ETH zürich

Department of Computer Science
Programming, Education, and
Computer-Human Interaction

A Collaborative Learning Extension for Jupyter Notebook

Pascal Linder

Bachelor Thesis

2024

Supervisor: Prof. Dr. April Wang, Prof. Dr. Dennis Komm

ABSTRACT

Peer assessment as a form of collaborative learning can encourage students to learn actively and improve their learning progress. Previous research has demonstrated the potential of systems that integrate peer assessment mechanisms. In this project, we designed a variation of computational notebooks to support teaching introductory programming in the JupyterLab environment. We were particularly interested in enabling instructors to share and demonstrate coding in real-time, allowing them to assess students' notebooks at scale while providing real-time feedback, and supporting students in collaboratively assessing their code. During development, we extracted core functionality into separate libraries to enhance reusability, including a React quiz UI component library and a Yjs extension for handling normalized data. Additionally, by deploying a JupyterHub server, we facilitated multi-user access to the extension, enabling comprehensive testing. Last but no least, we evaluated the system and reached out for user feedback.

ACKNOWLEDGMENT

I acknowledge the use of GPT [40], an AI language model , which assisted me with grammar and rephrasing in the preparation of this thesis. Additionally, I like to express my sincere gratitude to my professor, April Wang, for her guidance and support throughout the work on this thesis.

1 Introduction

Previous studies have shown that peer assessment, as a method of collaborative learning through reviewing and testing each other's solutions, can boost students' motivation, engagement, and learning achievements [29, 30, 41, 45], while reducing the effort required for instructors to provide scalable personalized feedback[30].

Notably, the paper PuzzleMe introduced an in-class exercise tool that allows instructors to create and share exercises of the types: free-response, programming, and multiple choice [50]. The system achieved two peer assessment mechanisms. Firstly, *Live Peer Testing* enables students to create lightweight test cases and add them to a shared, moderated collection, allowing them to validate their code against these test cases, thereby enhancing their confidence in their code [50]. Secondly, *Live Peer Code Review* intelligently groups students based on their solutions to maximize meaningful code review and improve their code understanding [50].

While PuzzleMe demonstrated the potential of such systems, it faced limitations in accessibility, robustness, and scalability. Building upon this foundation, our thesis introduces a more accessible solution: a JupyterLab extension that utilizes a conflict-free shared data model based on Yjs [37].

The extension aligns with the exercise types of PuzzleMe and implements one peer assessment technique: *Live Peer Testing*. Leveraging the JupyterLab ecosystem, it enables the registration and interpretation of a new file type, ".puzzle", which facilitates the storage of exercise data and allows displaying a UI. For students, the extension offers a minimalistic interface that supports interaction with exercises, including problem-solving, solution submission, and participation in peer testing activities. The instructor's interface provides tools for creating and managing exercises, monitoring student progress, and delivering feedback.

Throughout the development of this extension, we focused on enhancing the reusability of UI components by decoupling them from the JupyterLab ecosystem. This approach facilitated the creation and publication of a React Quiz Component library, which offers versatile and configurable UI elements tailored for various types of exercises.

In addition, managing complex nested data structures in the ".puzzle" file, particularly the challenge of observing changes in deeply nested properties, revealed the need for a more refined approach. Consequently, we developed a specialized framework designed for efficiently applying updates and monitoring changes to normalized semi-structured data within the Yjs environment. This approach improved data consistency, simplified the overall architecture, and ultimately made the system more scalable.

To validate the extension, we deployed a JupyterHub instance within the ETH environment, which allowed multi-user access to JupyterLab. During a subsequent testing session, several technical issues were identified, and potential areas for improvement in both the User Interface and collaborative features were highlighted through questionnaires. Despite these challenges, the testing affirmed the extension's potential.

The contribution of this work includes:

- Development of core libraries and the JupyterLab extension, including their underlying concepts and software design.
- Integration of a JupyterHub server within the ETH environment which facilitates user management, enables customizable private and collaborative JupyterLab servers, and provides persistent storage.
- Public release of the libraries and extension to encourages further development and use by the broader community, contributing to the ongoing advancement of collaborative learning tools in educational technology.

Additional project details can be found on the product page [32]. The React Quiz Component library and the Yjs Normalized framework are accessible via npm [33, 34].

2 Related Work

The thesis builds on four related research and applied topics: Peer assessment as a form of collaborative learning, real-time code sharing in educational settings, scaling feedback, and Jupyter as an educational platform.

2.1 Peer Assessment as a Form of Collaborative Learning

Peer assessment is a form of collaborative learning where students critique and provide feedback to each other [49]. It has been widely recognized as an effective form of collaborative learning, offering numerous educational benefits, such as enhanced learning outcomes, better understanding of solutions, increased engagement, and the development of critical thinking skills [14, 44]. However, providing high-quality feedback can be challenging due to students' varying levels of expertise, necessitating structured guidance and intelligent grouping [6, 51]. CodeDefenders [41] is a notable example of a tool that combines peer assessment with gamification to enhance collaborative learning in programming education. Students write, assess, and improve each other's code by engaging in a game-like environment where they attempt to find and fix defects in their peers' code. This system relies on a pairing mechanism, where one student writes code variants and the other student tests them. In comparison, our approach follows the mechanism of PuzzleMe [50], where all students can write tests and add them to a shared collection, enabling the evaluation of each other's code.

2.2 Real-time Code Sharing in Educational Settings

Real-time code sharing is an essential feature in collaborative learning environments, allowing students and instructors to interact and provide feedback to each other instantaneously. Prior research has shown that using real-time collaborative code editors can boost students engagement, encourage collaboration and peer learning, and facilitate a more dynamic and interactive classroom experience [25]. Codeopticon enables instructors to monitor multiple students' coding progress, including their history, and allows for direct interaction with individuals in need of help through a chat feature [22]. Conducted interviews reported that the tool was perceived as very powerful for instructors, enabling them to structure their classes more efficiently. For answer walkthroughs, live-coding has been found to be preferred by students when instructors write out solutions in front of the class. This approach allows the instructor to provide explanations and engage students by asking questions [42]. For example, PuzzleMe enables live coding and propagates the instructor's code changes to students [50]. Our extension follows the same approach for answer walkthroughs and also leverages the concept of code monitoring of students' solutions, similar to Codeopticon.

2.3 Scaling Feedback

Providing timely and constructive feedback to students is a critical component of effective teaching, but scaling this process can be challenging, especially in large classes [43]. Previous work has explored different approaches to scale instructors' efforts: clustering student solutions by leveraging high-level patterns [19], propagating bug fixes from one student to the rest of the class [23], and leveraging prior student activity data to model the solution space [27]. However, applying these approaches in a real-time environment can be complex and require expert knowledge.

Our extension builds on the concept of learningsourcing, an approach that synthesizes the efforts of previous students to create materials for future learners [48]. PuzzleMe introduces a method for sourcing student explanations and solutions, as well as a collection of test cases shared among students [50]. These concepts are adapted and also integrated into our system.

2.4 Jupyter as an Educational Platform

Having a well-established platform that instructors are familiar with is crucial for effective teaching. In the context of data science, instructors must navigate diverse student backgrounds, teach disciplined workflows, and address challenges related to software setup, dataset relevance, and managing uncertainty in data analysis [28]. Jupyter notebooks facilitate this by allowing users to integrate code, explanations, and analysis into a single document [20].

Students benefit from Jupyter notebooks through interactive exploration, effective visualization, and enhanced programming skills development [4]. For instructors, notebooks offer flexible adoption across various educational formats, including in-class demonstrations, interactive labs, and full course content, accommodating different teaching styles and levels of experience [4]. Previous work has proposed best practices for teaching with Jupyter and highlighted both technical considerations and the positive impact of Jupyter notebooks in classrooms [2, 26].

Our extension builds on this work and prior experience to create a comprehensive and integrated solution that enhances interactive learning, streamlines the teaching process, and supports effective collaboration.

3 Extension Design

We designed this extension as a platform to enhance in-class programming exercises for both instructors and students. To illustrate the user experience from both perspectives, we describe how a hypothetical instructor, Bob, and a hypothetical student, Alice, interact with the system. In our scenario, Bob aims to create exercises for each type: text response, multiple choice, and coding, while Alice aims to solve these exercises. For each type, we provide detailed screenshots illustrating the actions taken by Bob and Alice.

3.1 Exercise Creation

To create a new exercise, Bob utilizes the JupyterLab launcher panel to generate a ".puzzle" file. Upon opening this file, a toolbar is displayed (Figure 1), offering options for creating different exercise types.

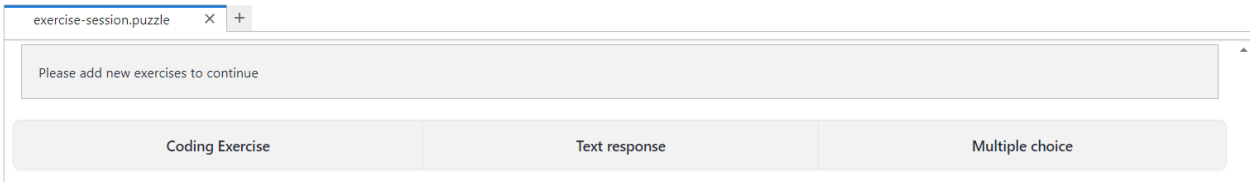

Figure 1: Exercise Creation Toolbar. Allowing for the creation of the three exercise type: text response, multiple choice, and coding

3.2 Text Response Exercise

Initially, Bob aims to assess Alice's understanding of Human-Computer Interaction (HCI) through a free-text response exercise. By clicking the corresponding button on the toolbar (Figure 1), he initiates the exercise creation process and proceeds to configure its details (Figure 2). Bob starts by defining the exercise question (2.1) and its corresponding solution (2.2) through a click-to-edit mechanism. Subsequently, he makes the exercise visible to Alice by selecting "Show Exercise" (2.3). Bob can then monitor her progress in a separate tab (2.7).

Figure 2: Bob perspective: Text response exercise. (1) Exercise description editor. (2) Solution editor. (3) Show exercise button. (4) Show solution button. (5) Exercise reorder Button. (6) Delete button. (7) Alice's solution tab.

Upon the exercise becoming visible, Alice is presented with a text response task (Figure 3). She begins by entering her solution (3.1) and then submits it for evaluation (3.2).

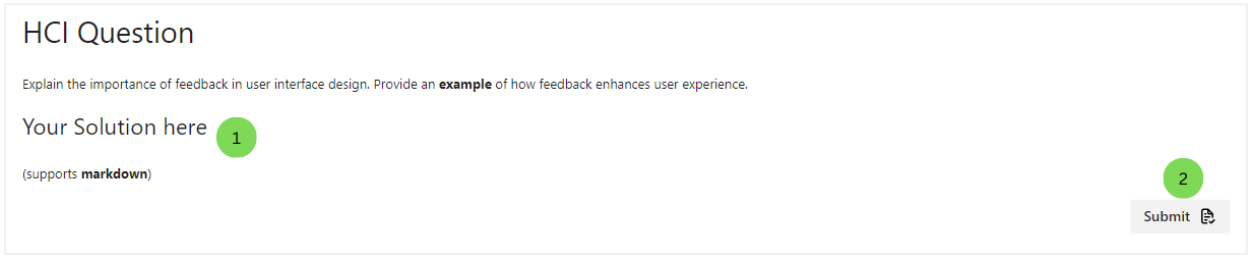

Figure 3: Alice perspective: Text response exercise. (1) Solution editor. (2) Submit button.

After observing Alice's response, Bob decides to provide feedback in the form of a comment (Figure 4). To allow Alice to view both the comment and the correct solution, he selects the "Show Solution" button (2.4).

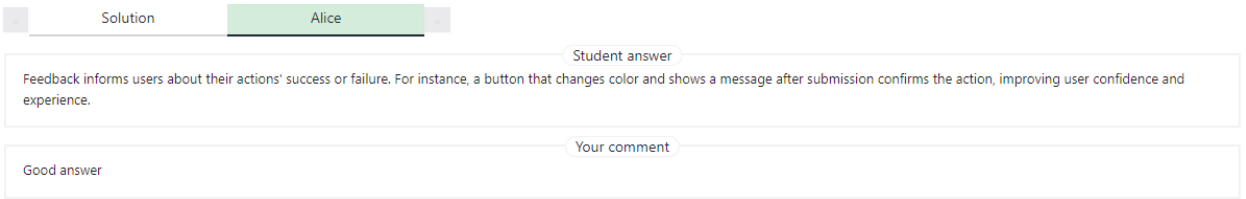

Figure 4: Bob perspective: Looking at Alice's solution

For future adjustments, Bob has the flexibility to reorder exercises (2.5) or remove unnecessary exercises (2.6).

3.3 Multiple Choice Exercise

To challenge Alice further, Bob opts for a multiple-choice exercise. By clicking the appropriate button on the toolbar (Figure 1), he accesses the exercise setup interface (Figure 5). Here, he can define the exercise prompt (5.1) and

customize settings like allowing multiple selections or randomizing answer order (5.2). Additional options include adding new answer choices (5.3), editing existing ones (5.4), and rearranging their sequence (5.5). Once satisfied, Bob can control exercise visibility (5.7) and analyse the distribution of student selections, including Alice's choice (5.8).

