidebar.

FRQ-9 is fulfilled. The coding suggestion sidebar was implemented. It only shows suggestions for codings inside the current document and is able to present them all, no matter how many there are.

FRQ-10 is also fulfilled. Correlation between codings and their suggestion is implicitly done by their relative proximity when scrolling. The user can also highlight a coding to directly see which suggestion targets it.

FRQ-11 is fulfilled as well. The user is able to interact with the suggestion as usual. Accepting, rejecting, commenting and voting are all supported.

FRQ-12 is not fulfilled. The small version of the sidebar unfortunately had to be cut due to time constraints. As a could requirement its priority wasn't as high as other requirements.

Document suggestions:

FRQ-13 and **FRQ-14** are fulfilled. The system is now able to create suggestions for adding and deleting documents.

FRQ-15 is not fulfilled. As stated at the beginning of 4.3, moving documents is done via the React-DND package. Due to complications with getting a reference to which specific document was being moved, this feature was not implemented. Without this reference the suggestion wouldn't be able to move the document violating FRQ-28 and

FRQ-29.

FRQ-16 is fulfilled. The user is prompted to add a comment when suggesting a new document. This works for both creating a new text document and uploading an existing document.

Document folder suggestions:

FRQ-17 and FRQ-18 are fulfilled. The system can now create suggestions for adding and deleting document folders.

FRQ-19 is not fulfilled. This feature was also cut for the same reason as FRQ-15.

FRQ-20 is fulfilled. Getting the folders name uses the openCreateSuggestion function which also adds comment functionality. Deleting document folders also opens the modal for comments after selecting what deletion strategy the user wants to use.

Suggestion creation:

FRQ-21 is fulfilled. Creating add-type suggestions creates a suggestion and a coding, document or document folder, depending on which button was pressed. If the user aborts creating the suggestion for documents or document folders by canceling the modal, the suggested item is also deleted.

FRQ-22 is fulfilled as well. Creating remove-type suggestions doesn't remove the object before the suggestion is accepted. Removal always happens in the apply function, meaning only when the suggestion is accepted.

FRQ-23 is also fulfilled. The only relocate-type suggestion is still the relocateCode suggestion. As it hasn't changed, it still works.

Accepting and rejecting suggestions:

As a reminder: Creating calls the setupDependency function. Accepting calls the apply function and rejecting calls the removeDependency function.

FRQ-24 is fulfilled. The addCoding ActionController correctly updates the visualization when its removeDependency function is called. Add-type visualization for documents and document groups is also implemented and updates automatically.

FRQ-25 is also fulfilled. The addCoding, addDocument and addDocumentFolder ActionController all remove their respective object in the removeDependency function.

FRQ-26 is fulfilled as well. The removeCoding, removeDocument and removeDocumentFolder ActionController all remove their respective object in the apply function.

FRQ-27 is fulfilled. The removeCoding ActionController correctly updates the visualization when its removeDependency function is called. Remove-type visualization for documents and document groups is also implemented and updates automatically.

FRQ-28 and FRQ-29 are also still fulfilled as no other relocate-type suggestions were added.

Suggestion validation:

FRQ-30 is fulfilled. Suggestion validation was updated to handle the new types of objects. The stale-closure problem was also fixed and validations are now always correct.

FRQ-31 is also fulfilled. The accept button is not clickable if a suggestion's validation status has not yet loaded or is invalid.

FRQ-32 is not fulfilled. One design decision allows a certain scenario that violates this requirement. If a user creates an addDocument suggestion the document is created before the suggestion is created. If the user reloads the page after the document was created, but while the openCreateSuggestion modal is still open, the document keeps existing but no suggestion for it exists. The document is then just a normal document as no persistent field marks it as a suggestion. The reason it was done this way was that currently there is no way to get a document reference before the document is created. A refactor of how documents are created in QDAcity would have been required and as such it was cut due to time constraints.

