Project Name: Automating Database Question Generation and Marking with PrairieLearn

Design Documents

Version:
Final Version - June 8th, 2023

Contacts:
Matthew Obirek (matthewobirek@gmail.com),
Skyler Alderson (skyler@thealdersons.org),
Andrei Zipis (Andrei_Zipis@hotmail.com),
Nishant Srinivasan (nishant.srinivasan236@gmail.com)

Table of Contents

1. Project Description

The project - “Automating Database Question Generation and Marking with PrairieLearn", is to successfully automate the delivery and evaluation for relational algebra and SQL questions on PrairieLearn. The purpose is to improve the delivery of COSC 304 for students at UBC-O for our client, Dr. Lawrence.

As of right now, students in COSC 304 are reliant on using multiple different platforms for their coursework - including time-sensitive evaluations such as midterms and final examinations. These tools include Canvas, PrairieLearn, RelaX, GitHub, and a DBMS software to run and verify their queries locally.

Our objective is to integrate questions from specific, existing labs for COSC 304 from Canvas and GitHub into PrairieLearn, integrate the necessary tools for these labs such as RelaX, and automate randomized question generation as well as their evaluation process. In addition, the project requires us to implement a feature that allows students to verify their answers on PrairieLearn so that they need not rely on the DBMS software.

2. User Groups and Personas

2.1. Students

Figure 2.1: *A persona for a student user profile named Sally Student.*

2.2. Professor

Figure 2.2: *A persona for a professor user profile named Travis Teacher.*

3. Use Cases

These use cases build upon existing PrairieLearn documentation and only include items modified by this project. They should not be taken as a complete set of design documents.

3.1. Student Use Cases Diagram

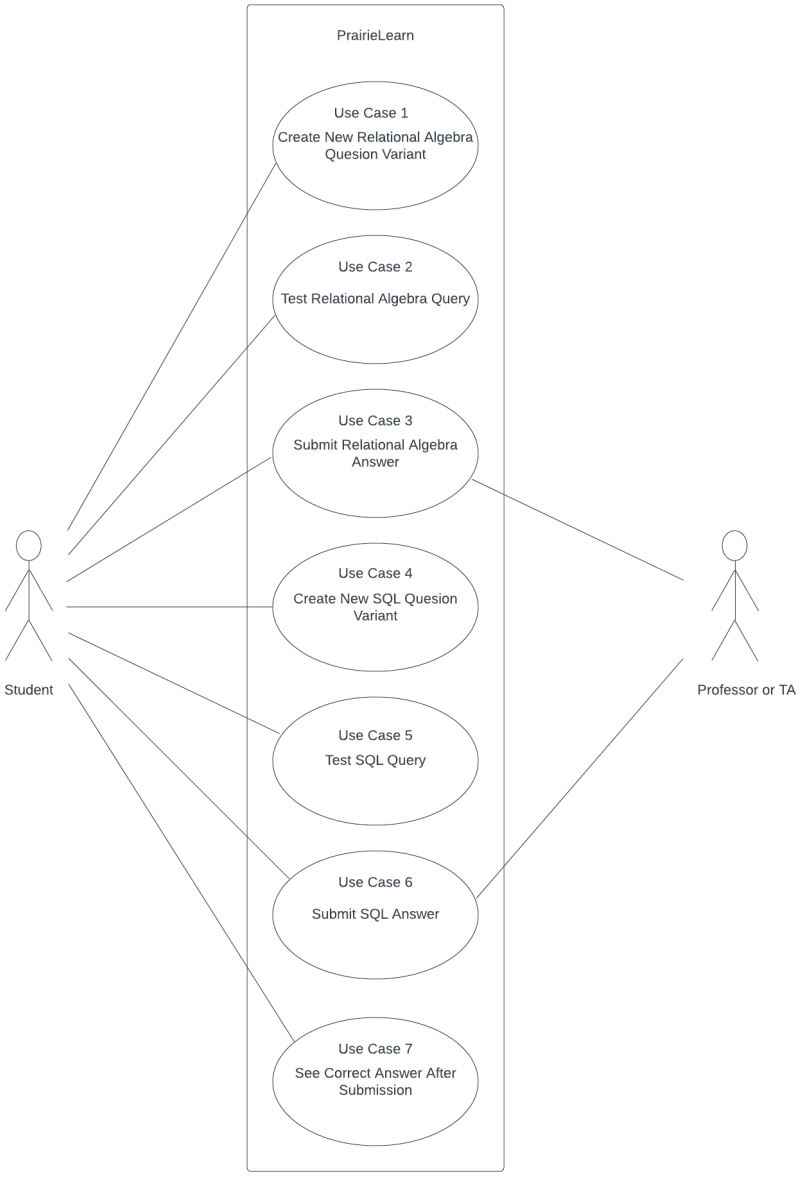

Figure 3.1: *A student may interact with the system (PrairieLearn) either through a Relational Algebra Lab or Answer SQL Lab. In either lab, the student may generate new question variants, test queries, and submit an answer. The professor or TA are able to see the student's grades after they have submitted their answer. After the final submission, a student may see the correct answer depending on the question's answer visibility settings.*

3.2. Use Case 1: Create New Relational Algebra Variant

ID: 1
Primary actor: Student
Description: A student wishes to obtain a new question variant
Precondition: Student has successfully logged in to PrairieLearn and selected a relational algebra question.
Postcondition: If the student has successfully accessed the question and generates a new question variant, the student sees a new variant of the relational algebra question.

Main Scenario:

1. Student selects the relational algebra question.
2. Student selects “Generate New Variant”.

Extensions:

None

3.3. Use Case 2: Test Relational Algebra Query

ID: 2
Primary actor: Student
Description: A student wishes to receive feedback on their answer prior to submission.
Precondition: Student has successfully logged in to PrairieLearn and selected the relational algebra question.
Postcondition: If the student successfully accessed the lab and formats their question, the student will see the results of their query.

Main Scenario:

1. Student selects the relational algebra question.
2. Student enters their answer in the RelaX editor.
3. Students tests their query without submission.
4. The output of the query is displayed for the student.