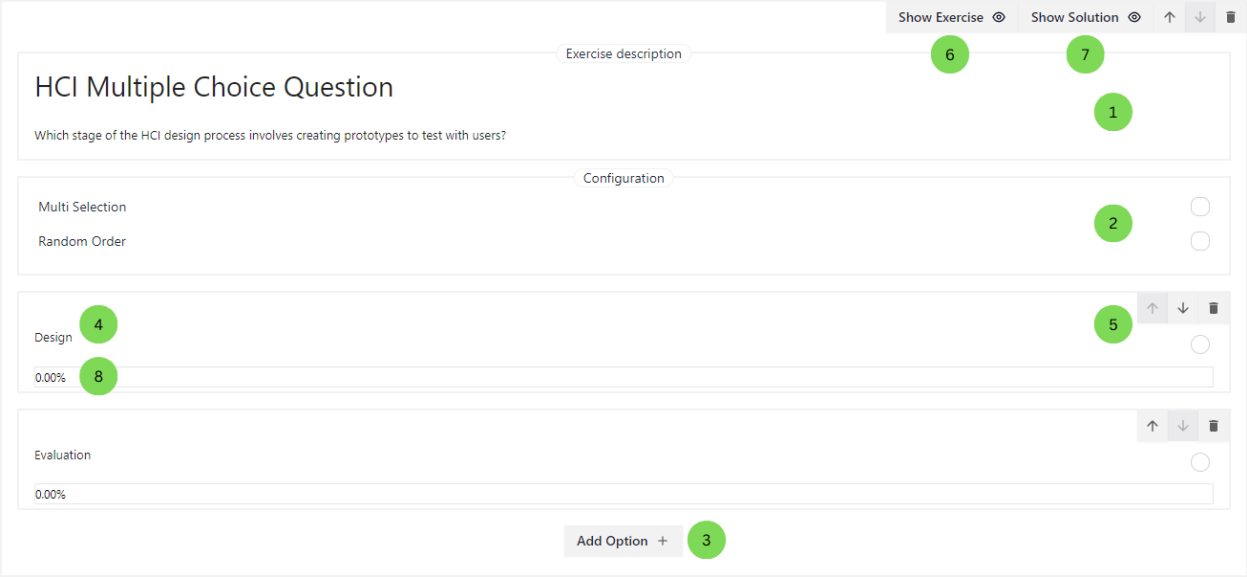

Figure 5: Bob perspective: Multiple choice exercise. (1) Exercise description editor. (2) Multiple choice configuration options. (3) Add option button. (4) Existing option. (5) Option reorder buttons. (6) Show exercise button. (7) Show solution button. (8) Distribution display.

Alice encounters a multiple-choice question (Figure 6) and selects her answer (6.1). As she chooses an option, the response distribution in Bob's interface (5.7) updates accordingly. To finalize her submission, Alice clicks the "Submit" button (6.2).

Figure 6: Alice perspective: Multiple choice exercise. (1) Multiple choice options. (2) Submit button.

Bob now wants the solution and distribution to be displayed to Alice, he achieves this by clicking on the "Show Solution" button (5.6).

3.4 Coding Exercise

Lastly, interested in assessing Alice's programming proficiency, Bob selects the coding exercise option from the toolbar (Figure 1). This action presents him with the coding exercise setup interface (Figure 7). Similar to previous

exercise types, Bob begins by inputting the exercise description (7.1). Subsequently, he configures the testing mode by choosing from three options (7.2):

1. Tests are active, and no tests are required.
2. Each student must verify one test before coding is allowed.
3. Tests are deactivated.

For this exercise, Bob opts for the first option, where no initial test verification is required. Following this, he enters the starting code (7.3) which will be provided to Alice and the expected solution code (7.4). A visual cue indicating successful code compilation (7.4) and confirms the code's executability. To enhance the exercise, Bob decides to create an additional test case (7.5). A prompt appears requesting a name for the new test case (7.6). After inputting the test case name, Bob proceeds to define the assertion code (7.7) and verifies its correctness against the master solution (7.8). Upon successful verification, the newly created test case becomes part of the shared test case collection. As a final step, Bob sets the exercise visibility to make it accessible to Alice (7.10). Once activated, he can monitor Alice's progress within the corresponding solution tab (7.12).

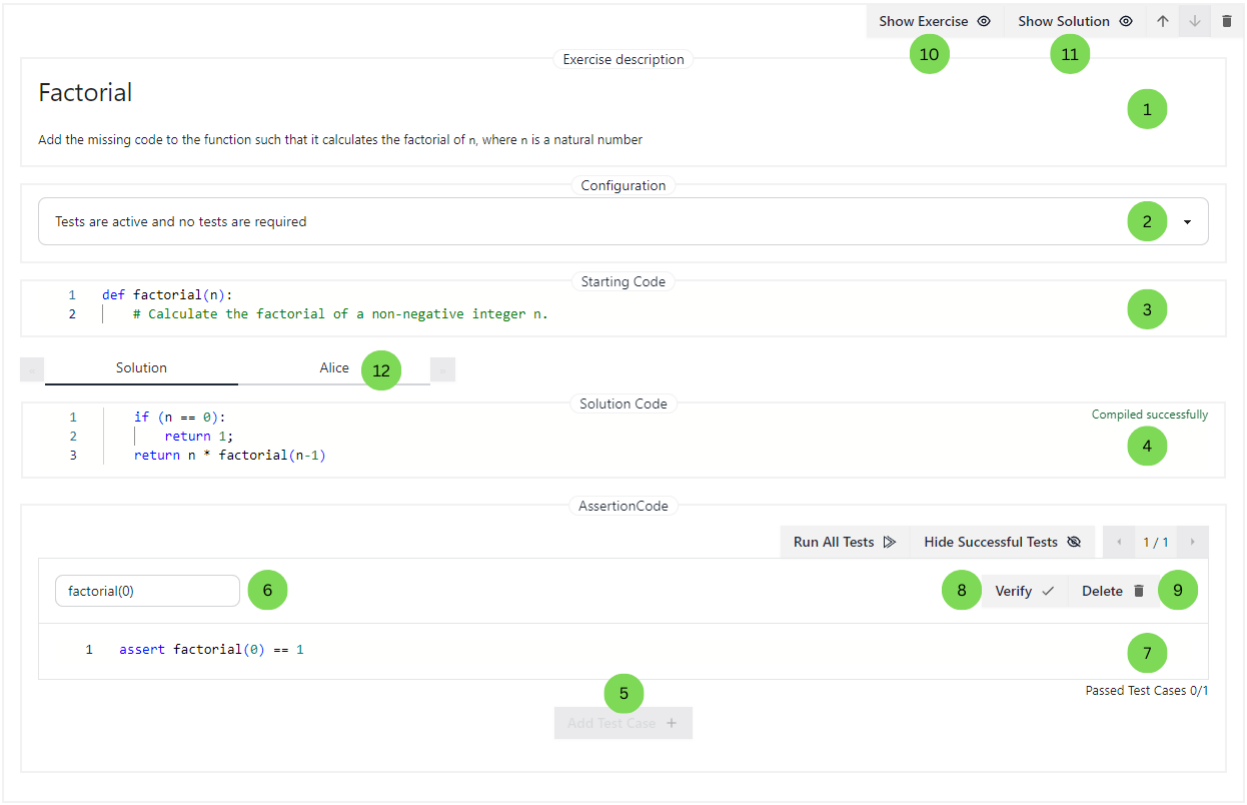

Figure 7: Bob perspective: Coding exercise. (1) Exercise description editor. (2) Testing mode selection. (3) Starting code editor. (4) Solution code editor. (5) Add test case button. (6) Test name input. (7) Assertion code editor. (8) Verify test case button. (9) Delete test case button. (10) Show exercise button. (11) Show solution button. (12) Alice's solution tab.

Alice is presented with the coding exercise interface (Figure 8) where she can initiate code implementation (8.1). Similar to Bob's interface, a visual cue confirms successful code compilation (Figure 8.1), while compilation errors

are indicated accordingly. Upon completing her code, Alice can execute all test cases collectively (8.2) or individually (8.3). To optimize the process, the interface allows her to filter test results, focusing solely on failed test cases (8.4). For further code validation, Alice has the option to create additional test cases (8.5). Once satisfied with her solution's correctness, Alice submits her code for evaluation. Notably, Alice retains the ability to contribute to the exercise's development even after submission by creating additional test cases to assist fellow students.

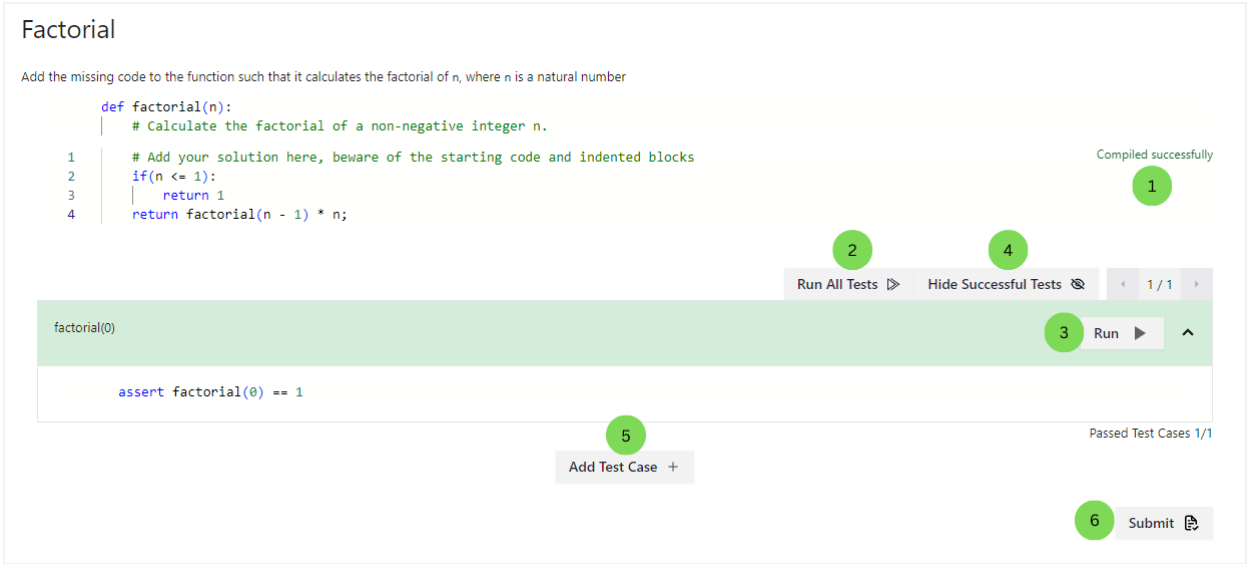

Figure 8: Alice perspective: Coding exercise. (1) Solution code editor. (2) Run all tests button. (3) Run single test case button. (4) Hide successful test cases button. (5) Add test case button. (6) Submit button.

After Alice submits her code, Bob can run the associated test cases to assess its correctness. Similar to the text response exercise, Bob can provide feedback in Markdown format (Figure 4). Additionally, he can reveal the master solution to Alice (7.11), allowing her to conduct a side-by-side comparison of her code and the master solution (Figure 9.

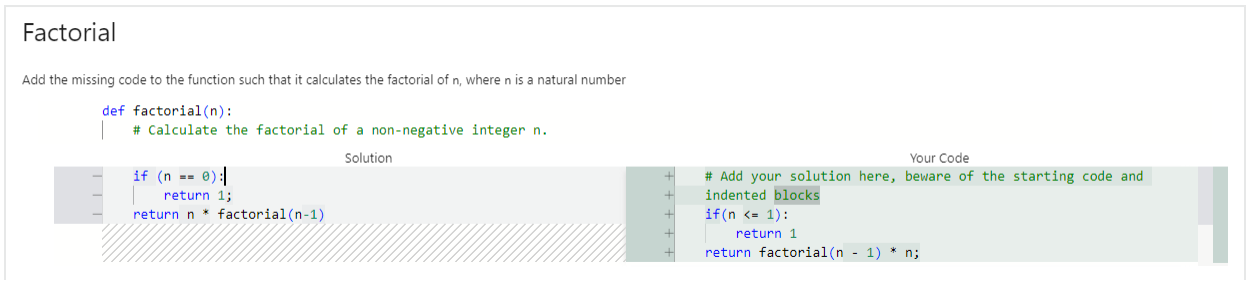

Figure 9: Alice perspective: Coding exercise - Diff view of her solution compared to the master solution

To maintain exercise quality, Bob has the authority to manage the shared test case collection by removing redundant or irrelevant test cases (Figure 7.9).

4 Implementation

Our implementation comprises four core components: a React Quiz UI component library, Yjs Normalized for collaborative data management, a JupyterLab extension for quiz creation and management, and JupyterHub for hosting multi-user JupyterLab environments. The following sections explore each component in detail, explaining their roles, functionalities, and how they integrate to create a comprehensive solution for collaborative quiz creation and management in JupyterLab.

