This report details the design, development, and key learnings from a proof-of-concept project to create Ulti-Quest, a full-stack Question Bank Management System (QBMS). The system is designed to import, manage, and render educational assessment questions originally created for the PrairieLearn platform. The project's central technical achievement is a hybrid rendering engine that interoperates between Node.js and Python to process PrairieLearn's custom HTML elements. The development process was executed in stages, progressively adding features such as a user-friendly zip file import pipeline, integration with an external AI-powered tagging service, and an evolution of the data model from a linear to a hierarchical tag structure. This report is divided into two primary sections: a detailed account of the work completed ("What I Did") and a reflection on the knowledge acquired throughout the project ("What I Learned").

What I Did

Over the course of this project, I developed a full-stack web application from the ground up, evolving it through two stages of functionality and complexity. The technology stack chosen was the MERN stack: MongoDB for the database, Express.js for the backend framework, React for the client-side user interface, and Node.js for the server runtime environment.

Stage 1: Foundational QBMS with Linear Tagging

The initial phase of the project focused on building the core infrastructure and establishing the baseline features required for a functional question bank.

Initial Data Population: To populate the system with an initial dataset, I developed a web scraper using Python and the Selenium library. This script was engineered to navigate to a live PrairieLearn course, authenticate by leveraging an existing browser session to bypass the need for programmatic login, and systematically extract essential question metadata. This included each question's unique ID, title, topic, URL, and its native PrairieLearn tags. The collected data was then serialized into a JSON file, which was manually imported into the MongoDB database to serve as the application's initial content.

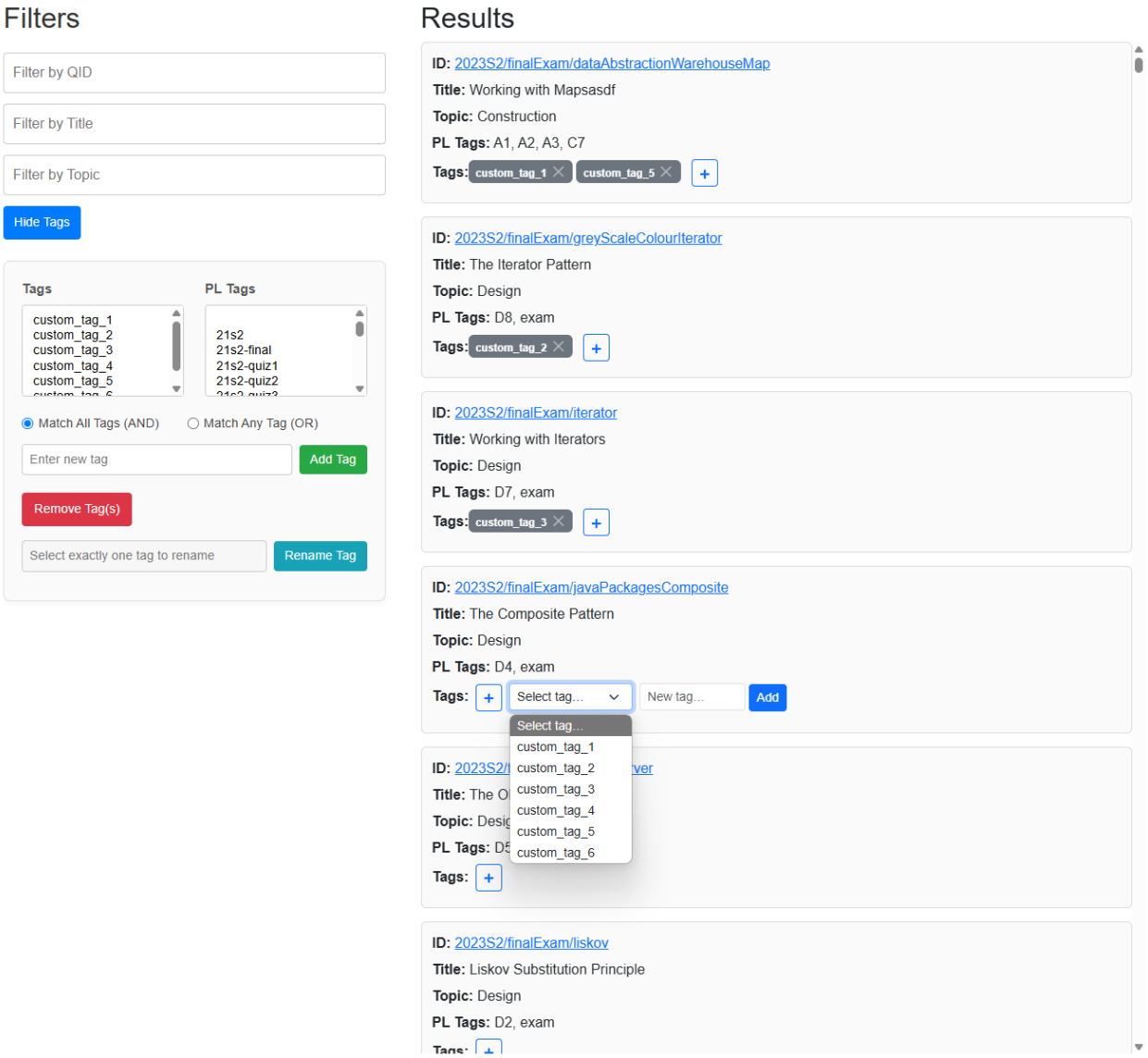
Core Web Application: With a dataset in place, I constructed the initial client-server application. The Node.js backend exposed a RESTful API to query the question collection in MongoDB. The React frontend consumed this API to fetch and display a filterable list of all questions, providing a simple interface for browsing the available content.