Extensions:

3a. The student’s answer includes one or more formatting errors.
    3a1. System issues an error message, informing the student of the error and prevents the query from being run.

3.4. Use Case 3: Submit Relational Algebra Answer

ID: 3
Primary actor: Student
Description: A student wishes to submit their answer for grading.

Precondition: Student has successfully logged in to PrairieLearn and selected the relational algebra question.
Postcondition: If the student successfully accessed the lab and formats their question, the student will receive feedback on their answer.

Main Scenario:

1. Student selects the relational algebra question.
2. Student enters their answer in the RelaX editor.
3. Student submits their answer.
4. Student receives feedback on their answer.

Extensions:

3a. The student's answer includes one or more formatting errors.
    3a1. System issues an error message, informing the student of the error and prevents the query from being run.
3b. The student's answer is incorrect and the student has remaining attempts.
    3b1. The student is informed their answer is incorrect and is prompted to try again.
3c. The student's answer is incorrect and the student has no remaining attempts.
    3c1. The student is informed their answer is incorrect.

3.5. Use Case 4: Create New SQL Variant

ID: 4
Primary actor: Student
Description: A student wishes to obtain a new question variant
Precondition: Student has successfully logged in to PrairieLearn and selected an SQL question.
Postcondition: If the student has successfully accessed the question and generates a new question variant, the student sees a new variant of the SQL question.

Main Scenario:

1. Student selects the SQL question.
2. Student selects "Generate New Variant".

Extensions:

None

3.6. Use Case 5: Test SQL Query

ID: 5
Primary actor: Student
Description: A student wishes to receive feedback on their answer prior to submission.

Precondition: Student has successfully logged in to PrairieLearn and selected the SQL question.
Postcondition: If the student successfully accessed the lab and formats their question, the student will see the results of their query.

Main Scenario:

1. Student selects the SQL question.
2. Student enters their answer in the SQL editor.
3. Students tests their query without submission.
4. The output of the query is displayed for the student.

Extensions:

3a. The student's answer includes one or more formatting errors.
    3a1. System issues an error message, informing the student of the error and prevents the query from being run.

3.7. Use Case 6: Submit SQL Answer

ID: 6
Primary actor: Student
Description: A student wishes to submit their answer for grading.
Precondition: Student has successfully logged in to PrairieLearn and selected the SQL question.
Postcondition: If the student successfully accessed the lab and formats their question, the student will receive feedback on their answer.

Main Scenario:

1. Student selects the SQL question.
2. Student enters their answer in the SQL editor.
3. Student submits their answer.
4. Student receives feedback on their answer.

Extensions:

3a. The student's answer includes one or more formatting errors.
    3a1. System issues an error message, informing the student of the error and prevents the query from being run.
3b. The student's answer is incorrect and the student has remaining attempts.
    3b1. The student is informed their answer is incorrect and is prompted to try again.
3c. The student's answer is incorrect and the student has no remaining attempts.
    3c1. The student is informed their answer is incorrect.

3.8. Use Case 7: See Correct Answer After Submission

ID 7:
Primary actor: Student
Description: A student wishes to complete their question to see if it is correct.
Precondition: Student has successfully logged in to PrairieLearn and selected the desired question.
Post condition: If the student successfully accessed the lab, submitted the question, and the professor has set the question's solution to be visible, then the question will be marked and the student will see the correct answer.

Main Scenario:

1. Student selects the desired question.
2. Student enters their answer.
3. Student submits their answer.
4. Student receives feedback on their answer and is able to see the correct solution.

Extensions:

3a. The student's answer includes one or more formatting errors.
    3a1. System issues an error message, informing the student of the error and prevents the query from being run.
3b. The student's answer is incorrect and the student has remaining attempts.
    3b1. The student is informed their answer is incorrect and is prompted to try again.
3c. The student's answer is incorrect and the student has no remaining attempts.
    3c1. The student is informed their answer is incorrect.

3.9. Professor or TA Use Case Diagram

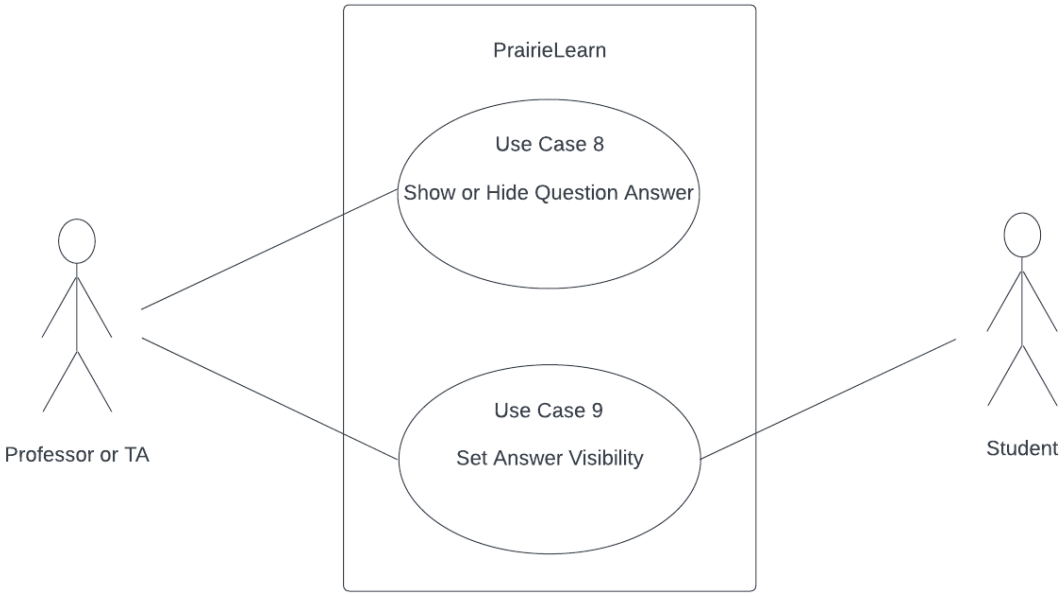

Figure 3.9: *A professor or TA is able to interact with the system (PrairieLearn) either by viewing a question's answer or by setting the visibility of a solution after the student submits an answer.*

3.10. Use Case 8: Show or Hide Answer

ID: 8
Primary actor: Professor
Description: A professor wishes to view the correct answer to a desired question.
Precondition: Professor has successfully logged into PrairieLearn and selected the desired question.
Post Condition: If the professor successfully accessed the lab, then the professor sees the question solution.