5 React Quiz UI

To create a reusable UI for quizzes, we separated visual elements from business logic, resulting in a dedicated package for these elements. This section discusses the design decisions that were carefully elaborated to maximize reusability.

5.1 Architecture and Components

The library is intended for use in projects adhering to the open/closed principle [36], favoring composition over inheritance for creating large-scale components. Following this and the single responsibility principle [36], we developed standalone components that encapsulate specific functionalities. Each component is designed to be reusable and maintainable, allowing developers to extend and customize their applications without modifying the existing codebase. This approach promotes code reuse and simplifies testing and debugging, as each component can be independently tested and verified.

Exercise components align with the types presented by PuzzleMe [50]. Each component corresponds to a specific type of exercise or a segment thereof.

- Multiple choice exercise: Users can select either 1 or m from n elements (with $m \leq n$). The presentation of items can be randomized, and user input can be optionally disabled. Visual evaluation of the user's response is also available. Additionally, it supports both radio button (1 out of n) and checkbox (m out of n) selection modes.
- Coding exercise: The user is asked to solve a coding problem. Optionally, a starting code can be provided.
- Text response exercise: The user is asked to write a text response based on a given question or problem.

Util components serve as versatile toolsets that can be used across various exercise types. One such util component is the markdown editor, which provides a combined view of rendered and editable markdown content. The two sub-views can be arranged vertically, horizontally, or organized into tabs labeled 'Source' and 'Preview'. Alternatively, users can switch between rendered and source views in a Jupyter notebook style by clicking on the rendered markdown to edit and pressing Ctrl/Cmd + Enter to switch back.

Source components are small modular building blocks essential for both exercise and util components. Their primary responsibility lies in facilitating the modification or rendering of a specific source property.

- Markdown: Renders Markdown text for formatted content display.
- Code: Offers a Monaco code editor with language highlighting for code editing and visualization.
- DiffCode: Provides a diff code editor with language highlighting for side-by-side comparison of two code pieces.

5.2 State Management and User Interactions

State management within components only stores internal state (e.g., active tab of a markdown editor). External state (e.g., multiple choice options) must be handled by the parent component or a global state management solution, such as the Context API or Redux. This approach does not bind the developer to any specific state management solution, making the package more accessible and lightweight.

Any interaction with an exercise component that changes the initial answer emits an event to the parent component, which is responsible for updating the external state accordingly.

5.3 Styling and Storybook

Components are styled using Tailwind CSS [24], leveraging the UI library Daisy UI [17]. The themes feature allows designers to redefine color palettes, making the package highly customizable.

A comprehensive Storybook setup is provided to test the library and see the components in action. It allows developers and designers to browse the component library, view different states of each component, and interact with them in a live environment.

5.4 Implementation

Each exercise component asks for the same generic props ExerciseProps<T extends IExerciseObject, E extends IExerciseAnswer> (A.1), where T and E are implemented according to the needs of its component. Firstly, the IExerciseObject comprises exercise-specific data to be rendered, e.g., multiple-choice options. Additionally, it may include an optional metadata property, which holds supplementary information about the exercise, e.g., configuration details and a solution. Secondly, the IExerciseAnswer interface encompasses the user's solution specific to the exercise. It holds information about the answer given by the user and may also consist of a reference ID.

6 Normalized Semi-Structured Data for Yjs

To facilitate a collaborative environment, we needed an algorithm designed to ensure conflict-free editing. JupyterLab already offers a foundational framework for this through Yjs, which allows multiple users to edit the same file simultaneously without conflicts. Although the framework provides essential features, managing and observing state for large data structures can become highly complex and cumbersome. To address this, we developed an extension on top of the existing framework, specifically tailored to the Yjs implementation.

6.1 Yjs

Yjs is a high-performance CRDT for building collaborative applications that sync automatically. It has an alternative approach to implementation of operational transformation (OT). It ensures convergence, preserves user intentions and allows offline editing and can be utilized for arbitrary data types in the Web browser [37].

6.2 Data Normalization

Data normalization, a concept rooted in the design of relational databases, involves organizing data to minimize redundancy and improve integrity [3]. Dealing with nested or relational JSON data in Yjs applications can be complex and cumbersome. Observing updates in a deeply nested properties can be extremely inefficient, and lead to errors. Applying the same concept of normalizing data to a JSON structure can be beneficial. It ensures:

- Data consistency: Redundancies are reduced, lowering the risk of inconsistencies.
- Scalability: Managing large and complex data structures becomes more straightforward.
- Simplified state observations: The max depth level for nested data is 4 and therefore less complex to handle.

This approach follows the procedure of the section: Normalizing State Shape in the Redux documentation [16]. A normalized structure consist of collections. Each collection maintains a *byId* property and an *allIds* array property. The *byId* property is an object containing all entities of the collection, indexed by their unique identifiers. The *allIds* array holds the IDs of all entities in the collection. Consider a collection of users. The normalized structure would look like this:

```
{
  "users": {
    "byId": {
      "user1": { "name": "Alice", "age": 30 },
      "user2": { "name": "Bob", "age": 25 }
    },
    "allIds": ["user1", "user2"]
  }
}
```

6.3 Maintainer

Maintainer classes enable manipulation of normalized data structures in Yjs applications. That means they ensure that all changes are made within a Yjs transaction. Each class is responsible for one collection within the data structure. They provide key features to manipulate them:

- Add objects: Add new objects to the collection.
- Delete objects: Remove objects from the collection, with optional cascade functionality.
- Update objects: Modify existing objects
- Reorder objects: Change the position of objects in the collection. The order within *allIds* property implicates the real order
- Manage Property Arrays: Add, remove, and reorder elements in array properties of an object.

6.4 Observer

In general, observers facilitate the observation of semi-structured, normalized data within a Yjs Y.Map, track changes to the data, and dispatch events accordingly.

RootObserver The RootObserver class is an abstract class that primarily functions as a root data observer and secondly acts as a registry for Sub-Observers that track more granular changes within the data. The class defines several abstract dispatcher methods that subclasses must implement to handle various types of data updates:

- Root dispatcher: Dispatches the entire normalized state.

- Add dispatcher: Dispatches an event to add an object.
- Delete dispatcher: Dispatches an event to delete an object by its id.
- Update property dispatcher: Dispatches an event to update a specific property of an object.
- All ids dispatcher: Dispatches an event to update the list of all object ids.

6.5 Sub-Observers

Sub-Observers are managed by the library itself. During runtime, they are created and deleted, matching the underlining data structure.

- AllIdsObserver: Observes changes in the order of the *allIds* Y.Array
- ByIdObserver: Observes additions and deletions of objects with in the byId Y.Map
- ObjectObserver: Observes changes in each Y.Map within the *byId* Y.Map

6.6 Architecture

The figure 10 illustrates a collaborative data synchronization system that leverages the introduced library. Upon initialization, the root observer registers itself to monitor changes in the root data structure. If the properties *byId* and *allIds* are detected in the root, two sub-observers, ByIdObserver and AllIdsObserver, are created. These sub-observers immediately begin monitoring the *byId* and *allIds* properties, respectively. When ByIdObserver detects a new property within the *byId* object, it creates an ObjectObserver, which starts observing changes in the corresponding object. If the object is deleted, the associated ObjectObserver stops observing and is subsequently removed.

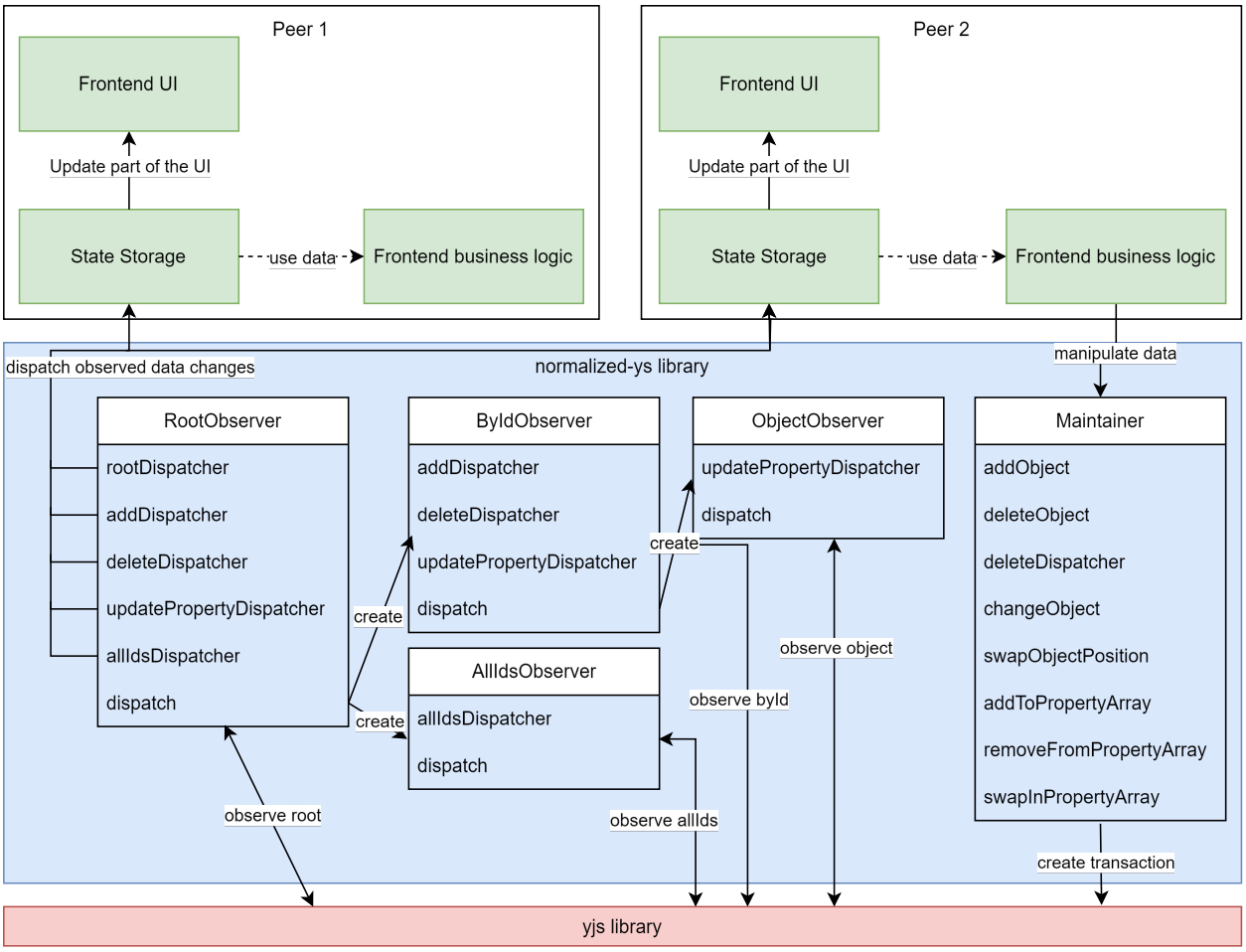

Figure 10: Normalized Yjs architecture. The diagram illustrates how the library establishes a hierarchical observer pattern to efficiently track changes within a normalized data structure.

Peers can now manipulate the data through the Maintainer. Any changes to the data structure will eventually be observed by all peers, and the appropriate payload will be dispatched accordingly.

7 JupyterLab Extension

JupyterLab supports the enhancement of its environment through extensions, which can improve existing functionalities or introduce new ones [10]. Our extension contributes a new document type to the JupyterLab ecosystem. Specifically, it interprets any ".puzzle" file as a JSON file and offers functions and a UI interface to manipulate this data.

7.1 Data Model

The data model draws significant inspiration from JupyterLab's notebook format. In Jupyter Notebooks, cells dictate the rendering and user interaction for each segment [7]. To incorporate additional properties while excluding some existing ones, we opted to create a new data structure.

7.1.1 Structure

Similar to the Jupyter Notebook, we introduced various cell types for different exercise formats, including multiple choice, text response, and coding. Each cell contains the following properties:

- **id:** Universally unique identifier (UUID)
- **type:** Defines the type of the cell (e.g., code-cell, multiple-choice-cell).
- **metadata:** Holds configuration values.
- **description:** A field that outlines the task or question for the specific exercise.
- **studentSolutions:** A list of fields, each representing a student's solution.
- **documentId:** An technical reference to the unique identifier of the document.

Next, we introduced *fields* as a new structure, defined as follows:

- **id:** Universally unique identifier (UUID)
- **type:** Defines the type of field (e.g., markdown, code).
- **createdBy:** Holds a reference to the creator by their user ID.

Each cell can contain additional nested properties that hold a field or a list of fields. However, this introduction highlighted a problem with following the Jupyter Notebook model:

A cell *A* might hold a property defining a field *C*, and a second cell *B* might hold the same or another property defining the same field *C*. Instead of referencing field *C*, a new field *D* is created, resulting in two identical fields *C* and *D*.