Linear Tag Management: The first version of the custom tagging system was implemented. This feature empowered users to create, edit, and delete their own text-based tags and associate them with one or more questions. In this initial stage, the tags were represented in a simple linear structure - essentially an array of strings associated with each question document. This provided a layer of organization on top of the pre-existing PrairieLearn tags.

Filtering and Search: To make the question bank useful, I implemented a multi-faceted filtering system. The user interface included controls that allowed users to search and narrow down the question list

based on a combination of criteria, including question ID, title, PrairieLearn-native tags, and the newly implemented custom tags.

Foundational QBMS with Linear Tagging



Stage 2: Advanced Rendering Pipeline and Hierarchical Tagging

The goal was to transform the system from a simple metadata manager into a system capable of rendering the full content of PrairieLearn questions, which was a non-trivial challenge. **This stage of the project was completed with the assistance of the AI coding tool, Cursor.**

The Hybrid Rendering Engine: The core challenge was rendering PrairieLearn's custom HTML tags in a standard browser. Instead of rewriting the platform's logic, I developed a hybrid rendering engine that uses the Node.js backend to orchestrate the execution of PrairieLearn's original Python scripts. This "emulation via orchestration" approach formed the technical heart of the project.

- **File Storage:** A critical prerequisite was a robust file storage strategy. Upon import, each question's files are separated into two locations. All original source files, including server-side

logic, are stored in a private *server/question_source* directory. Correspondingly, any client-facing assets like images are copied to a *server/public/question_assets* directory, making them web-accessible. This separation mirrors PrairieLearn's security model, ensuring server code is never exposed.

- **Multi-Stage Rendering:** The rendering process is a carefully sequenced pipeline, managed by a Node.js orchestrator that communicates with Python scripts via a custom bridge using JSON over standard I/O. At a high level, the steps are as follows:
  1. Data Generation: The process begins by executing the question's generate script to produce the initial data and parameters.
  2. Templating: This data is then used with a Mustache template to render the question's base HTML structure.
  3. Pre-processing: The resulting HTML is scanned for any markdown content, which is converted into standard HTML.
  4. Element Preparation: The pipeline performs a bottom-up traversal of the HTML. For each custom <pl-*> element it finds, it executes a prepare function, which allows child elements to register data before their parents are processed.
  5. Iterative Rendering: Next, an iterative top-down render phase begins. In a loop, the system finds all remaining custom <pl-*> elements and executes their render functions, replacing them with standard HTML. This loop continues until no custom elements are left, which correctly handles cases where elements render other elements.
  6. Finalization: The final HTML string is then saved to the database, ready to be served to a user.