Main Scenario:

1. Professor selects the desired question.
2. Professor selects "Show/Hide Answer".

Extensions:

None

3.11. Use Case 9: Set Answer Visibility

ID: 9
Primary actor: Professor
Description: A professor wishes to change whether the answer is viewable after the question is answered by a student.
Precondition: Professor has successfully logged into PrairieLearn and selected the desired question.
Post Condition: If the professor has successfully logged into PrairieLearn, then the question visibility is set.

Main Scenario:

1. Professor selects the desired question.
2. Professor navigates to the “Files” tab.
3. Professor selects “info.json”.
4. Professor sets desired visibility.

Extensions:

None

4. System Architecture

4.1. PrairieLearn Question Lifecycle

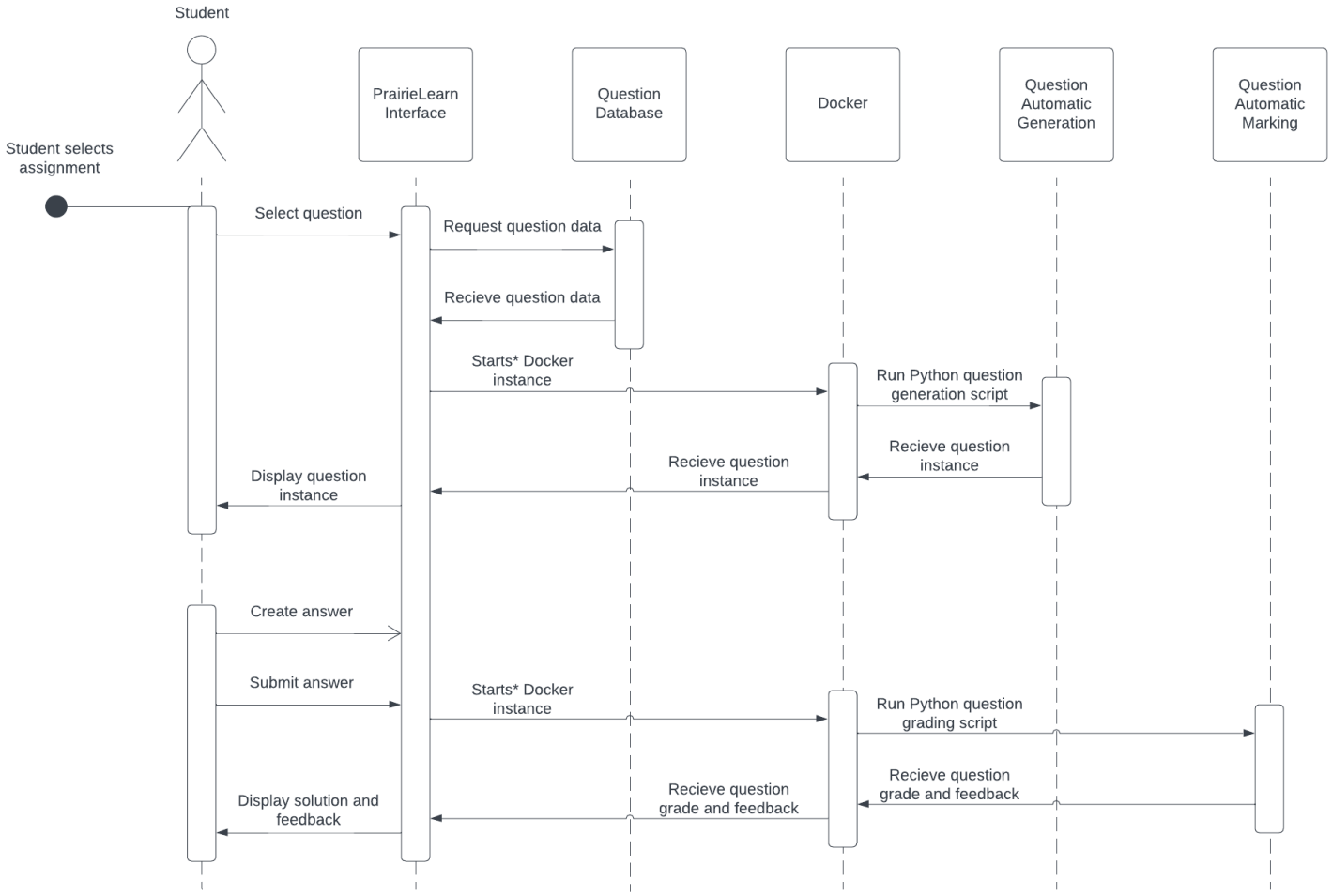

Figure 4.1.1: *After a student selects an assignment and a question, PrairieLearn retrieves the question's base data before using Docker to run Python to generate a question variant. After a student submits their answer, PrairieLearn uses a Docker instance to run Python to grade the question and give feedback to the student.*