Moreover, when using yjs, observing and manipulating deeply nested JSON objects posed a challenge. Tracking changes in deeply nested structures is complex because each level of nesting might require its own observation mechanisms. Furthermore, changes at deeper levels must propagate correctly to maintain consistency.

When it comes to data manipulation, updating or accessing deeply nested fields often involves verbose and error-prone code. Moreover, ensuring data integrity during updates, especially in collaborative environments, can be challenging.

Lastly, frequent updates in deeply nested structures can cause significant performance overhead due to the need to traverse and modify multiple levels. This led to the normalization of the data structure and the development of the *yj-normalized* package, described in section 6.

7.1.2 Normalizing the Structure

Normalizing the data structure leads to fields being declared and defined in a separate collection called "fields". In this structure, each field property within a cell contains only a reference ID to the corresponding field object in the "fields" collection. The empty structure looks as follows:

```
{ "cells": {"byId":{}, "allIds:[]}, "fields": {"byId":{},"allIds":[]}}
```

This approach enhanced data management significantly and resolved the previously mentioned issues.

7.1.3 Factories

Instead of creating a new cells or fields directly via constructor, we opted for the Factory Method. The intent behind the Factory Method design pattern is the following: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses [35].

A Factory Method is structured where one type of class, a Creator, is creating instances of a different type of class, a Product [35].

Product and ConcreteProduct In our data structure we can identify: *Cell* and *Field* as a Product, which are defined by the two interfaces *ICell* and *IField* respectively. Each Product, may encompass multiple concrete implementations. Every type of exercise: text response, multiple choice, and coding exercise introduces its corresponding concrete *ICell* implementation. Regarding *IField*, implementations are categorized into *SourceField* and *SolutionField*. While *SourceField* encapsulates a *src* property and specifies its interpretation (e.g. *ICodeField*, *IMarkdownField*), *SolutionField* represents a student's solution, holding additional properties such as instructor comments or grades. For each exercise type and therefore each concrete implementation of *ICell* there exists a corresponding concrete *SolutionField* interface (e.g., *ICodeSolution*, *IMultipleChoiceSolution*).

Creator and ConcreteCreator A creator defines a factory method that returns an object of type Product. It may also provide a default implementation of this method that returns a default ConcreteProduct object [35]. In our implementation, we have defined two creators: **CellFactory** and **FieldFactory**. Each creator specifies a default implementation for a concrete *ICell* and *IField*, respectively.

Each concrete *IField* and *ICell* corresponds to a ConcreteCreator identified by its *type* property, facilitating the creation of specific cell or field types within the application logic.

FactoryService To centralize object creation, we have implemented two singleton services [36]: *CellFactoryService* and *FieldFactoryService*. These services invoke the appropriate ConcreteCreator to instantiate objects.

The architecture is depicted in Figure 11.

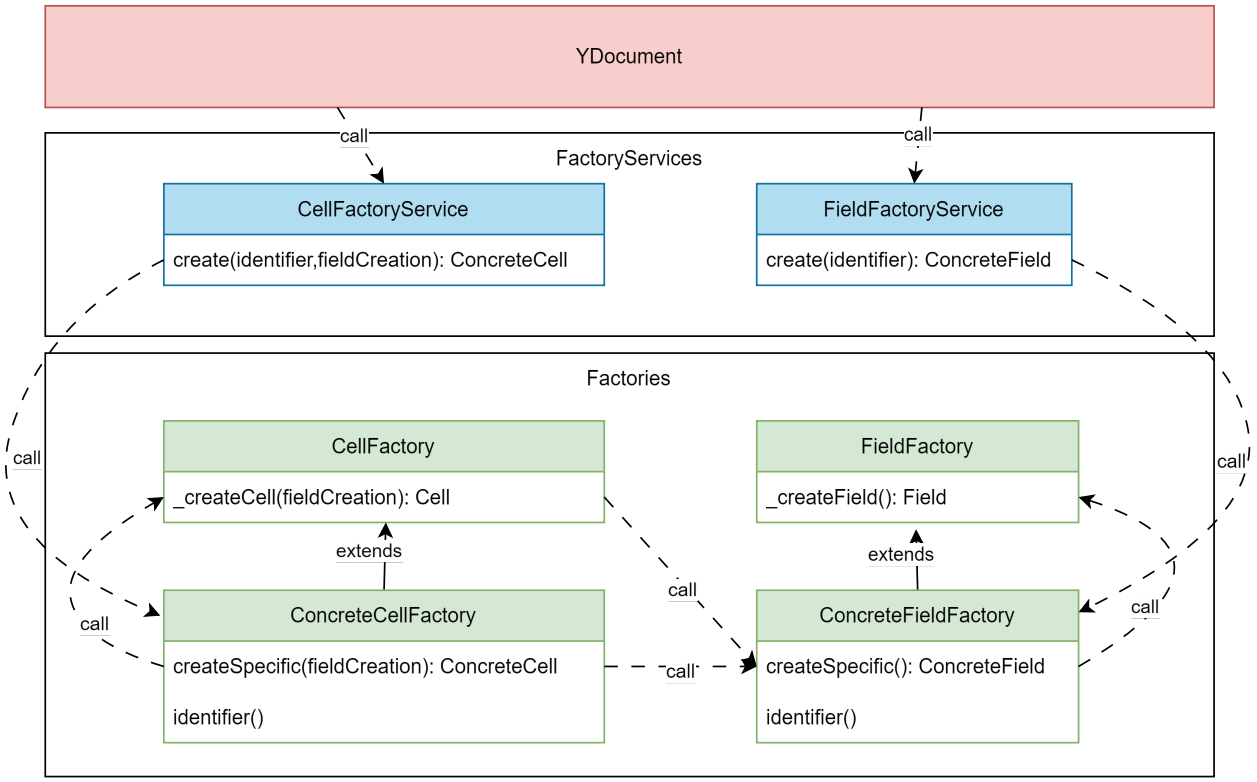

Figure 11: Factory architecture. The diagram depicts the interplay between YDocument, FactoryServices, Creators and ConcreteCreators.

7.1.4 Handling Relational Dependencies

Relational data comes with dependencies. Having a Cell *A* that referencing field *B* as the only cell. Field *B* might only exist because of Cell *A*. Therefore, deleting Cell *A* should also delete *B*.

Utilizing the Maintainer class (refer to section 6.3) of the yjs-normalized package, allows catching any reference in an on delete cascade callback. This allows to delete any field referred by a cell when deleting the corresponding cell.

Furthermore, a similar relationship may arise when creating a cell. A Cell *A* is deemed valid only if the referenced Field *B* also exists. Therefore, creating Cell *A* necessitates the creation of Field *B* simultaneously. Extending the *CellFactory* to include a parameter that accepts a field creation function enables us to address this challenge effectively. The field creation function ask for a type and returns the corresponding Field.

7.1.5 YDocument

The YDocument functions as an abstraction layer for the ".puzzle" file, providing a structured interface for data manipulation and the subsequent invocation of Maintainers. It plays a crucial role in initializing both Maintainers and Observers, ensuring that each collection —namely *cells* and *fields* — has an associated Maintainer and Observer.

When tasked with creating a new cell of a specific type, the YDocument propagates this request to the appropriate FactoryServices. This process involves not only the instantiation of the new cell but also the creation of associated fields. Both the new cell and the referenced fields are encapsulated within a single transaction to maintain data integrity and consistency. This transactional approach ensures that all related modifications are applied atomically,

thus preserving the coherence of the overall data structure.

7.2 State Management

The official documentation for a Jupyter Lab extension describes state management through signalling, where Signals represent dynamic data changes within JupyterLab [11]. After evaluating and prototyping with Signals, we have determined that our approach to state management will not primarily depend on signalling. This decision is closely tied to our preference for using a normalized data structure.

The signalling library includes a React component, UseSignal, which facilitates listening to individual signals and processing their payloads. However, managing numerous properties that evolve over time necessitates nesting multiple instances of the UseSignal component. Moreover, when handling arrays of fields that undergo changes, this leads to further nesting of UseSignal components.

As each React component relies on multiple parts of the data structure, complexity and maintenance overhead for developers increase significantly.

Therefore, we have opted to utilize Redux as our global state management library. While this approach introduces some redundancy in data storage, we contend that the impact on performance is negligible due to our operation with small file sizes.

A redux application is defined by three parts [15]:

- The state, the source of truth that drives the app
- The view, a declarative description of the UI based on the current state
- The actions, the events that occur in the app based on user input, and trigger updates in the state

7.2.1 Shared Yjs Document State

In our scenario, part of the state directly corresponds to the content of the *.puzzle* document on a one-to-one basis. By leveraging the yjs-normalized library, user interactions with UI components that modify data within the document invoke a Maintainer (refer to section 6.3). The Maintainer subsequently updates the document's content, which triggers observations of these changes. These observations then prompt an action, which is processed by a reducer. The reducer computes the latest state based on the provided payload. Finally, only the part of the UI affected by the changes is updated.

Following the introduction of two collections, namely *cells* and *fields*, we have implemented corresponding Redux slices to manage their data structures.

Aligned with the event dispatching mechanism of the yjs-normalized library (see section 6.4), the following reducers have been defined for the *cells* slice: addCell, deleteCell, setCells, updateCellProperty, and updateCellsAllIds. Similarly, equivalent reducers are implemented for the *fields* slice.

Every action payload includes a document ID to manage cases where multiple documents are simultaneously open and manipulated. This ensures that updates intended for one document do not inadvertently affect another, maintaining clear separation between document interactions.

This state is shared among all clients interacting with the Yjs document and exhibits eventual consistency.

7.2.2 Client State

Some state is specific to individual clients and not shared. User-related information, such as usernames and assigned groups, is stored in its own slice named *user*.

Regarding interactions with the Kernel (see section 7.4), we adopt a normalized approach with two slices. Firstly, the *kernelTestResultSlice* which holds information concerning the success or failure of test runs. Secondly, the *kernelExecutionResultSlice* that stores the output generated from code execution. Each of these slices is equipped with reducers designed to set or remove new objects as needed.

7.2.3 Overview

The Redux store can be visualized as follows:

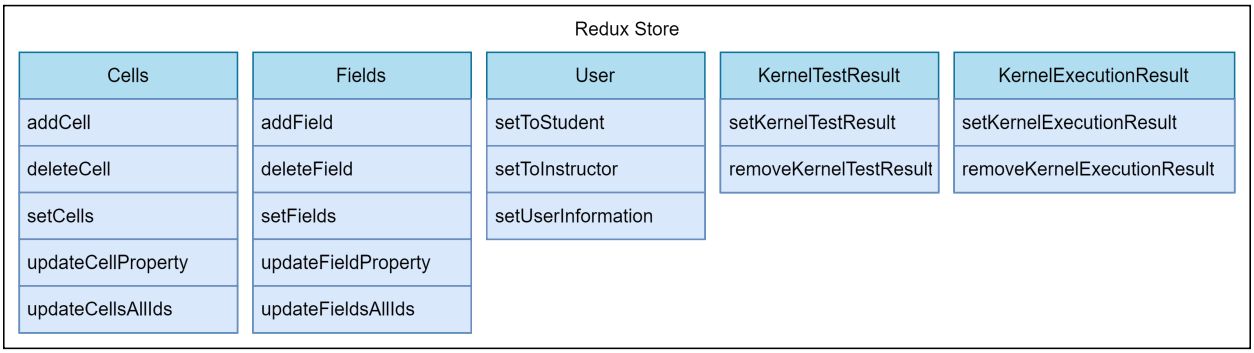

Figure 12: Redux Store. The diagram visualizes the slices and their reducers.

7.3 Design

Our objective was to create a simple and intuitive interface that users can easily interact with, drawing on some of the UI principles from the Jupyter Notebook interface. This approach ensures that users do not have to learn a new system. The following components were adapted from the notebook interface:

- Icons: All icons are directly rendered from the Jupyter UI components library [12].
- Markdown Editor: Allows users to switch between rendered and editing views by clicking on the rendered markdown or pressing CMD/Ctrl + Enter while editing.
- Cells: Cells are highlighted with a border when hovered over or when interacting with a component within the cell.
- Cell Toolbar: A toolbar offering several actions is displayed for each cell when the cell is hovered.

The design builds upon the *react-quiz-ui* library (refer to section 5). Jupyter Extensions typically implement and define new widgets. For reusability, consistency, and testing purposes, we first implemented the UI components without any business logic. Consequently, we also developed an abstraction layer on top of the *react-quiz-ui* library to consistently apply the same configuration to the same component within that library. Then we developed the widget with business logic using these UI components.

7.3.1 UI

Similarly to the *react-ui-quiz* component library, we first implemented small modular components and subsequently built more complex components with them. Additionally, we utilized storybook to do visual testing with different initial component configurations. This approach allowed for the development of consistent visual elements.

Firstly, additional icons were defined using the interface of the Jupyter UI components library. Secondly, common components and exercise-specific components were designed (see B.1). Each *react-ui-quiz* component is encapsulated by a wrapper component.