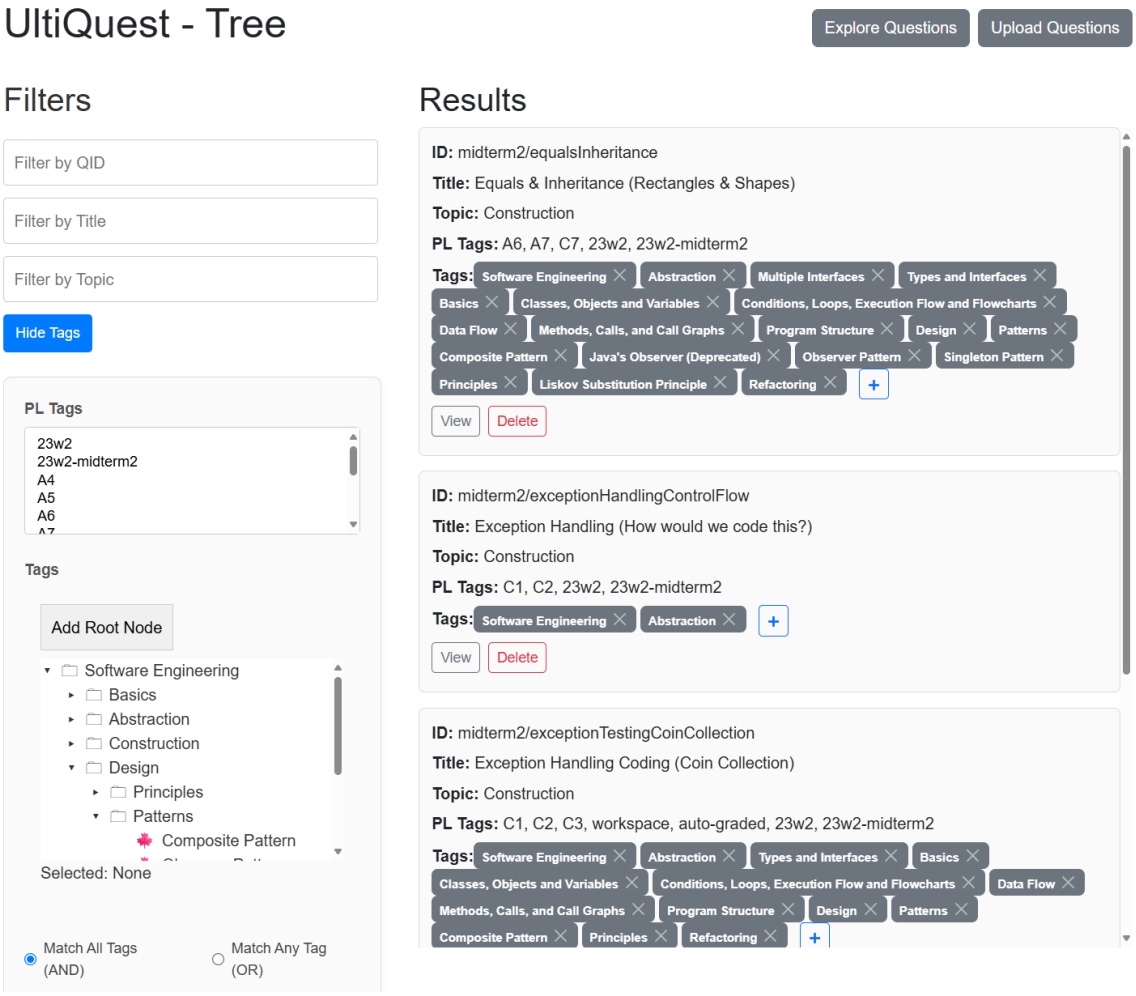
Zip File Import: To streamline the process of adding new questions, the manual data loading process was replaced with a user-friendly zip file uploader. This feature, implemented in the *client/src/components/ZipUploader.js* React component, leverages the JSZip library to process .zip files directly in the browser. The client-side logic unzips the archive, identifies the constituent files for each question, handles binary assets like images by Base64 encoding them, and constructs a structured JSON payload for each question to be sent to the backend. This approach significantly improves the user experience and reduces server-side load.