Reviewing process:

FRQ-33 is fulfilled. The suggestions ActionMessage was constantly updated to give the user as precise as possible message what the suggestion does.

FRQ-34 and FRQ-35 are fulfilled. Accepted or rejected suggestions can be viewed in a list inside the SuggestionsEditor. All suggestions keep their comment history for traceability. This functionality also already existed before this thesis and wasn't changed.

FRQ-36 is not fulfilled. As the requirement already stated it won't be fulfilled in this iteration, not fulfilling it is fine.

Other editor suggestions:

FRQ-37 is not fulfilled. Adding suggestion functionality for editing text documents using the current suggestion framework would likely be very complex. Currently suggestions are created with a certain event, like a button click. Tracking changes in text documents and constantly updating the correlating suggestion would cause constant yDoc updates and excessive API calls. A better method would need to be created.

FRQ-38 and FRQ-39 are not fulfilled. This thesis focused on the coding editor and its functionality. Other parts of QDAcity were not worked on.

5.2 Non-functional requirements

Must have:

NFRQ-1 is fulfilled. All add-type suggestions are identifiable via the same green color defined in the Theme.js file.

NFRQ-2 is also fulfilled. All remove-type suggestions are identifiable via the same red color also defined in the Theme.js file.

NFRQ-3 is fulfilled as well. Any colors used in suggestion mode components always used ones defined in Theme.js.

Should have:

NFRQ-4 is fulfilled. Using the suggestion icon the user can always directly access the suggestion review. Code, document and document folder suggestions are all supported by this. Coding suggestions can be review in one click by opening the suggestion sidebar, which is also just one button click.

NFRQ-5 is also fulfilled. The naming scheme in the UI consistently refers to suggestions as “suggestions”. Inside the codebase all instances of “recommendation” were replaced by “suggestions” and correlating components also use a consistent naming scheme, like the addCodingSuggestionButton, addCodingSuggestion function and addCoding ActionController. One inconsistency is that documents are deleted but the ActionController is called removeDocument. Keeping the naming scheme only locally consistent was decided to be acceptable here, as “deleting” documents is considered more natural terminology than “removing” them.

NFRQ-6 is fulfilled. Colorblind users are able to differentiate suggested codings from normal codings because suggested codings are also visualized by a dotted line and not just color. Suggested codes, documents and document folders can be identified by the suggestion icon which is also not dependent on color.

NFRQ-7 is fulfilled as well. Responsive design of the SuggestionReview component was enhanced. It remains usable in both the SuggestionsEditor and coding sidebar for QDAcity supported screen sizes. The xs screen size which is used for smartphones, excludes the coding editor as a whole. Since the base coding editor functionality is missing, suggestion mode as a whole is not present so it does not violate the requirement. Should a mobile coding editor be implemented, the suggestion mode features would also need to support it.

Could have:

NFRQ-8 is not fulfilled. While some codebase functionality provides concise error logging on expected conditions, not nearly all of the new functions cover enough edge cases. As a lot of the suggestion system is still in JavaScript, where type safety is not guaranteed, a lot of typing and null checks are still missing. The parts of the suggestion system that are in TypeScript, like the coding sidebar, are a lot more robust.

NFRQ-9 is not fulfilled. A lot of QDAcity components re-render a lot, with the coding sidebar being no exception. No useMemo hooks or similar were used in any of the new features as more focus went to making them work, before optimizing them.

NFRQ-10 is also not fulfilled. As just stated, no useMemo hooks were used. The SuggestionProvider still does an API call each time it updates the suggestions via the yDoc.

Won’t have:

No non-functional won’t have requirements were defined.

5.3 Project success

The majority of functional requirements were satisfied, with 34 out of 39 having been achieved. The remaining 5 had to be cut due to time constraints or only being discovered in late stages of the project. Of the non-functional requirements 3 out of 10 were not satisfied, displaying a lower success rate. Since none of the “must have” requirements failed, a complete failure of the project can be ruled out. Still the impact of the non-fulfilled requirements will now be evaluated to give a final assessment of the project’s success.