4.2. Sequence Diagrams

4.2.1. Sequence Diagram for Use Cases 1 & 4

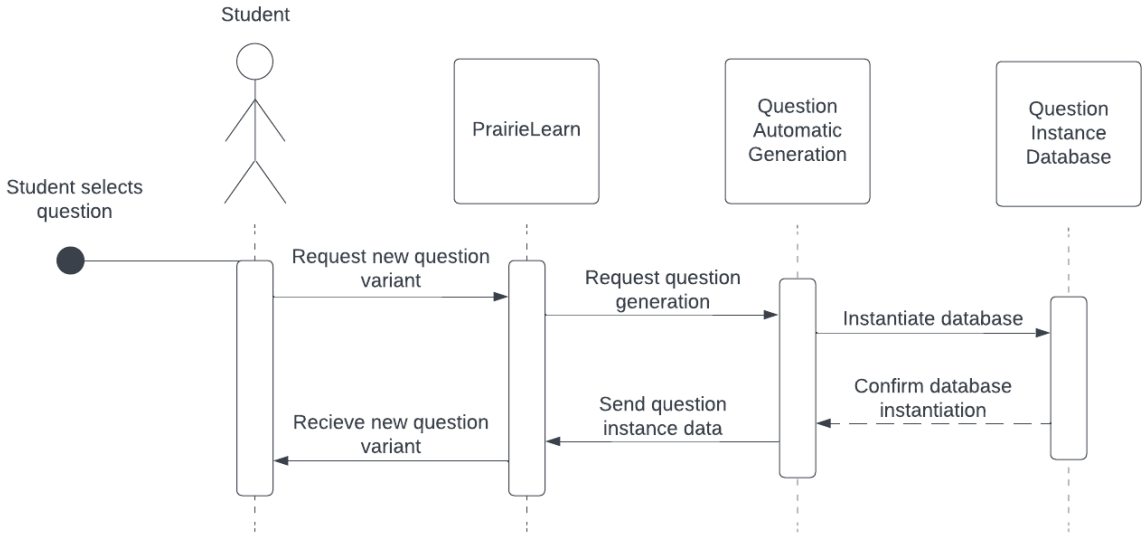

Figure 4.2.1: *After a student selects a question, they request a new question variant from the PrairieLearn system. PrairieLearn uses its automatic question generation to create a new question variant and in so doing instantiates a new database. The question instance data is returned to the student.*

4.2.2. Sequence Diagram for Use Cases 2 & 5

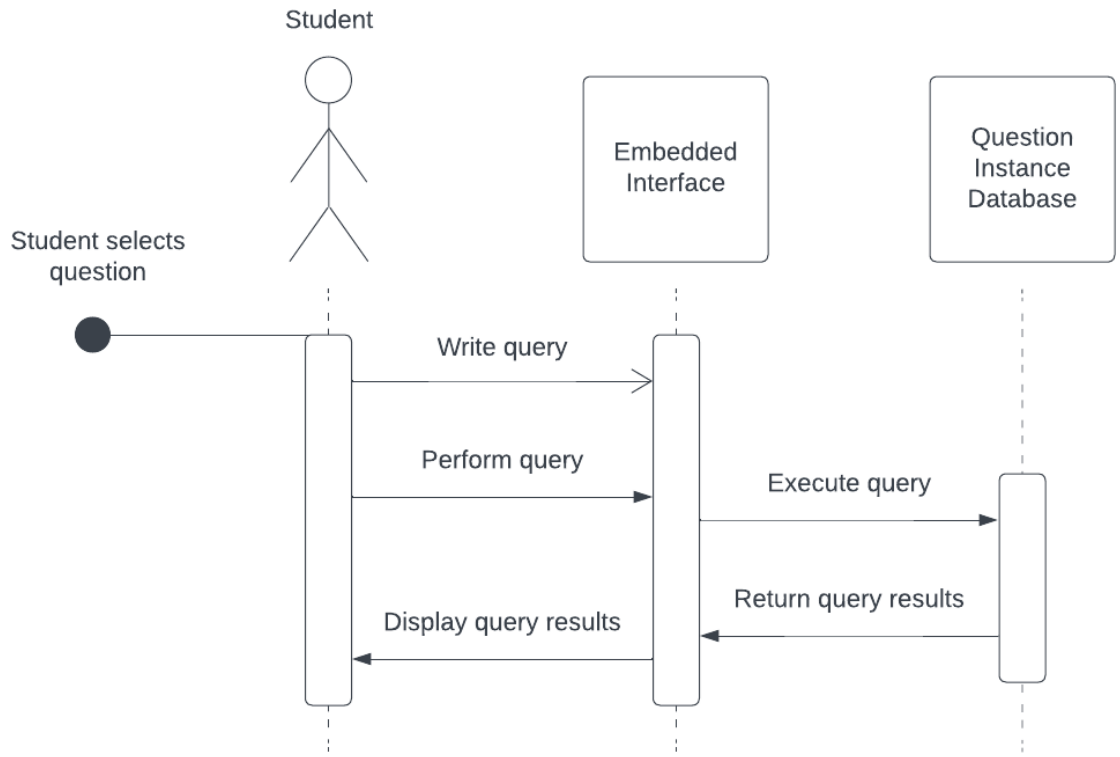

Figure 4.2.2: *After a student selects a question, they use the embedded editor to create a query. The student then tests the query, where PrairieLearn executes the query on a database instance. The results of the query are returned to the student.*