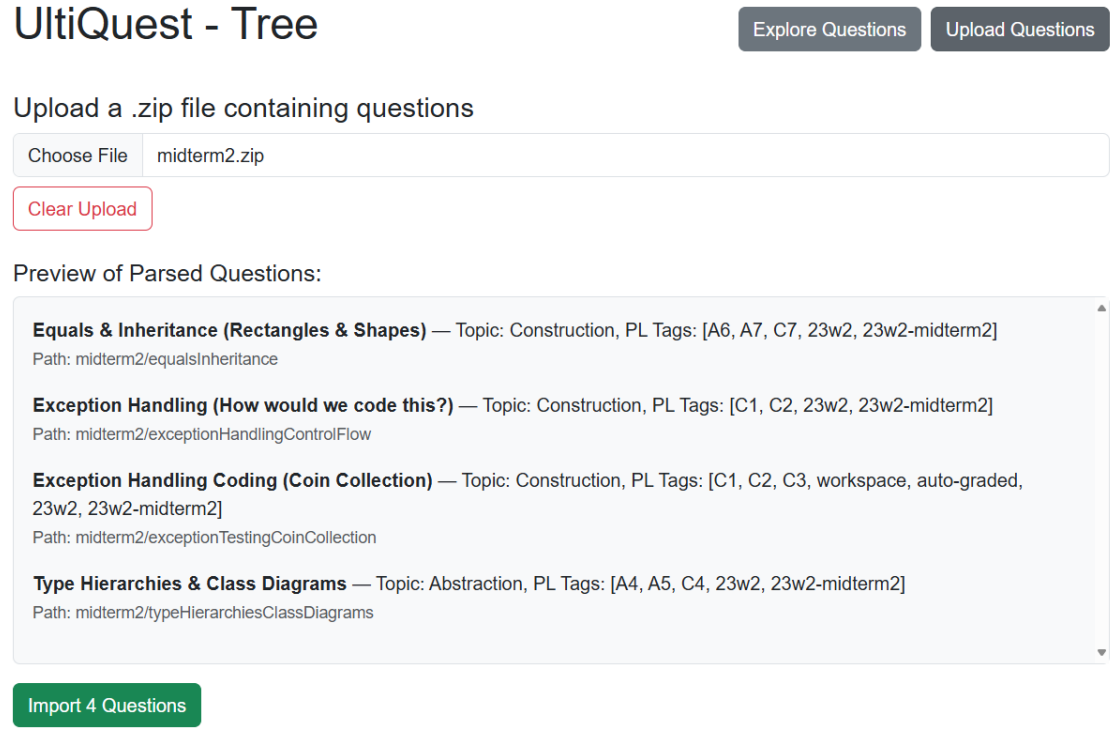
AI Tagger Integration: A specific step in the question rendering process was designated to pass an intermediate HTML representation of each question to an external AI tagging service endpoint. This allowed for the integration of automatic tag generation as part of the question import workflow.