FRQ-12 was not satisfied. As a “could have” requirement, its priority was not as high as other ones and the effect on the user experience is minor.

FRQ-15 and FRQ-19 are not fulfilled. While this requirement is defined as a “could have” requirement, its effect on the user experience is bigger than FRQ-12’s. Not quite enough to be a “should have” requirement but bigger, nonetheless.

FRQ-32 has not been achieved. Being a “should have” requirement makes its failure more detrimental, but it is eased by only failing in one very specific scenario.

FRQ-37 was also not fulfilled. As a “could have” requirement its priority wasn’t that high and due to its complexity, it had to be cut from the current iteration. Editing text documents while in suggestion mode only works for suggested documents, which means the user experience is not severely impacted by this failure.

Most of the focus of this thesis was on making the new functionality work. This resulted in an overall prioritization of functional over non-functional requirements. Even so the non-functional requirements need to be respected and will now be evaluated:

NFRQ-8 was not satisfied. Default error logging is still available and for most suggestion objects, if any of the steps in the creation of a suggestion fails, the whole process fails. While this failure makes debugging a bit harder, it doesn’t have any impact on the user experience, so it is acceptable.

NFRQ-9 and NFRQ-10 were not met, indicating areas where the suggestion system's performance can be improved. Nevertheless, the overall performance impact of suggestion mode remains minor, since the number of suggestions is not expected so big that it would cause issues. Suggestions are also not created in huge amounts at a time so while some unnecessary API calls are being made, it also shouldn’t impact overall performance that much.

In conclusion: Most of the failed requirements are “could have” requirements with small or minor impact on the user experience. Just one “should have” requirement is violated but only in one specific scenario. Meanwhile all the features the project sought out to add are implemented and functional. Overall, the project can be considered a success as suggestion mode was successfully expanded.

6 Conclusions

With this thesis a lot of new functionality was added to suggestion mode, and it became a default feature of QDAcity. The user's ability to work together with his collaborators was improved and any users who sought out QDA software with suggestion functionality could become interested in QDAcity now.

While the project can be considered a success, there is still a lot of engineering work that can be done in conjunction with suggestion mode. The future work section goes over some ideas but as more users work with suggestions, more feature requirements will surely emerge.

This thesis also built on a lot of fundamental systems already existing. The system's ability to be expanded is good at the moment, which makes the addition of related features simple. Still, some changes might become necessary as other features, like text editing, are also supposed to be expanded with suggestion functionality.

6.1 Future work

While all features this thesis aimed to add are implemented, not all requirements are met. Suggestion mode can also still be expanded with a lot of new functionality, unfinished parts from this thesis can be completed and performance can be improved.

For example, validations of suggestions in correlation with fetches from the yDoc can still be improved and optimized. The current implementation still causes a yDoc update, even if we are the ones that caused it. This also causes a second validation of all suggestions, even after the suggestions were just validated before the update was made.

Another performance hit is caused by the coding sidebar re-rendering each time the user scrolls. Each re-render also causes the position calculations to run again, even if the scroll has no impact on the ideal positions of the ReviewCards. A possible solution for this would be to split the scrolling component and the main sidebar component and then using a useMemo hook to keep the sidebar from re-rendering and re-calculating all the positions. This didn't make it into the final version though due to time constraints.

The failed requirements related to already existing functionality should also be finished up. The incorrect state issue, which causes FRQ-32 to fail, could be fixed by more closely tying the document creation to the suggestion creation. This would require refactoring a number of components, like the FileSelectorModal and others. Error logging throughout the new features should also be improved.

After that new functionality could be added. Relocating documents and document folders should be added to allow users to work together on the structure of their documents. Editing text documents as suggestions like in google docs would also be a great addition. A smaller sidebar as defined in FRQ-12 could also be added but other features might take priority.

References