4.2.3. Sequence Diagram for Use Cases 3, 6, & 7

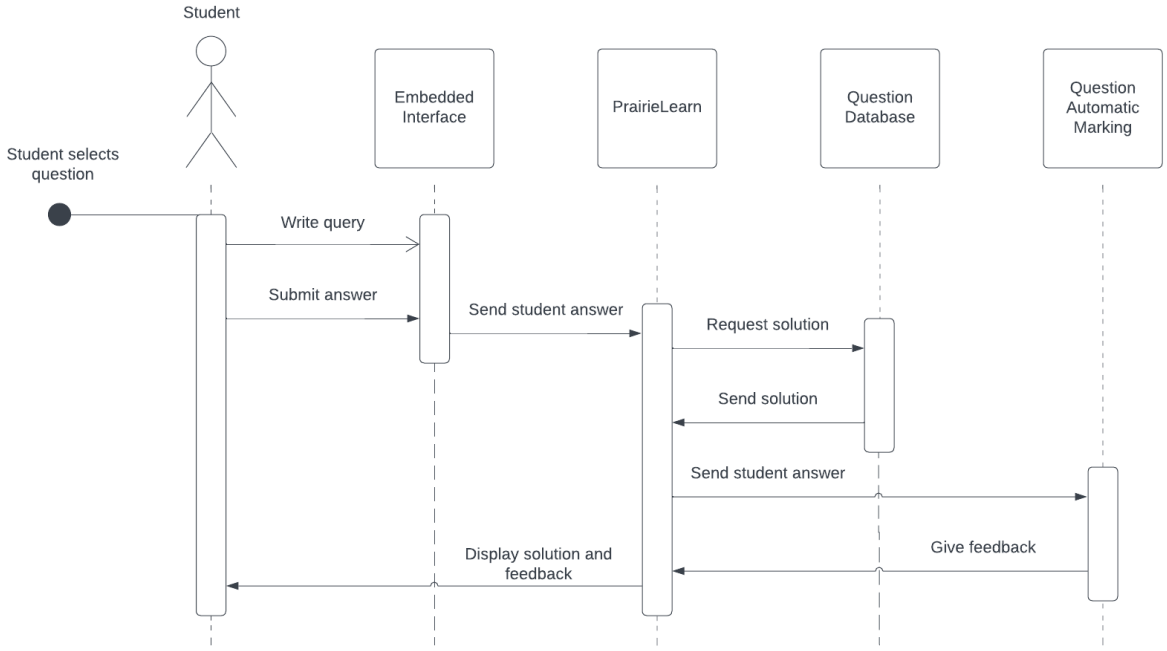

Figure 4.2.3: *After a student selects a question, they use the embedded editor to create a query. The student then submits their answer, where PrairieLearn obtains the solution from the questions database and sends both the solution and the student's answer to the automatic grader. The feedback of the student's answer is returned to the student.*

4.2.4. Sequence Diagram for Use Case 8

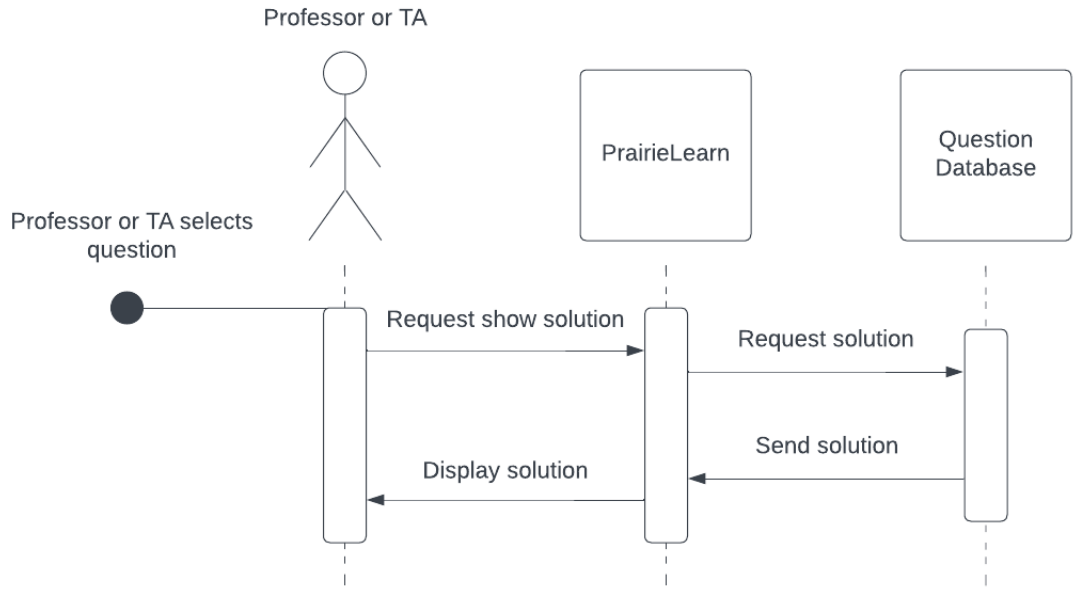

Figure 4.2.4: *After a professor or a TA selects a question, they request to see the answer. PrairieLearn then obtains the solution from the question database and shows it to the professor or TA.*