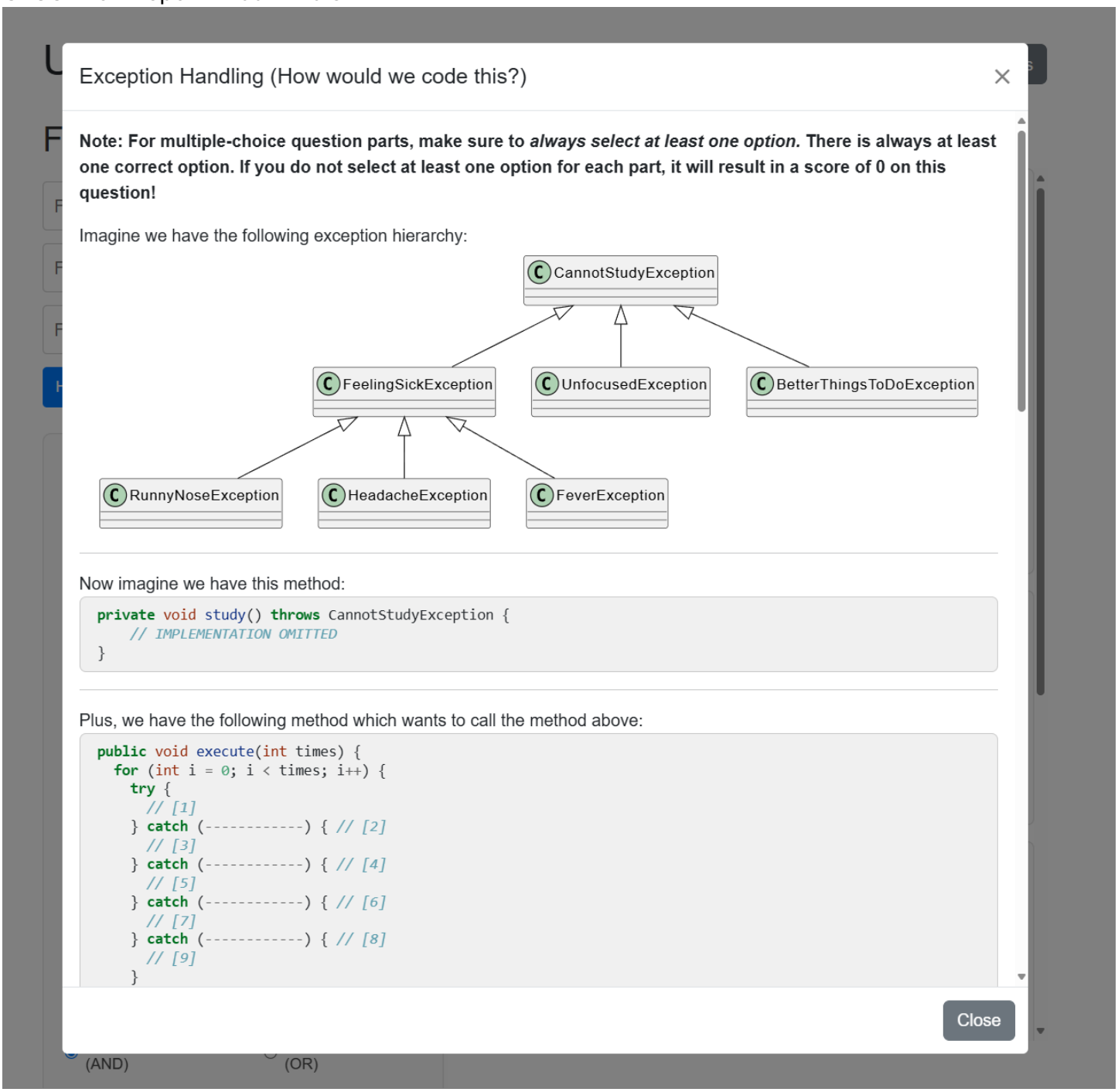
Hierarchical Tagging: The data model for tags was upgraded from a linear list to a tree structure. This allowed for the creation of parent-child relationships between tags, enabling a much more granular and logical categorization of questions (e.g., Computer Science → Data Structures → Trees → Binary Search Trees).