The implementation of wrapper components offers several advantages. Consistency is a primary benefit, as wrapper components ensure a uniform appearance and behavior across different parts of the widget. Simplified configuration is another significant advantage; centralizing the configuration in wrapper components reduces redundancy and potential errors, thereby facilitating easier maintenance of the codebase.

7.3.2 Widget

The widget connects the state (refer to section 7.2) with the UI components (refer to section 7.3.1) and implements further application logic.

Context Passing props can become verbose and inconvenient when deeply nesting components or when multiple components require the same data [46]. To mitigate the issue of prop drilling, React Context was utilized. Two specific contexts were introduced to facilitate communication within different parts of the application: the *DocModelContext*, which encapsulates methods for manipulating the Yjs document (e.g., *changeCell*, *deleteCell*), and the *KernelContext*, which provides methods for interacting with the kernel (e.g., *executeTest*, *verifyTest*; see section 7.4).

Integrating React Context with a Redux store results in streamlined props, primarily requiring the transmission of only the relevant cell ID or field ID.

Components The components employ Redux hooks to load the relevant state data and utilize the context methods described above to connect to the UI components. Additionally, they implement further application logic, including determining the visibility of components or parts thereof based on the current Redux state and the provided user information (e.g., instructor components are displayed only to users with the instructor role). They also map a list of cells or fields to the corresponding components based on the type presented in the state data (see section 7.1.1). Finally, they calculate exercise-specific values based on the current state.

7.3.3 Themes

Leveraging the theming features of Daisy UI [17] and the Monaco Editor [1] allows for changing the default colors. For Daisy UI components, we decided to adapt the color scheme to match that of Bootstrap [47]. The colors are more accessible for color-blind people. The DiffCode component implemented in the *react-quiz-ui* library (see section 5) is designed to show the differences between the master solution and the student solution. Since green and red might be misleading and could indicate whether something is correct or incorrect, we opted to use neutral colors.

7.4 Kernel communication

To enable the execution of code, we utilized JupyterLab kernels. We introduced the *KernelMessagerService* which allows communication with the these kernels.

7.4.1 Methods

The service provides three functions that return results through dispatching of corresponding Redux actions. The *executeCode* function retrieves code from multiple fields, executes it, and dispatches the *setKernelExecutionResult* action with the resulting outputs. The *executeTest* function also retrieves code from multiple fields and checks for

the presence of assertion code (e.g., *assert 2 == 1 + 1*). It evaluates the test's meaningfulness, executes the code, and dispatches *setKernelExecutionResult* with the resulting outputs. Additionally, it dispatches *setKernelTestResult* to indicate whether the test was successful. The verifyTes function follows the same steps as executeTest, but with the added capability of emitting a signal if the test is successful. This signal is then captured and processed by the YDocument, which updates the data entry of the corresponding test case to verified.

7.4.2 Meaningful Tests

Without meaningful tests, students could easily create test cases that do not provide any further insight into how a function or application operates (e.g., *assert 1 == 1*). Determining whether a test is meaningful is a challenging problem. However, assuming the assertion code follows a certain structure, we can simplify the problem by leveraging regex matching.

An assertion code can contain multiple assertions. Therefore, we test each assertion for meaningfulness, assuming the following structure:

```
assert {value}
OR
assert {left side} {operator} {right side}
```

For the first type of assertion, we consider only the simplest case where the value is "True", excluding functions that return boolean values. For the second type, we account for operators such as ==, !=, >=, <=, >, and <. For simplicity, if none of these operators are used, we assume that the test is meaningful. Utilizing regex matching, we identify the different parts of the assertion. Using the JavaScript method *eval*[18], we can evaluate both the left side and the right side of the assertion. We compare the evaluated and raw values of both sides. If either the raw values or the evaluated values are equal, we can conclude that the assertion is meaningless.

This method can be easily extended and refined.

7.4.3 Execution output

Kernel executions can produce various types of outputs. To identify the format of these outputs, we have implemented a dedicated helper class for this task. The ExecutionOutputHelper takes the execution results as input and formats them into a standardized output that can be displayed using the KernelOutput component (refer to section 7.3.1).

The following output formats are supported:

- **application/json**: JSON objects are displayed using a third-party library (*react-json-view-lite*).
- **image/png**: Images are converted to base64 format and displayed using an HTML *<img>* tag.
- **text/html**: HTML code is sanitized and then inserted into a * * tag. HTML sanitization is performed to mitigate security risks [13].
- **text/plain**: Text messages are displayed in a neutral alert `<div>`.
- **error**: Error messages are displayed in an error alert * *.

Any output format other than those listed will result in an error indicating that the format is not supported.

7.5 Plugin

The plugin serves as the entry point for the JupyterLab extension manager and provides several injectable services crucial for managing specific components of the JupyterLab ecosystem. It facilitates access to user information through the JupyterFrontEnd service, which suffices for single-user setups. However, in a multi-user environment with JupyterHub, additional details such as user group assignments are needed. To acquire this information, we issue a request to the JupyterHub API with the username, allowing us to distinguish between instructors and students. Furthermore, the Plugin introduces a command for creating empty ".puzzle" files, which is essential since a completely empty ".puzzle" file could lead to errors. These files are initialized with the structure defined in section 7.1.2 and the command is incorporated into the JupyterLab launcher, which appears when no document is open. The plugin also manages document type registration, enabling the addition of a new file type, ".puzzle" , and configures kernel behaviour to ensure that a new kernel is created when opening a ".puzzle" document and shut down when the document is closed. Additionally, the Restorer, working in combination with a tracker, allows the application to restore users' previous sessions by reopening documents that were active during the last session.

7.6 Overview

The figure 13 illustrates the key components and their relation to each other.

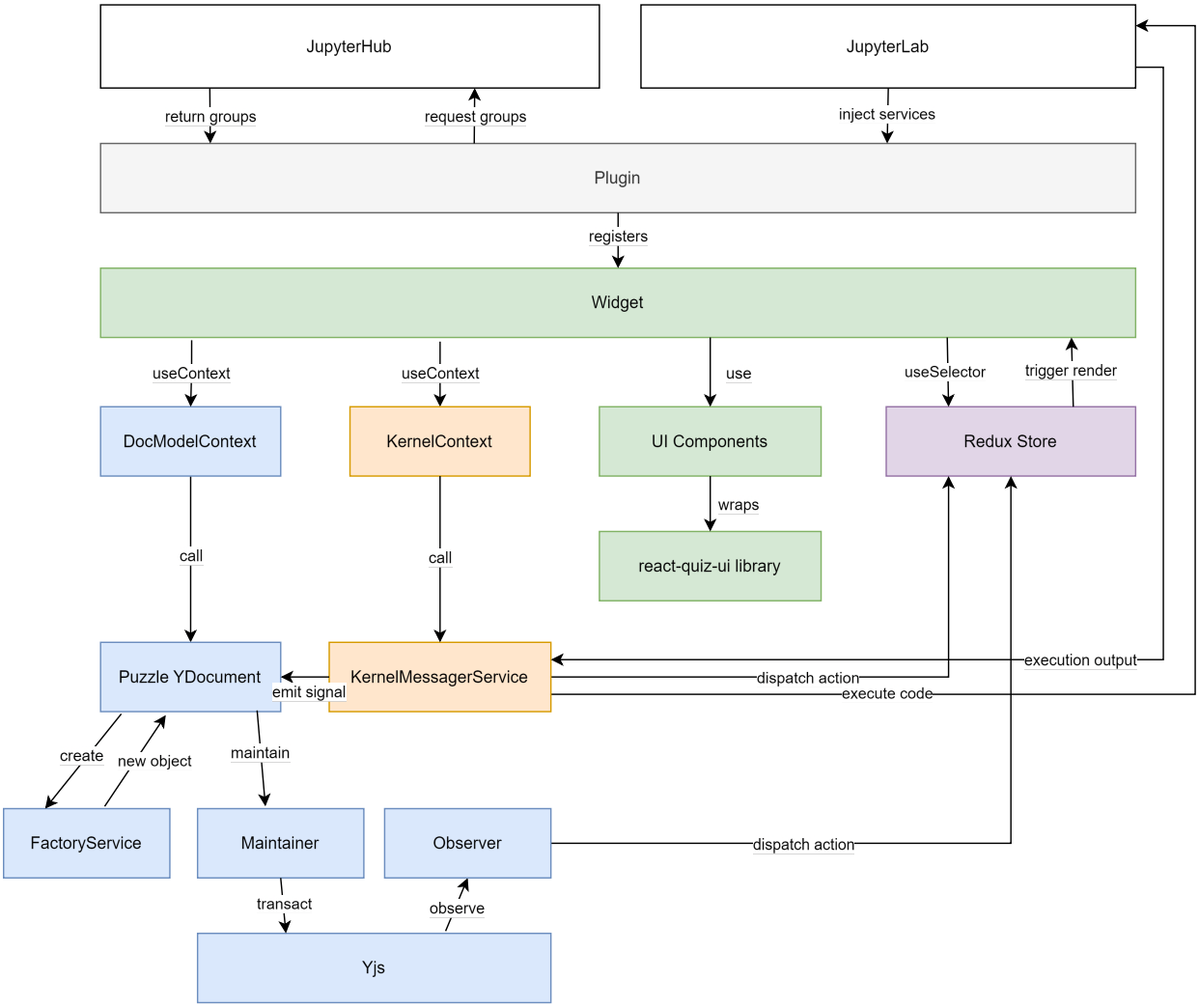

Figure 13: JupyterLab extension architecture overview. This diagram illustrates the core components of the JupyterLab extension, categorized by their functions: visual elements (green), communication with the JupyterLab kernel (orange), exercise data management (blue), storing the current state (violet), and integration into the JupyterLab ecosystem (white and grey).

JupyterHub and JupyterLab serve as platforms with distinct APIs for interaction: JupyterLab offers services that facilitate seamless integration of plugins into its ecosystem, while JupyterHub provides a REST API for retrieving user group assignments.

In the plugin architecture, the widget is registered with the injected services, leveraging the Redux store as the single source of truth for maintaining the current state. This setup ensures that any state changes automatically trigger a re-rendering of the widget components. The widget components are built using modular React components, which include wrappers for react-quiz-ui to maintain a consistent user interface while implementing the application logic.

To manage component interactions efficiently and avoid prop drilling, React contexts are utilized. Specifically,

DocModelContext facilitates interaction with the YDocument, and KernelContext grants access to the methods provided by KernelMessagerService.

The creation and management of new cells are handled by the YDocument through the FactoryService. The Maintainer oversees any new cells, fields, or modifications to existing entities, creating transactions that capture these changes to the state. Observers then register these changes and dispatch the relevant actions to the Redux store.

Additionally, when execution requests are sent to the kernel, actions are dispatched with the execution output, which is then updated in the Redux store.

8 JupyterHub

To make the extension more accessible and also testable, we integrated a JupyterHub server into the ETH infrastructure. JupyterHub is an open-source platform designed to provide multi-user access to JupyterLab. It offers to us a scalable, shared computing environment where multiple users can interact with Jupyter notebooks simultaneously. For each user, it provides an isolated JupyterLab server on a shared infrastructure. It supports customizability for authentication, environment spawners, permissions and more. Furthermore, it provides a user-friendly interface for both administrators and users, facilitating the management of user accounts, resources, and server configurations [9].

8.1 Authentication

JupyterHub offers many authenticators to build upon. For simplicity, we decided to use the GitHub Authenticator. In Figure 14 the procedure for OAuth is visualized. It follows the following steps [9]:

1. A client makes an HTTP request to the JupyterHub server, which acts as the OAuth client.
2. If no credentials are provided, the server redirects the client to the OAuth provider, which in this case is GitHub. The provider receives additional information such as the OAuth client ID and the redirect URL to navigate back to the JupyterHub server.
3. The user is prompted to authenticate using their GitHub login credentials, which may already be stored in a cookie.
4. After the user consents to authentication via GitHub, the provider issues an OAuth code, a short-lived record of the authorization.
5. Finally, the user is redirected back to the JupyterHub server using the provided redirect URL.

The necessary configuration values are stored in an "env" file on the server. This file defines 3 environment variables:

- `OAUTH_CLIENT_ID`: An unique identifier assigned to the client application by the GitHub authorization server when the application registers.
- `OAUTH_CLIENT_SECRET`: The `OAUTH_CLIENT_SECRET`, along with the `OAUTH_CLIENT_ID`, is used to authenticate the client application to the GitHub authorization server. This secret is never exposed to the public.
- `OAUTH_CALLBACK_URL`: It is the URL to which the GitHub authorization server redirects the user-agent after the user has authorized the client application.