4.2.5. Sequence Diagram for Use Case 9

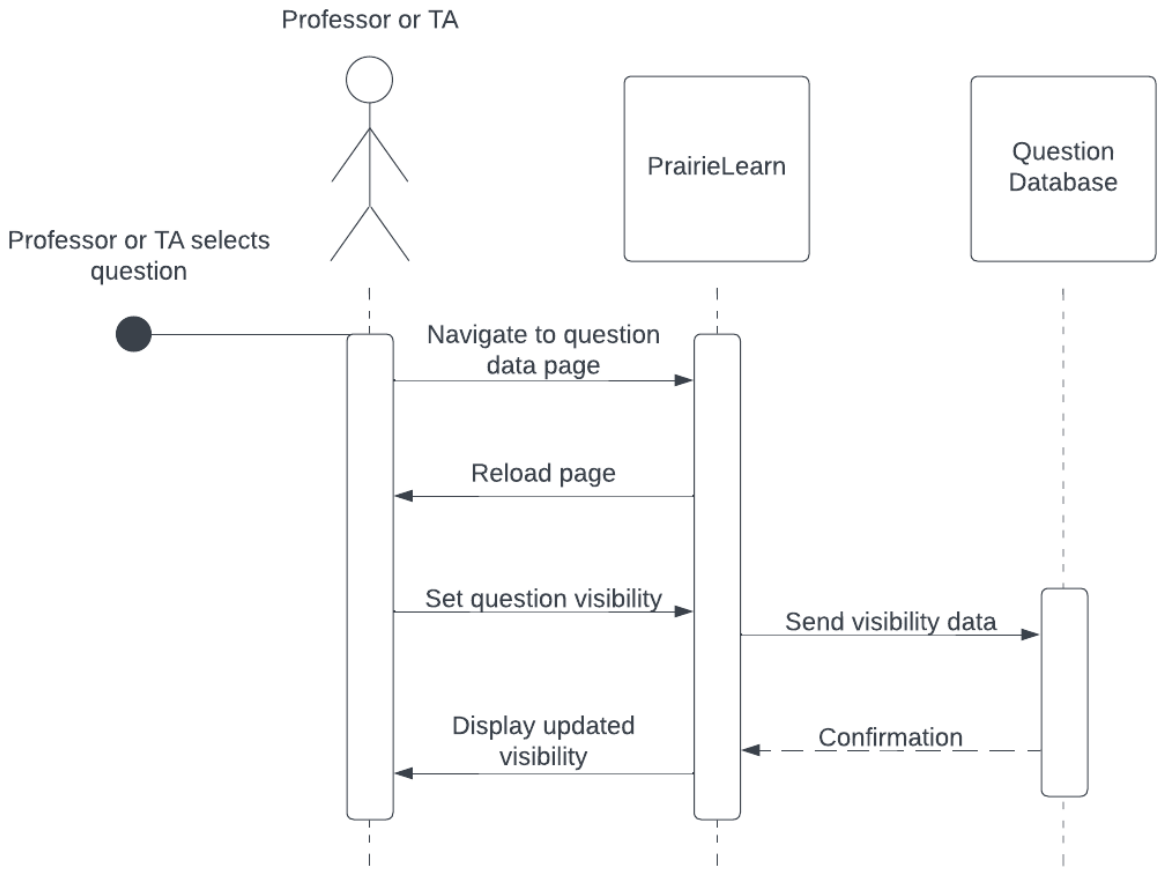

Figure 4.2.5: *After a professor or a TA selects a question, they navigate to the files tab of PrairieLearn and selects the question data. The professor or TA then updates the solution's visibility, where PrairieLearn changes the visibility settings and displays the updated status.*

4.3. Data flow Diagrams

4.3.1. Level 0 Data Flow Diagram

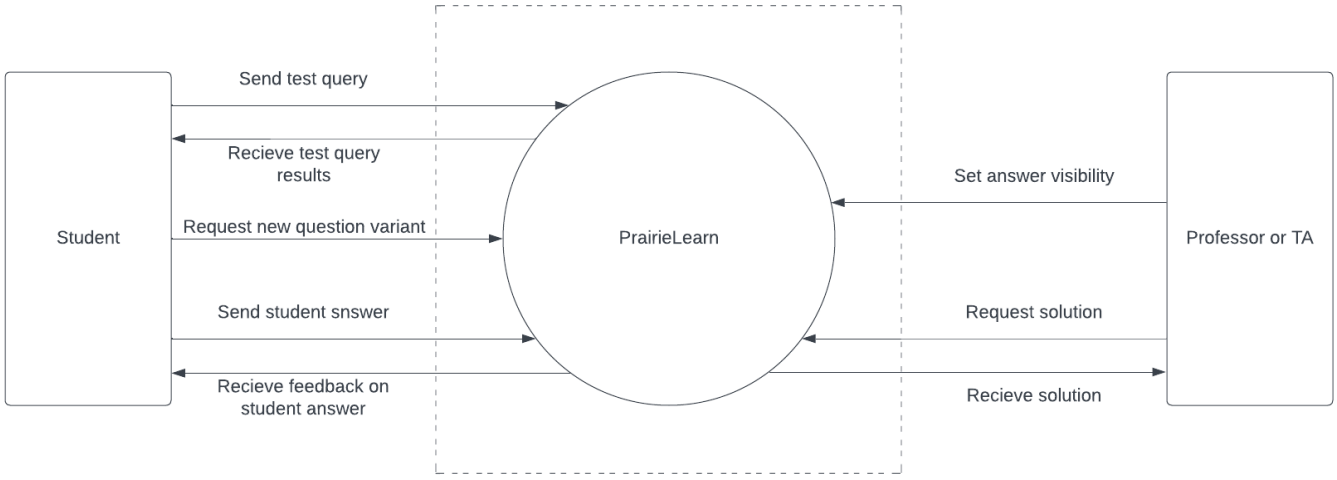

Figure 4.3.1: *The test query a student submits is sent to PrairieLearn, which returns the query's results. When a student submits their answer, PrairieLearn gives the student feedback. A student may request a new question variant. A professor or TA can show or hide the solution or set question visibility.*