CPSC 448 – Report – Adam Mais

QBMS with hierarchical tagging, question upload, and question visualization





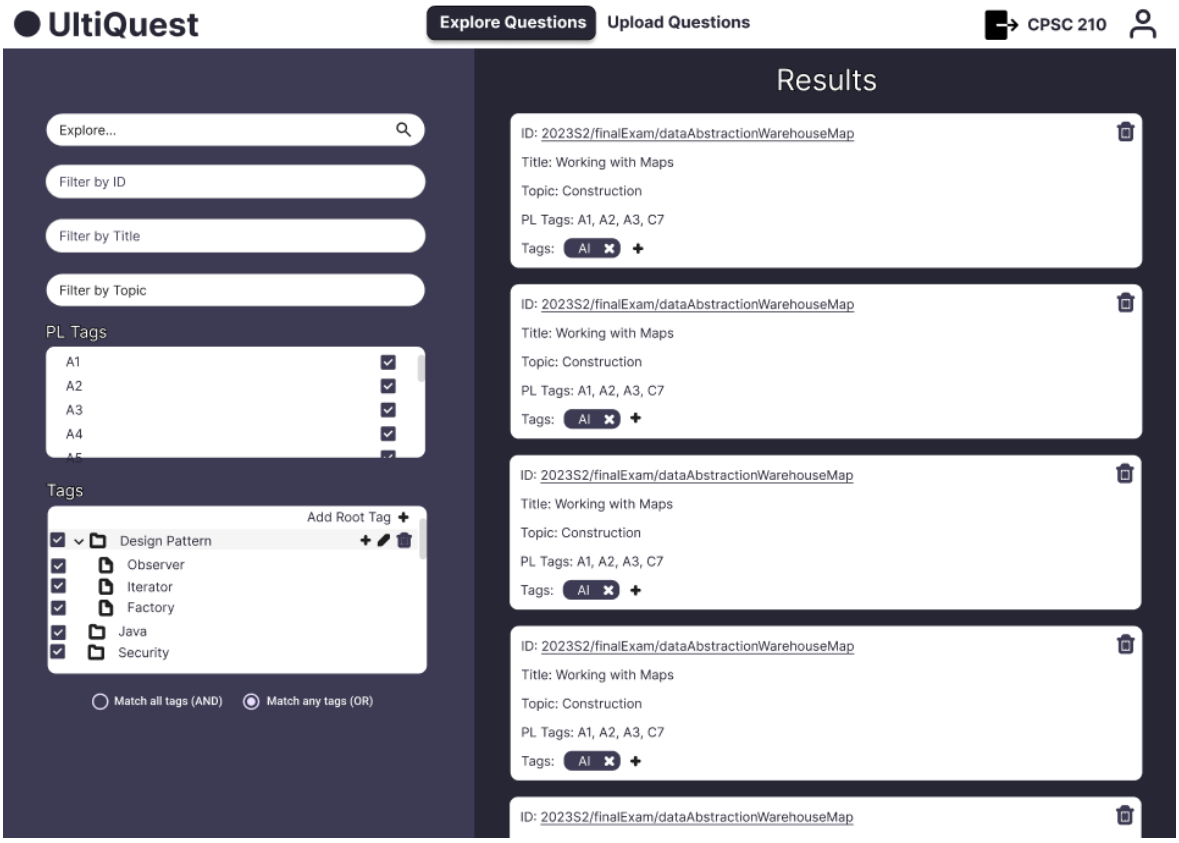


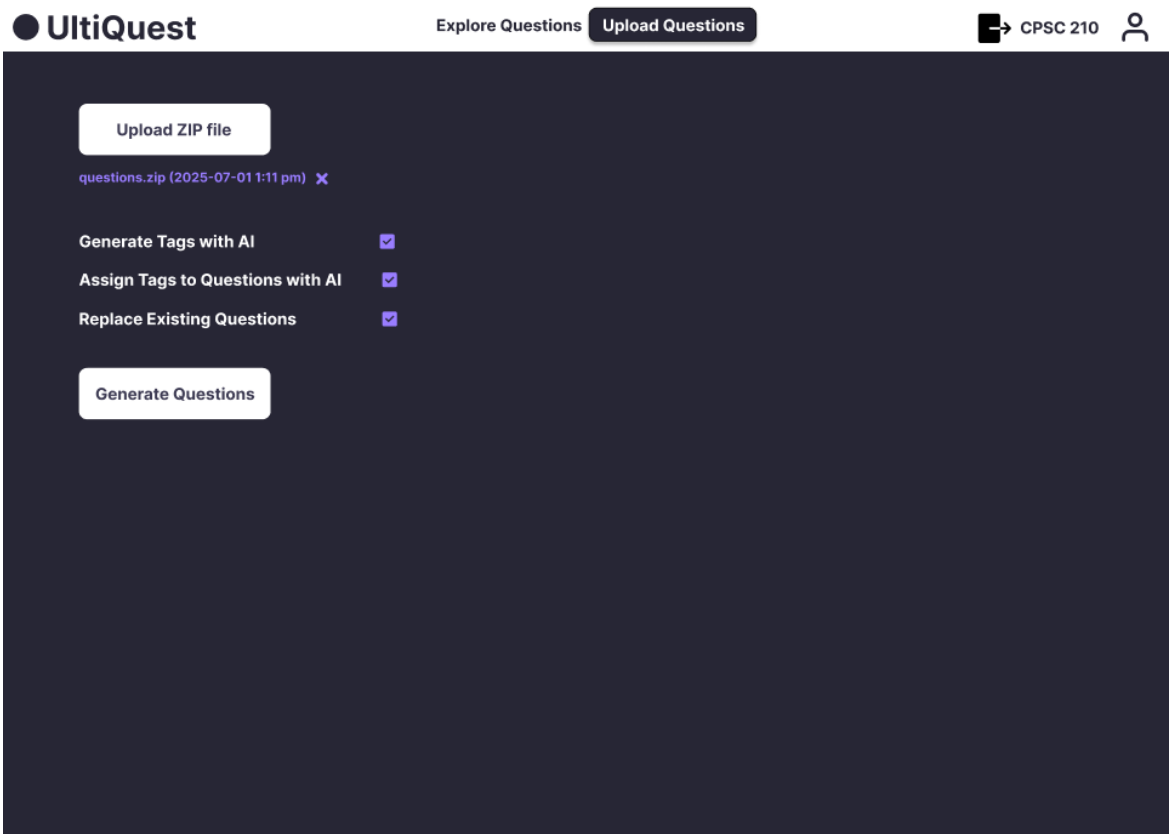
Section 2: What I Learned

UI/UX Design with Figma: I learned the fundamentals of UI design by creating mock-ups for the application in Figma. This involved designing interfaces for both the hierarchical tag structure and a conceptual knowledge graph view. While these were primarily for planning and conceptualization, the process taught me the basics of using Figma to visualize user interfaces and product features.