8.2 Spawner

A Spawner initiates a single-user notebook server. It serves as an abstract interface to a process, and is able to perform 3 actions [9]:

- Starting a process
- Polling to check if a process is still running
- Stopping a process

The Jupyter Project offers various spawner options. We chose the DockerSpawner because our server environment includes a Docker daemon. We had to consider the following configurations:

- The DockerSpawner supports the use of multiple prebuilt Docker images, each potentially incorporating additional JupyterLab extensions. For our implementation, we leverage a Docker image specifically designed for collaborative learning, which is produced through a GitHub pipeline. When multiple images are available, JupyterHub prompts the user to select the desired image for launching their JupyterLab instance.
- Network: To allow JupyterHub to communicate with the JupyterLab instances, they need to run in the same docker network. We used a predefined network to also expose the JupyterHub service (port 8000) to the public.
- Working Directory: jupyter/docker-stacks notebook images run the Notebook server as user jovyan and set the user's notebook directory to */home/jovyan/work* [8]. We followed this convention.
- Volumes: For us, two factors were relevant. Firstly, persistency, which is ensured by using user-assigned named Docker volumes. Secondly, a populated working directory, which we achieve by mounting an additional volume containing test files.

  The user-assigned volume is mounted to */home/jovyan/work*, and the additional volume, including test files, is mounted to */home/jovyan/work/testfiles*. Within the *testfiles* directory, users are only allowed to modify existing files but are not permitted to create or delete directories or files. On the other hand, on the user-assigned volume, users have full privileges.

8.3 Allowed Spawner Images

Allowed images must be declared in a dictionary. The key refers to the value displayed to the user, and the value represents the actual image [8]. To simplify maintaining this dictionary, we use a dedicated *allowed_images.yaml* file. On startup, the contents of this file are passed to the JupyterHub configuration. Any changes to this file take effect only after restarting the server.

8.4 User and Groups

User and group management for JupyterHub can be accomplished through two methods: On startup, by providing a file named *admins*, JupyterHub registers each user in this file as an administrator. Additionally, by mounting a *room-config.yaml*, collaborative servers are created, and users are assigned to the instructor group. Such a *room-config.yaml* is structured as follows:

```
instructors:
    - alice
rooms:
    collaborative-test:
        members:
            - bob
```

The instructors' property maintains a list of all instructors, who can access each collaborative server and act as instructors. The members' property contains users who have access to that particular collaborative server but are not assigned the instructor role. Changes to the files only take effect after the server restarts. The second method is through the JupyterHub Admin UI, where the above permissions can also be granted or revoked by an administrator during runtime. Additionally, the UI allows for creating new collaborative servers and assigning users to them.

8.5 Copier

Populating users' working directories posed a challenge. In JupyterLab notebook images, creating a new directory or copying a folder to a non-existing path within a Docker instance results in ownership by the root user, which prevents the user *jovyan* from operating on this path.

To address this issue and achieve populated working directories, we implemented a workaround: A copier Docker service mounts the testfiles directory from local storage and a named Docker volume (see figure 14). It then overrides all existing files within the named Docker volume with the content from the mounted testfiles directory. To allow the user *jovyan* to operate on this path, permissions are recursively changed. After completing its task, the service is removed. This process can be manually repeated at later stages to modify existing files or add new files to the shared Docker volume.

This Docker volume is subsequently mounted in each JupyterLab instance, as described earlier (see section 8.2).

8.6 Architecture

Figure 14 visualizes the most important aspects of the system discussed in the previous sections.

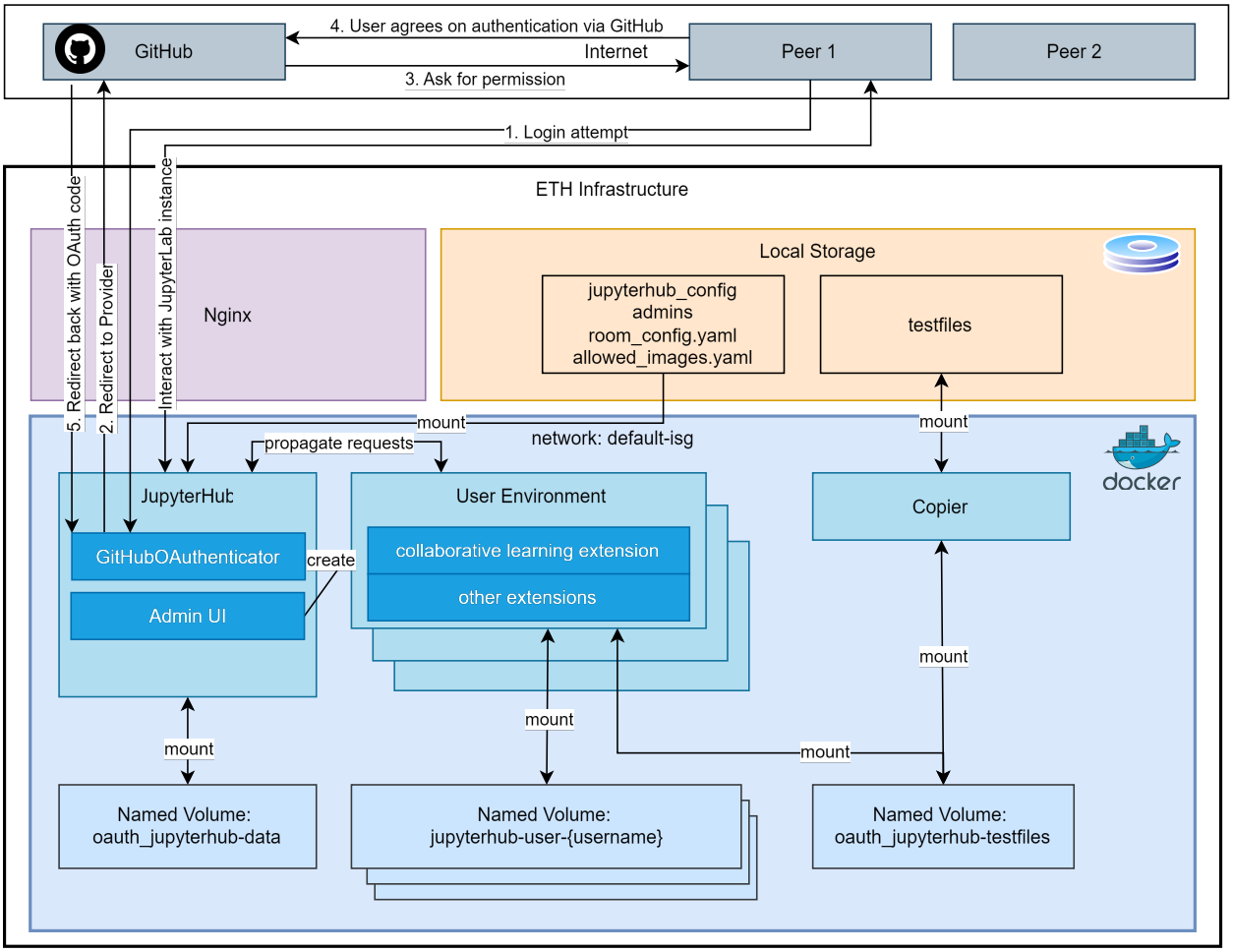

Figure 14: JupyterHub Architecture. The diagram illustrates the architecture of the JupyterHub environment, showcasing the interaction between key components: JupyterHub, Jupyter Lab spawner containers, Copier service, and local storage. The diagram highlights the authentication flow, user management, and data provisioning processes.

9 User Testing

To evaluate and test the system, we conducted two user testing sessions, each targeting different user roles: students and instructors.

The objectives of these evaluations were to assess several key aspects. First, we aimed to evaluate the usability of the interface, examining how intuitive and user-friendly it is for both students and instructors. Second, we focused on performance to determine how effectively the extension operates under real-world conditions, with particular attention to responsiveness and stability. Third, we sought to gather user feedback, collecting insights into users' experiences, suggestions, and pain points to identify areas for improvement. Lastly, we aimed to measure the effectiveness of the extension, investigating whether it enhances the collaborative learning experience for students and supports instructional tasks for instructors.

In the subsequent sections, we will provide a detailed discussion of our procedure, outlining the steps undertaken for each user testing session.

9.1 Student Perspective

To gain insights into how students perceive the system, we conducted an evaluation process and collected quantitative data in the form of a post-test assessment of usability using the System Usability Scale (SUS), as well as qualitative data through a questionnaire.

9.1.1 Recruitment

The selection process involved a signup form that was accessible to lab members. We successfully recruited a total of 7 participants (5 females and 2 males), with all but one participating in person.

9.1.2 Apparatus

The evaluation was conducted using the system outlined in section 8. Each participant utilized their own machine, working with their preferred browser and operating system. All participants chose Google Chrome as their browser, and both Windows and macOS were used as operating systems.

9.1.3 Procedure

Participants were given a brief presentation on the thesis and the developed system, followed by an introduction to the tasks. The testing session encompassed all three types of exercises. For each task, participants were not provided with the answers, and the role of the instructor was assigned to one member of the project team. To minimize delays, a ".puzzle" file was prepared in advance.

In the text response exercise, participants were initially tasked with writing a free-text response to a prompt (see Appendix C.1.1). After completing their responses, participants submitted them for evaluation. The instructor then reviewed each submission, provided feedback, and presented a master solution to the participants.

In the multiple-choice exercise, participants answered a multiple-choice question (refer to Appendix C.1.2), with the option to select multiple answers. After 1 minute, the correct answer and the distribution of selections were displayed to the participants.

For the coding exercise, a pre-selected participant with Python coding knowledge wrote code and shared their screen with all participants. Other participants proposed new assertion codes in the form of simple yet meaningful asserts. This approach ensured that Python coding knowledge was not required for participation in the evaluation. The participant responsible for coding was provided with the master solution (refer to Appendix C.1.3) and adjusted the code iteratively based on new test cases that initially failed. This collaborative approach enabled real-time testing and refinement of the code, incorporating diverse input from participants with varying levels of coding expertise. Finally, the instructor showed the solutions to all participants, allowing them to validate and compare their code using a diff editor.

After completing the tasks, participants were asked to fill out a System Usability Scale (SUS) assessment [5] and a questionnaire with seven questions (refer to Appendix C.1.4).

9.2 Instructor Perspective

For evaluating the instructor interfaces, we conducted a heuristic evaluation following the guidelines outlined in the paper "How to Conduct a Heuristic Evaluation" by Jakob Nielsen [38]. We considered the original ten heuristics introduced by Jakob Nielsen [39].

9.2.1 Recruitment

Participants in the evaluation were involved both on-site and remotely. They were selected through a signup process via an online form. A total of 4 participants (2 males and 2 females) were recruited for the evaluation procedure.

9.2.2 Apparatus

The evaluation was carried out on the system described in section 8. Each participant used their own laptop and chose their preferred browser and operating system for evaluation. All participants operated on a Windows operating system, with one using Brave browser and the remaining participants using Google Chrome.

9.2.3 Procedure

First, the participants were introduced to the ten heuristics relevant for the evaluation, and each was provided with a heuristic-evaluation workbook in the form of a Google form following the workbook created by the Nielsen Norman Group [21]. This form allowed participants to document their findings for each task. The participants were then asked to evaluate the interface on their own. Each task was performed twice in a row. The first round served as a practice session to get familiar with the system. In the second pass, participants identified design elements, features, or decisions that violated one of the ten heuristics. After completion of the independent evaluation, the workbook was collected to synthesize the identified issues. For all tasks (see appendix C.2), a member of the project played the role of the student.

10 Results

10.1 Evaluation of the student perspective

10.1.1 Qualitative Questionnaire

The questionnaire results indicated that the multiple-choice and text response exercises were perceived as intuitive components of the interface. Participants also positively evaluated the feedback box containing the instructor's comments and the markdown editor.

However, the coding exercise, particularly the test case functionalities, were found to be confusing for most participants. The purpose and operation of buttons for adding, verifying, and deleting test cases, as well as running the student's code and submitting solutions, were not clearly understood. Additionally, the absence of visual indicators for the state of test cases left participants uncertain about whether a test case had been verified. There was also confusion regarding the input box for the test name and the code editor for assertion code, with many participants mistakenly entering assertion code into the test name field.

The testing session also revealed several technical issues. For instance, exercises sometimes failed to appear when initially made visible to participants, suggesting a synchronization problem. Exercises only became visible after refreshing the webpage on each client. Moreover, during the coding exercise, the verification function was disabled for all participants after the first test case was verified, preventing further test case verification. Some

participants encountered difficulties with creating new test cases, and one participant reported that their test case was overwritten by another participant's submission.

Participants proposed several improvements, such as redesigning the test case verification and code submission functionalities to ensure clearer separation of their roles. They also recommended prepopulating test cases with default code that includes the assert keyword. In addition, the participants called for more detailed indicators on the status of the submitted solutions and the verification of test cases. Participants suggested allowing students to add additional test cases after submission to help peers who had not yet finished their work.

In terms of desired features, participants expressed a preference for having aggregated information on how many test cases passed and how many students out of the total could or could not pass a test case. They also highlighted the potential benefit of integrating large language models (LLMs) to automatically generate test cases.

Lastly, participants provided suggestions to enhance the collaborative learning experience. One recommendation was to implement a collaborative coding mode that pairs two students together: one student writes the code while the other attempts to break it by adding new and more complex test cases. Another suggestion was to incorporate gamification mechanisms, such as a leaderboard. Additionally, participants recommended enabling students to provide feedback on each other's submissions and displaying the number of users currently working on the exercise.