4.3.2. Level 1 Data Flow Diagram

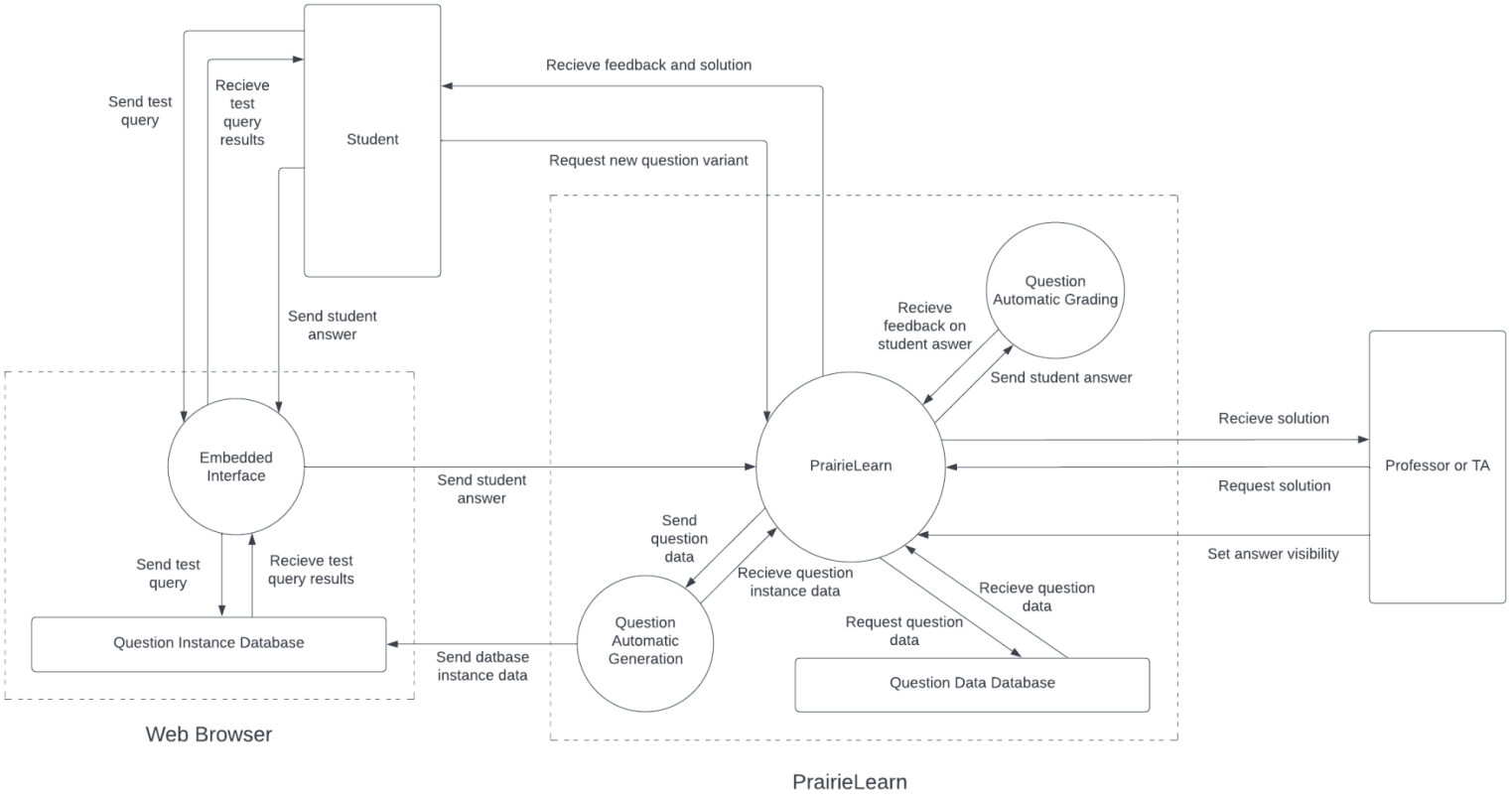

Figure 4.3.2: *This figure has the same information as figure 4.2.1 but has expanded upon the PrairieLearn system. The student sends their query to an embedded interface, which redirects it to a database instance; the database instance returns the query's results, which are displayed using the interface. A student submits their answer through the embedded interface and to PrairieLearn, where it is then sent to be automatically graded. The student's feedback is returned through PrairieLearn and then to the student.*