CPSC 448 – Report – Adam Mais

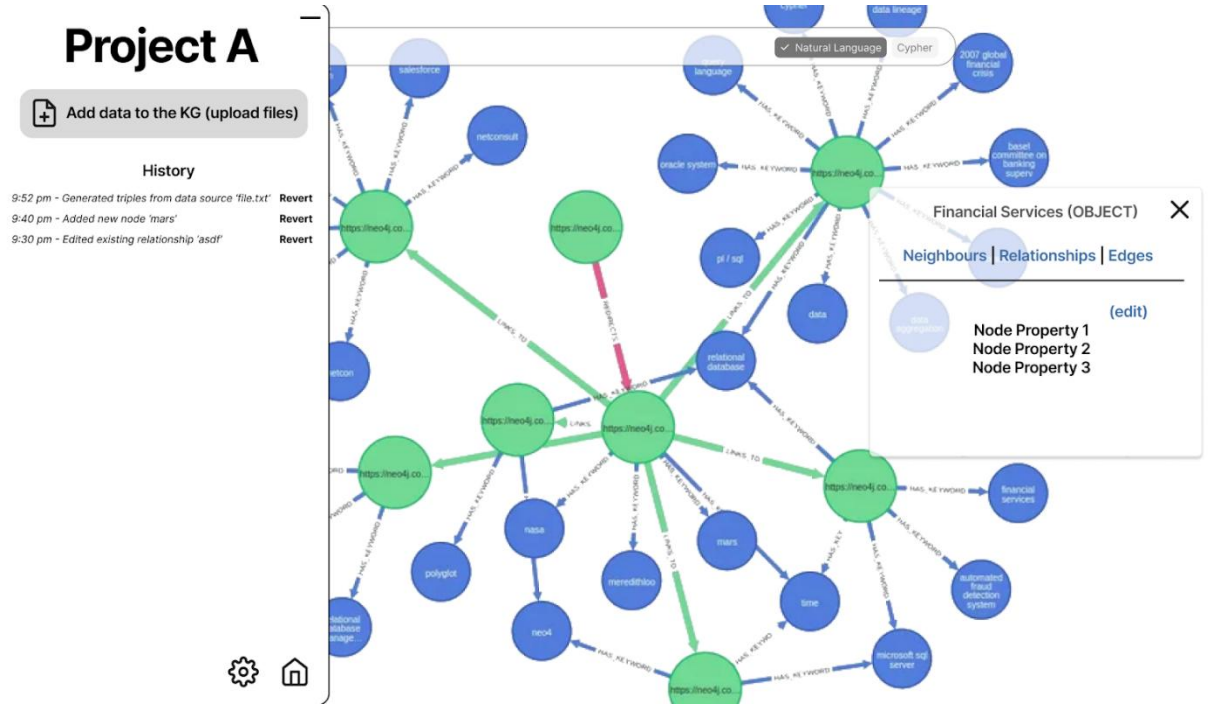
Figma design for a QBMS with tags represented in a tree structure:

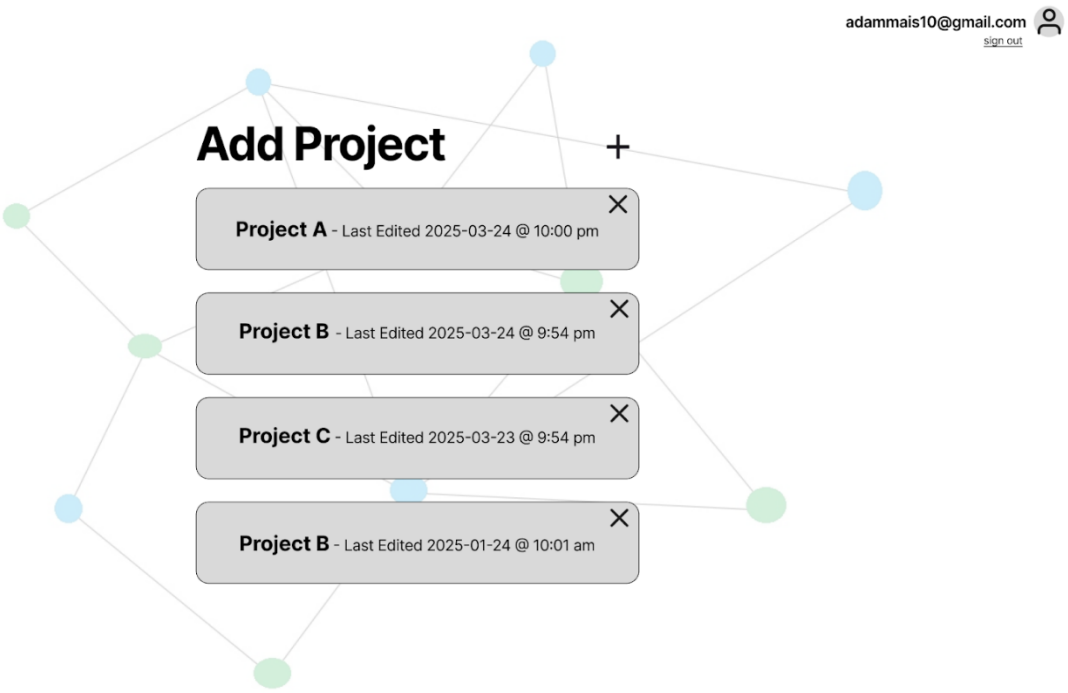




CPSC 448 – Report – Adam Mais

Figma design for a QBMS with tags represented in a knowledge graph:





CPSC 448 – Report – Adam Mais

AI-Assisted Development Workflow: I learned a structured workflow for collaborating with the AI coding assistant, Cursor.