10.1.2 SUS Questionnaire

The evaluation of the SUS questionnaire, presented in Figure 15, effectively highlights the issues of the interface mentioned above. However, there are some discrepancies between the opinions of the participants reflected in the statements S3, S4, S5, S7, S9 and S10. These differences in perception might be related to participants' familiarity with similar interfaces, such as the JupyterLab interface, as we also adapted some of its visual principles. For statements S1, S2, and S8, most participants demonstrated a high level of agreement with each other. The average SUS score is approximately 58.92, while the mean SUS score is calculated to be 65, indicating the presence of potential outliers. The SUS score ranges from 17.5 to 87.5 with a standard deviation of 19.99 confirming the discrepancies. According to the benchmark by James R. Lewis et al. [31], a score above 68 indicates an average user experience, while a score above 80 suggests a good user experience. A score below 68, suggests that our extension needs improvement.

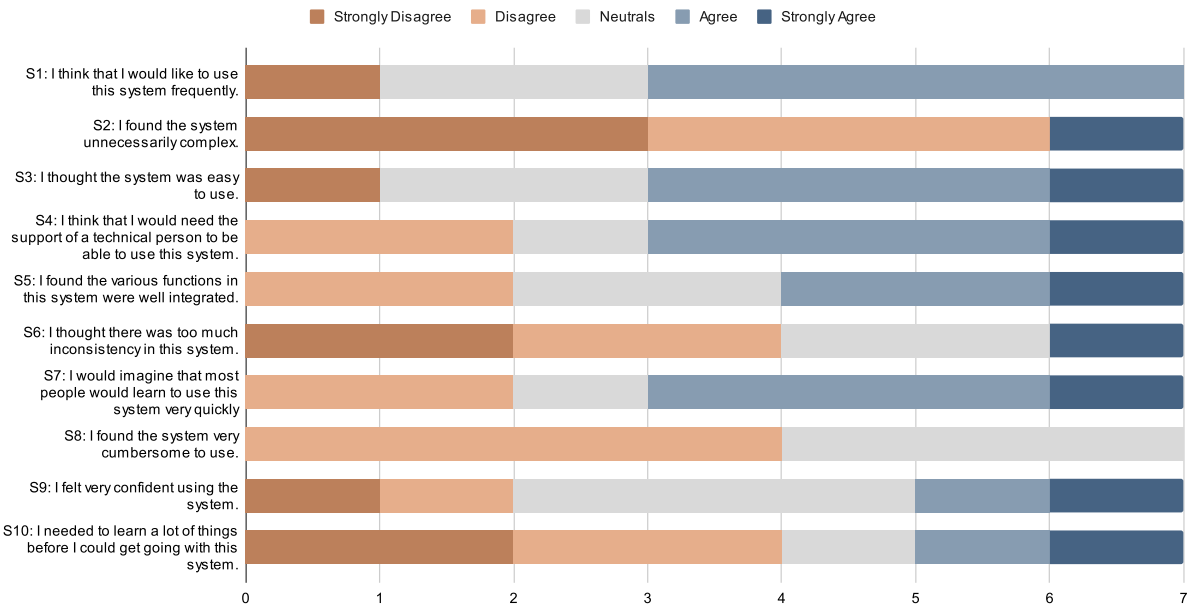

Figure 15: System usability scale chart. The chart visualizes participants' responses to each SUS statement.

10.2 Heuristic evaluation of the instructor perspective

The heuristic evaluation highlighted issues that align with those discussed in the qualitative questionnaire from the student's perspective (see Section 10.1.1). This overlap arises because some features and visual elements are shared between the student and instructor views, leading to similar concerns being identified from both perspectives. Nevertheless, the evaluation revealed new issues specifically related to the instructor UI. The participants recommended implementing warnings for critical actions, such as deleting exercises or making them visible, to prevent accidental changes. Additionally, an undo functionality was suggested to allow instructors to revert unintended modifications. Furthermore, the evaluation uncovered confusion or absence of critical state indicators, such as the visibility status of exercises, the instructor's comments, the solution display status, and the student's solution. Participants recommended implementing color-coding, icons, or text messages to clarify these states. Moreover, a majority of participants found the commenting function non-intuitive and suggested incorporating functionalities to change visibility, as well as to add or remove comments entirely. Further requested features included options to mark solutions as correct or incorrect, shortcuts for various functionalities, and indicators to show whether a student is actively working on an exercise. Specifically for the multiple-choice exercises, participants noted that reordering options was not very user-friendly. They suggested that adding animation or enabling a drag-and-drop mechanism would significantly improve the user experience. In addition, a recommendation was to include an indicator showing how many students have submitted their selections, which could provide valuable feedback to instructors. Concerning the coding exercise, participants reported confusion regarding the visual separation between the starting code and the solution code, since both are concatenated during execution. They suggested implementing a clearer visual grouping. Lastly, implementing a real-time compilation check while editing both the starting code and the solution code would enhance the user experience. This feature would allow the instructor to identify errors and issues immediately.

11 Discussion

Our testing session revealed that while some participants had a positive experience with the interface, there were clear areas that needed improvement, particularly regarding technical issues, collaborative features, and user interface design. Despite these challenges, the system demonstrated significant potential for enhancing in-class programming courses. In this section, we reflect on the system's design and discuss its implications for future educational software development in the context of Human Computer Interaction (HCI).

11.1 Implications

11.1.1 User Testing Findings

Firstly, it is essential to address bugs related to synchronization issues and the coding exercise, especially concerning the test case feature. Secondly, the feedback highlights the critical role of intuitive and effective UI design. Clear indicators and a well-designed interface are essential for ensuring that users can navigate and utilize the system efficiently. Therefore, a comprehensive enhancement of the user interface is necessary to improve the overall user experience and communicate the system state more effectively. Lastly, expanding existing features and incorporating new ones will further enhance the system's functionality and user engagement.

Note that the synchronization issues were resolved, and several UI improvements were implemented before the submission of this thesis.

11.1.2 Enhanced Learning Experience

The integration of interactive and real-time collaborative features within the system proved valuable for both instructors and students. In particular, the live peer testing introduced by PuzzleMe benefits instructors by saving time and allowing improvised exercises, while helping students verify their understanding and practice test-driven development [50]. In addition, providing individual real-time feedback enhances the learning experience by helping students compare their solutions with the master solution. This personalized feedback aids students in identifying and addressing gaps in their understanding [22]. Designers of future collaborative learning platforms should consider incorporating similar interactive and feedback-driven elements to maximize educational benefits.

11.1.3 Integration into the Jupyter Ecosystem

The integration of our system into the Jupyter ecosystem allows for scalability and adaptability across different educational contexts. The extension can be adapted to various course formats, from introductory programming classes to advanced data science modules. By leveraging Jupyter's existing infrastructure, our system can be seamlessly integrated into educational settings where Jupyter is already in use. This minimizes the learning curve for both instructors and students, as they are already familiar with the core functionalities of Jupyter notebooks. The open source nature of our system encourages community-driven development and continuous improvement. Educators and developers can contribute to the codebase, suggest enhancements, and share best practices.

11.1.4 Further Libraries

By separating quiz UI components from the extension codebase and creating a generic framework for managing normalized data in Yjs, we contributed to the modularity and reusability of our system. This separation allows the quiz components to be used independently in other projects, enhancing their utility beyond our specific application. Additionally, the framework for managing normalized data in Yjs provides a robust solution for handling complex

data structures in collaborative environments, which can be adopted by other developers and educators to build more efficient and scalable applications.

11.2 Future Work

11.2.1 Exercise Types

Future work could encompass the development of additional exercise types, such as simple yes/no quizzes, picture-based quizzes, and more sophisticated exercises in which students are given incomplete code snippets to complete, thus improving their understanding of the programming syntax. Furthermore, existing exercise types could be refined and new functionalities introduced. For example, the instructor-student interaction facilitated through the comment feature could be enhanced to support bidirectional communication similar to Codeopticon[22], allowing students to pose questions and allowing real-time interaction during the exercise, not solely post-submission.

The integration of automated grading for specific types of exercise could provide immediate feedback, streamline the evaluation process, and reduce the workload of instructors. Furthermore, implementing adaptive difficulty levels based on individual student performance could tailor the learning experience to accommodate both novice and advanced learners.

In terms of multiple choice questions, an optional configuration could be added to require students to justify their selected answers, promoting critical thinking and deeper engagement with the material. For free-text response exercises, the inclusion of hints and word limits could encourage more concise writing.

The coding exercise currently lacks support for advanced testing techniques such as exceptions, callbacks, and dynamic tests. Integrating these features would enhance the functionality and provide a more comprehensive evaluation.

The architecture outlined in this thesis provides a robust framework that facilitates the seamless extension of the existing system and the integration of new features.

In our opinion, the potential for further development in this area is substantial.

11.2.2 More Collaborative Features

The extension currently offers a peer assessment feature in the form of live peer testing. As a next step, the live peer code review mechanism introduced by PuzzleMe could be implemented. This mechanism would intelligently group students based on their solutions, allowing them to discuss and review each other's work [50]. Such an approach could enhance their understanding of the code. Furthermore, the qualitative questionnaire 10.1.1 revealed additional ideas. One promising suggestion was the introduction of a collaborative coding mode that pairs students together, with one student assigned the coding task and the other tasked with attempting to break the code by creating test cases, similar to CodeDefenders[41]. Other suggestions included potential enhancements for the collaborative immersive environment, such as displaying the presence of other students and providing a feedback system. These features would promote a sense of community and enable students to offer constructive feedback to one another.

11.2.3 Enhancing Security and Privacy

Since the presented system operates as a frontend extension, it does not guarantee security and privacy. Essentially, any user with sufficient technical knowledge could potentially access data from the ".puzzle" file. This vulnerability means that a malicious student might be able to view the master solution or the solutions of other students. Furthermore, the current implementation does not prevent a malicious user from altering the contents of the ".puzzle" file or modifying its data structure, which could lead to corrupted or erroneous files.

To address these security concerns, the implementation of a JupyterLab server extension is necessary. Such an extension could enforce controlled access and modifications to the ".puzzle" file, potentially incorporating encryption schemes to protect solutions from unauthorized users. Further research and development in this area are required to establish robust security measures and ensure the integrity of the data.

Additionally, the current implementation does not adequately protect user privacy. Both created exercises and submitted solutions currently store usernames. To enhance user privacy, a server-side anonymization process for usernames could be implemented.

11.2.4 Integration of LLMs

Integrating Large Language Models (LLMs) into the extension could significantly enhance its functionality. LLMs could be utilized for various purposes, such as generating exercises, providing personalized feedback to students, and assessing the correctness of student solutions even in the absence of predefined test cases. Additionally, LLMs could evaluate the meaningfulness of test cases, contributing to a more intelligent learning environment. The potential of LLMs in this context is substantial, but further evaluation is necessary to assess their technical feasibility and integration within the system.

12 Conclusion

This thesis introduces a collaborative learning extension for JupyterLab, specifically designed to manage quizzes and assess student solutions at scale through live peer testing. In addition, we present a React Quiz UI Component Library and a Yjs Framework Library for managing normalized semi-structured data. Furthermore, we introduce the integration of a multi-user JupyterHub instance into the ETH environment. Our testing sessions revealed both technical issues and potential enhancements, while also demonstrating the system's promise. All contributions are publicly available, paving the way for further development and research in this domain.

A React Quiz UI

A.1 Implementation

The IExerciseObject interface and IExerciseAnswer interfaces are defined as follows:

```
interface IExerciseObject {
  metadata?: Record<string, any>;
}
interface IExerciseAnswer {
  referenceId?: string;
  answer: any;
}
```

B JupyterLab Extension

B.1 UI Components:

B.1.1 Common Components

- **KernelOutput, KernelOutputContainer, and CompilingKernelOutputContainer**: Displays kernel outputs in the specified order with appropriate styling, e.g., an error message implies a red background color. The *CompilingKernelOutputContainer* additionally indicates the compilation status.
- **BaseButton and SubmitButton**: The *BaseButton* ensures consistent styling for all buttons displayed in the extension. The *SubmitButton* extends *BaseButton* by adding another feature: a "submitted" badge is displayed when the button is pressed.
- **Code and Markdown**: These are wrappers of the corresponding components of the react-quiz-library that configure the provided components to be suitable for the extension.
- **Content and ContentBody**: Designed to be used together, these components apply a border and standardized padding around the content. They also enable hovering and interaction within the *Content* component.
- **Toolbar, ToolbarButton, and ToolbarToggle**: Common toolbar components for interacting with cells.
- **Indicators**: Components to show various statuses or notifications.
- **Feedback**: Components to collect and display user feedback.
- **HTML Sanitizer**: Ensures that HTML content is sanitized for security.
- **Tabs**: Components to manage tabbed content within the extension.

B.1.2 Exercise Components

- **Coding**: A container component defining what components can be included within a coding exercise component.
- **CodeElement**: A code editor with a label specifying the purpose of the editor.

- **MultipleChoiceInstructor, MultipleChoiceInstructorItem, and MultipleChoiceInstructorConfigItem**: Instructor components for creating and editing multiple choice exercises.
- **MultipleChoiceStudent**: This component wraps the multiple-choice component of the react-quiz-ui library and adds a submit button.
- **TextResponseStudent**: This component wraps the text response component of the react-quiz-ui library, adding a feedback container and a submit button.

C Evaluation

C.1 Student Perspective

C.1.1 Text Response Exercise

Prompt: Explain the importance of feedback in user interface design. Provide an example of how feedback enhances user experience.

Solution: Feedback in user interface design is crucial because it helps users understand the system's response to their actions, ensuring they feel confident and in control. It confirms actions, prevents errors, guides users, enhances satisfaction, and aids learning. Example: In an online form, real-time validation (e.g., highlighting an incorrectly formatted email address) prevents errors. Upon submission, a loading icon and success message ("Thank you for registering!") confirm that the form was processed correctly. This immediate feedback improves usability and user satisfaction by making interactions clear and reassuring.

C.1.2 Multiple Choice Exercise

Question: Which stage of the HCI design process involves creating prototypes to test with users?

Options:

- Analysis
- Design (correct answer)
- Evaluation
- Implementation

C.1.3 Coding Exercise

Exercise: Write a function format_username(username) that takes a single username as input and returns a formatted version of the username based on the following rules:

- Remove any leading or trailing whitespace from the username.
- If the username is empty after removing whitespace, consider it invalid (return "Invalid username").
- If the username is between 1 and 20 characters long (inclusive), return it unchanged.
- If the username is longer than 20 characters, truncate it to 20 characters and add ...
- Consider usernames containing special characters (@, #, $, %, etc.) invalid (return "Invalid username")

Solution:

```
def format_username(username):
    username = username.strip()

    if len(username) == 0:
        return "Invalid username"

    if any(char in username for char in '@#$%&*'):
        return "Invalid username"

    if len(username) <= 20:
        return username
    else:
        return username[:20] + "..."
```

C.1.4 Questionnaire

- What aspects of the interface did you find most intuitive?
- Were there any features that you found confusing or difficult to use?
- Have you encountered any bugs? If yes, describe them.
- Did you notice any performance issues while using the extension? If yes, when did they occur?
- What improvements would you suggest for the extension?
- Are there any features you would like to see added?
- How could the collaborative learning experience be enhanced?

C.2 Instructor Perspective

C.2.1 Task: Create a Text Response Exercise

1. Create a text-response exercise asking students: What's the capital of France?
2. Add the solution: Paris
3. Make the exercise visible to the students
4. Wait for submission and observe the student answering the question
5. Add the comment "Good job!" to the answer of the student
6. Show the solution to the students

C.2.2 Task: Create a Multiple Choice Exercise

1. Create a multiple choice exercise asking students: Which of the following are fruits?
2. Add the options: (a) Apple, (b) Carrot, (c) Banana, (d) Potato
3. Allow multi selection
4. Reorder the options to obtain the following order: c, a, d, b
5. Select the correct answers: a, c
6. Make the exercise visible to the students
7. Wait and observe the student selecting the answer
8. Show the solution to the students
9. Move the exercise above the text response exercise
10. Delete the multiple choice exercise

C.2.3 Task: Create a Coding Exercise

1. Create a coding exercise asking students: Write a Python function *add_numbers* that takes two numbers as input and returns their sum.
2. Add the starting code: def add_numbers(a, b):
3. Add the solution code: return a + b
4. Add and verify an assertion code: assert 4 == add_numbers(2,2)
5. Configure: Student should have to submit one testcase before starting coding
6. Make the exercise visible to the students
7. Wait for submission and observe the student coding
8. Show the solution to the students

References

[1] 2024. Monaco Editor Theme. *documentation* (2024). https://microsoft.github.io/monaco-editor/docs.html#interfaces/editor.IStandaloneThemeData.html.

[2] Abdulmalek Al-Gahmi, Yong Zhang, and Hugo Valle. 2022. Jupyter in the Classroom: An Experience Report. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1* (Providence, RI, USA) *(SIGCSE 2022)*. Association for Computing Machinery, New York, NY, USA, 425–431. https://doi.org/10.1145/3478431.3499379

[3] Nicola Antonio Roberto Amato. 2023. Mastering database normalization: A comprehensive exploration of normal forms. (10 2023).

[4] Lorena A Barba, Lecia J Barker, Douglas S Blank, Jed Brown, Allen B Downey, Timothy George, Lindsey J Heagy, Kyle T Mandli, Jason K Moore, David Lippert, et al. 2019. Teaching and learning with Jupyter. *Recuperado: https://jupyter4edu. github. io/jupyter-edu-book* (2019), 1–77.

[5] John Brooke. 1995. SUS: A quick and dirty usability scale. *Usability Eval. Ind.* 189 (11 1995).

[6] Julia Cambre, Scott Klemmer, and Chinmay Kulkarni. 2018. Juxtapeer: Comparative Peer Review Yields Higher Quality Feedback and Promotes Deeper Reflection. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3173868

[7] Project Jupyter Contributors. 2024. IBaseCell. *documentation* (2024). https://jupyterlab.readthedocs.io/en/stable/api/interfaces/nbformat.IBaseCell.html.

[8] Project Jupyter Contributors. 2024. JupyterHub DockerSpawner Documentation. *documentation* (2024). https://jupyterhub-dockerspawner.readthedocs.io/en/latest/.

[9] Project Jupyter Contributors. 2024. JupyterHub Documentation. *documentation* (2024). https://jupyterhub.readthedocs.io/en/stable/index.html#.

[10] Project Jupyter Contributors. 2024. JupyterLab Documentation. *documentation* (2024). https://jupyterlab.readthedocs.io/en/stable/index.html.

[11] Project Jupyter Contributors. 2024. JupyterLab Extension: Lumino Singals. *documentation* (2024). https://jupyterlab.readthedocs.io/en/stable/extension/virtualdom.html.

[12] Project Jupyter Contributors. 2024. UI Components library of Jupyter. *documentation* (2024). https://jupyterlab.readthedocs.io/en/stable/api/modules/ui_components.html.

[13] Wikipedia Contributors. 2023. HTML sanitization. *encyclopedia* (2023). https://en.wikipedia.org/wiki/HTML_sanitization.

[14] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. CodeWrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 471–476. https://doi.org/10.1145/1953163.1953299

[15] Dan Abramov et. al. 2024. Redux Documentation: Concepts. *documentation* (2024). https://redux.js.org/tutorials/essentials/part-1-overview-concepts.

[16] Dan Abramov et. al. 2024. Redux Documentation: Normalization. *documentation* (2024). https://redux.js.org/usage/structuring-reducers/normalizing-state-shape.

[17] Pouya Saadeghi et. al. 2024. Daisy UI: Themes. *documentation* (2024). https://daisyui.com/docs/themes/.

[18] Mozilla Foundation. 2024. Javascript: eval method. *documentation* (2024). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval.

[19] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (mar 2015), 35 pages. https://doi.org/10.1145/2699751

[20] Brian E. Granger and Fernando Pérez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science Engineering* 23, 2 (2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

[21] Nielsen Normal Group. 2024. Heuristic Evaluation Workbook. *Workgroup* (2024). https://media.nngroup.com/media/articles/attachments/Heuristic_Evaluation_Workbook_1_Fillable.pdf.

[22] Philip J. Guo. 2015. Codeopticon: Real-Time, One-To-Many Human Tutoring for Computer Programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology* (Charlotte, NC, USA) *(UIST '15)*. Association for Computing Machinery, New York, NY, USA, 599–608. https://doi.org/10.1145/2807442.2807469

[23] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale* (Cambridge, Massachusetts, USA) *(L@S '17)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[24] Tailwind Labs Inc. 2024. Tailwind. *documentation* (2024). https://tailwindcss.com.

[25] Veeramreddy Jagadishreddy, Bandari Madhu, Aditya Banik, Chaitanya Vatluri, Sumith Shivam, et al. 2023. Enhancing Computer Science Education: The Benefits of Real-Time Collaborative Code Editors in Schools and Universities. (2023).

[26] Jeremiah W. Johnson. 2020. Benefits and Pitfalls of Jupyter Notebooks in the Classroom. In *Proceedings of the 21st Annual Conference on Information Technology Education* (Virtual Event, USA) *(SIGITE '20)*. Association for Computing Machinery, New York, NY, USA, 32–37. https://doi.org/10.1145/3368308.3415397

[27] Kenneth R. Koedinger Kelly Rivers. [n. d.]. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor - International Journal of Artificial Intelligence in Education — link.springer.com. https://link.springer.com/article/10.1007/s40593-015-0070-z.

[28] Sean Kross and Philip J. Guo. 2019. Practitioners Teaching Data Science in Industry and Academia: Expectations, Workflows, and Challenges. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3290605.3300493

[29] Chinmay Kulkarni, Koh Pang Wei, Huy Le, Daniel Chia, Kathryn Papadopoulos, Justin Cheng, Daphne Koller, and Scott R. Klemmer. 2013. Peer and self assessment in massive online classes. *ACM Trans. Comput.-Hum. Interact.* 20, 6, Article 33 (dec 2013), 31 pages. https://doi.org/10.1145/2505057

[30] Chinmay E. Kulkarni, Michael S. Bernstein, and Scott R. Klemmer. 2015. PeerStudio: Rapid Peer Feedback Emphasizes Revision and Improves Performance. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale* (Vancouver, BC, Canada) *(L@S '15)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/2724660.2724670

[31] James Lewis and Jeff Sauro. 2018. Item Benchmarks for the System Usability Scale. *Benchmark* 13 (05 2018), 158–167.

[32] Pascal Linder. 2024. Collaborative Learning Extension: Product Page. https://eth-peach-lab.github.io/collaborative-learning-extension-jupyter/.

[33] Pascal Linder. 2024. React Quiz UI Library: npm. https://www.npmjs.com/package/react-quiz-ui.

[34] Pascal Linder. 2024. Yjs Normalized Library: npm. https://www.npmjs.com/package/yjs-normalized.

[35] James E. McDonough. 2017. *Factory Design Patterns*. Apress, Berkeley, CA, 173–190. https://doi.org/10.1007/978-1-4842-2838-8_14

[36] Dmitri Nesteruk. 2019. *The SOLID Design Principles*. Apress, Berkeley, CA, 3–25. https://doi.org/10.1007/978-1-4842-4366-4_1

[37] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. 39–49. https://doi.org/10.1145/2957276.2957310

[38] Jakob Nielsen. 1995. How to conduct a heuristic evaluation. *Nielsen Norman Group* 1, 1 (1995), 8.

[39] Jakob Nielsen. 2005. Ten usability heuristics. *Nielsen Norman Group* (2005).

[40] OpenAI. 2024. ChatGPT. Large language model.

[41] José Miguel Rojas, Thomas D. White, Benjamin S. Clegg, and Gordon Fraser. 2017. Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 677–688. https://doi.org/10.1109/ICSE.2017.68

[42] Marc J. Rubin. 2013. The effectiveness of live-coding to teach introductory programming. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*. Association for Computing Machinery, New York, NY, USA, 651–656. https://doi.org/10.1145/2445196.2445388

[43] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[44] J. Sitthiworachart and M. Joy. 2003. Web-based peer assessment in learning computer programming. In *Proceedings 3rd IEEE International Conference on Advanced Technologies*. 180–184. https://doi.org/10.1109/ICALT.2003.1215052

[45] Joanna Smith, Joe Tessler, Elliot Kramer, and Calvin Lin. 2012. Using peer review to teach software testing. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (Auckland, New Zealand) *(ICER '12)*. Association for Computing Machinery, New York, NY, USA, 93–98. https://doi.org/10.1145/2361276.2361295

[46] Meta Open Source. 2024. Passing Data Deeply with Context. *documentation* (2024). https://react.dev/learn/passing-data-deeply-with-context.

[47] Bootstrap Team. 2024. Bootstrap Colours. *documentation* (2024). https://getbootstrap.com/docs/5.0/customize/color/.

[48] Kim Technology. 2015. *Learnersourcing : improving learning with collective learner activity.* Ph. D. Dissertation.

[49] S. Trahasch. 2004. From peer assessment towards collaborative learning. In *34th Annual Frontiers in Education, 2004. FIE 2004.* F3F–16. https://doi.org/10.1109/FIE.2004.1408638

[50] April Yi Wang, Yan Chen, John Joon Young Chung, Christopher Brooks, and Steve Oney. 2021. PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2, Article 415 (oct 2021), 24 pages. https://doi.org/10.1145/3479559

[51] Alvin Yuan, Kurt Luther, Markus Krause, Sophie Isabel Vennix, Steven P Dow, and Bjorn Hartmann. 2016. Almost an Expert: The Effects of Rubrics and Expertise on Perceived Value of Crowdsourced Design Critiques. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing* (San Francisco, California, USA) *(CSCW '16)*. Association for Computing Machinery, New York, NY, USA, 1005–1017. https://doi.org/10.1145/2818048.2819953