1. Context Priming: I provided the LLM with project goals, documentation, and the codebase, then prompted it to ask clarifying questions to resolve ambiguities and establish a shared understanding.
    a. Example Prompt: *"Act as an expert developer in creating conversion tools between similar computer programming languages. You will help me build a way to convert html documents that contain custom elements into their vanilla html equivalent. The originals will be questions as rendered by PrairieLearn (please see @PrairieLearn for a full reference of the potential customs tags). As an example in @original.html we have a PrairieLearn question about Java class declarations, which when rendered in a browser looks like what you see in @rendered.html. Our task is to create a way that we can convert questions into their counterparts automatically. The questions are uploaded as a zip file by a user in my webapp. You will find the code for the webapp in @/qbms. The code that contains functionality for uploading the zip file of questions is in @ZipUploader.jsx. Currently, the zip uploader finds the associated info.json file for each question and extracts the relavent meta-data - it does not extract the question.html containing the content of the question. I want the zip uploader to extract the content of the question.html files and store them in a database entry for that question. For reference zip uploader currently calls an endpoint 'http://localhost:3001/questions/upload' which is created in the backend part of the application in @question.js. Before we write code, we will need to form a plan of how we will tackle this task. What questions do you have to which the answers I provide will give you the best possible chance of success in this task?"*
2. Collaborative Planning: We then jointly created a high-level plan that was broken down into a granular, markdown-based task list to serve as a clear roadmap.
3. Iterative and Supervised Execution: The AI completed one task at a time, after which I would pause to review, test, and verify the changes before proceeding. This tight feedback loop ensured quality and control.
4. Context Handoff: As an LLM nears its context limit, answer quality declines sharply, so it's best to switch models before the drop-off. At the bottom of each chat, Cursor shows how much context has been used up in the form of a percentage. You should ask the old LLM to generate a prompt to provide the new LLM with context of your task.
    a. Example Prompt: *"Now act as an expert prompt engineer. Help me write a prompt for another language model which will be helping me to complete the tasks in @conversion-plan-tasks.md. The other LLM will have ZERO context of this project so we'll need to describe what we're doing, what we've already established, what files are useful to look at, what questions the LLM should ask me to help glean its own context etc."*

Web Scraping and Automation: I refined my web scraping skills with Selenium. A key technical learning was how to bypass programmatic authentication by instructing Selenium to use an existing browser profile that already had an active, authenticated session. This works because the browser instance

launched by Selenium inherits all the cookies and session data from the specified profile, making the website believe it is being accessed by a logged-in user.

Full-Stack Development (MERN Stack): I gained extensive, hands-on experience in building a complete web application. This involved not only mastering the individual components of the MERN stack but understanding how they integrate. I designed and implemented a RESTful API with Node.js and Express, managed complex application state and lifecycle events in a React single-page application, and modeled and interacted with data in a NoSQL database (MongoDB).

Complex System Architecture & Interoperability: The most valuable lesson came from designing the hybrid Node.js/Python rendering engine. This task taught me how to architect a solution where two disparate technology stacks collaborate to perform a single, complex task. It was a practical exercise in interoperability, requiring the use of child processes for execution and the definition of a strict, JSON-based data contract for communication over stdio. This pattern is highly relevant to building modern microservices-based applications, where different services written in different languages must communicate reliably.

API Design for Bulk Operations: I researched and came to understand the trade-offs in designing APIs for bulk data processing. I analyzed the two approaches: a single endpoint that accepts a batch of items versus an endpoint that accepts one item at a time, called multiple times. While the latter offers granular progress feedback, it is inefficient due to the overhead of repeated HTTP requests. Through this analysis, I learned that the industry standard solution is the asynchronous job pattern. In this model, the client makes a single request to a job-creation endpoint, which immediately returns a jobId. The server performs the long-running task in the background. The client then polls a separate status endpoint using the jobId to get progress updates. This provides the best of both worlds: the network efficiency of a single request and the responsive user experience of granular feedback.