5. UI Mockups

5.1. RelaX Integration

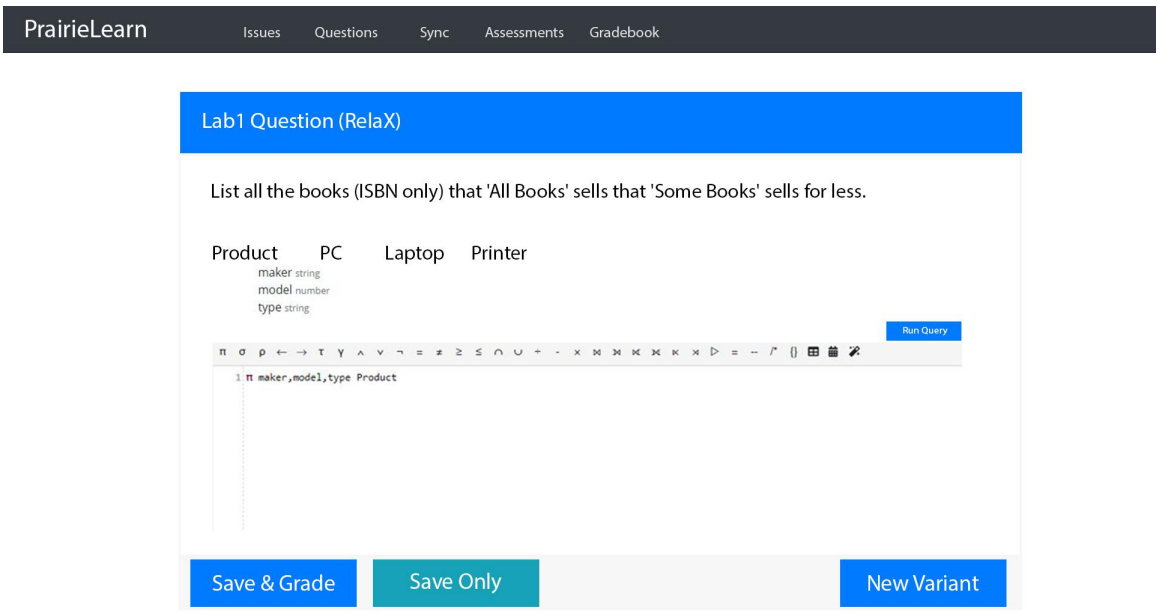

Figure 5.1.1: *A mock UI for the RelaX editor within PrairieLearn.*

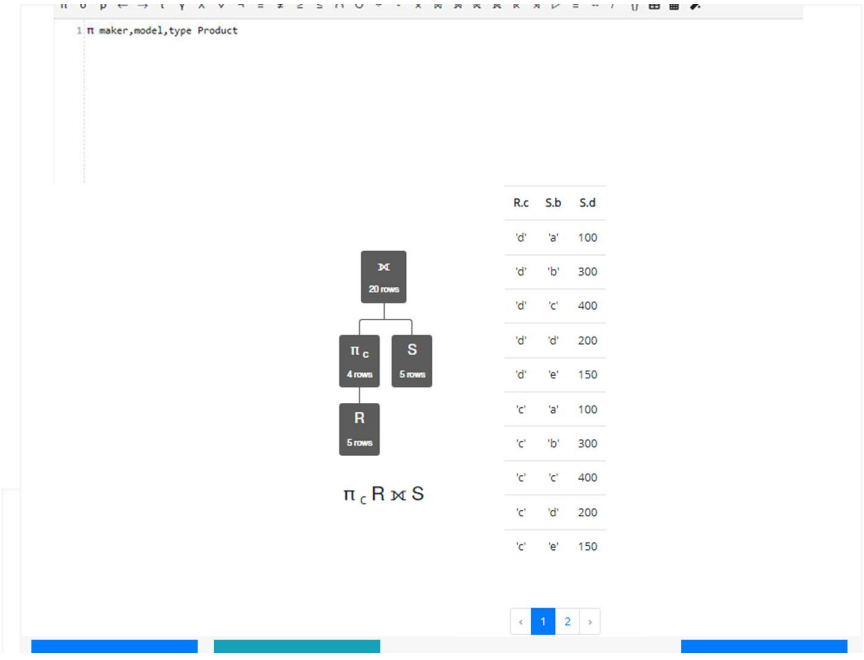

Figure 5.1.2: *A mock UI for the RelaX query results within PrairieLearn.*

5.2. SQL/DDL Integration

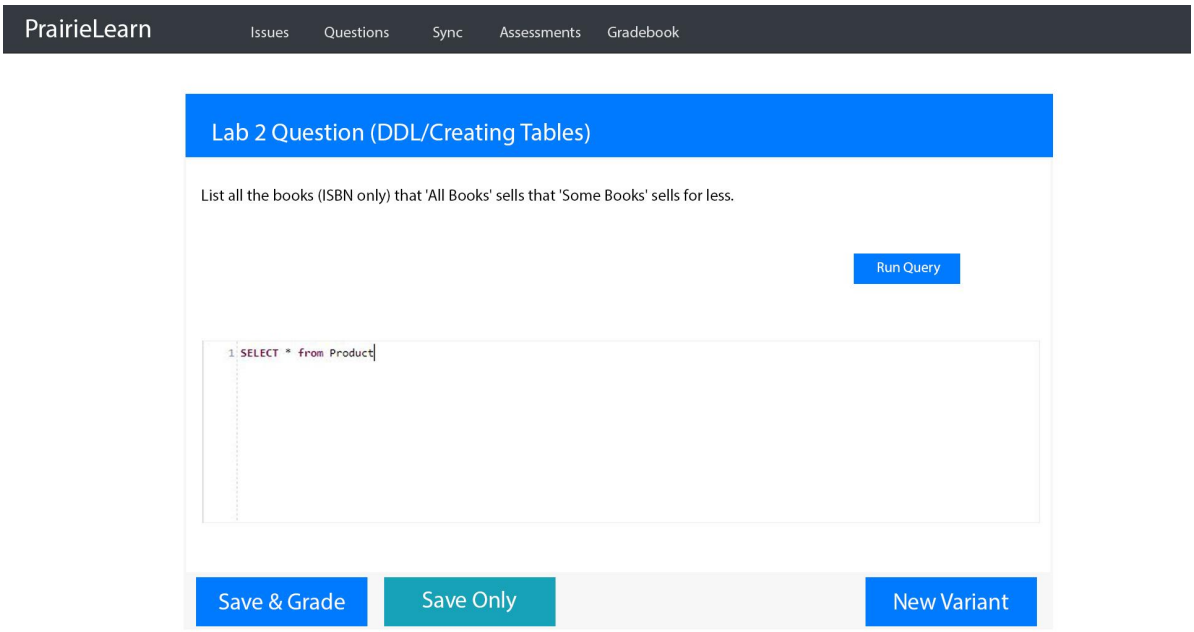

Figure 5.2.1: *A mock UI for the SQL editor within PrairieLearn.*

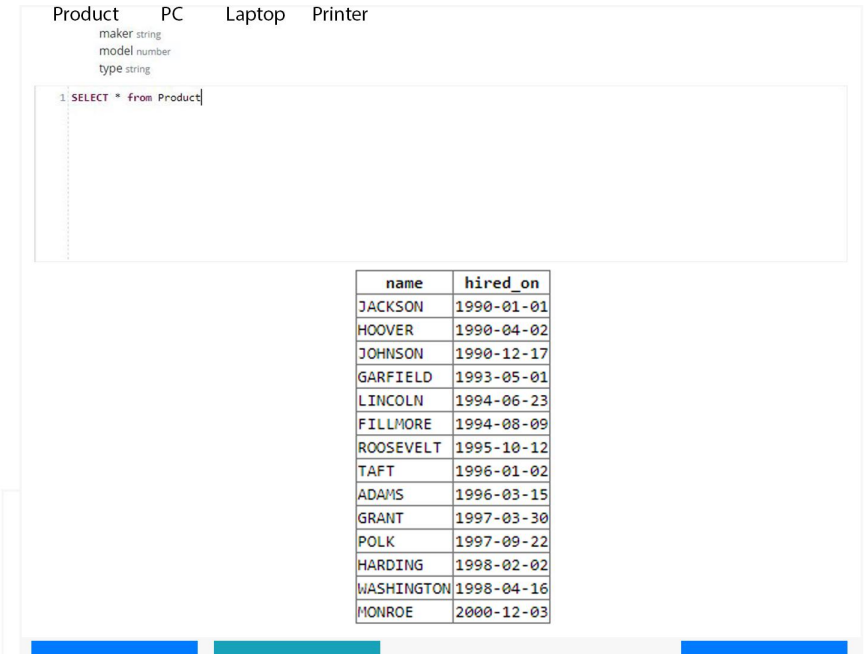

Figure 5.2.2: *A mock UI for the SQL query results within PrairieLearn.*

6. Technical Specifications

6.1. Frontend

Integration and development of front-end development will continue to be in JavaScript, HTML, Mustache, and CSS.

6.2. Backend

Back-end development will continue to be in Python. PrairieLearn is built with Node.js but will not be directly used in our system's development.

6.3. Database

The data for the labs (questions and solutions) will be stored using PostgreSQL. This is also handled through the PrairieLearn system and will not be directly used in this system's development.

6.4. Tools to Build Software

- IDE: Visual Studio Code
- Time Tracking: Clockify
- Task Management: GitHub Projects
- CI/CD: DroneCI

7. Testing

7.1. Overview

Testing is a crucial component of the software development process. It ensures that the system meets all of our specified requirements and functions as the client intends. In this design document, we explain the various types of testing that will be applied and the purpose that it serves. These include: unit testing, integration testing, performance testing, and functionality testing. As we are following a test-driven development structure, these tests will guide our development and reduce the amount of resources dedicated to bug fixes later in development.

7.2. Unit Testing (Structural)

Unit testing is a white-box testing technique. It involves designing tests for each function to ensure that it produces the expected output. At minimum one unit test will be designed prior to a function being created with additional unit tests designed during development. The goal is to achieve coverage testing to a degree to ensure that the majority of our code is tested. However, irrelevant tests will not be designed solely for the purpose of increasing our coverage. An example of a unit test would be an empty

submission correctly being given a score of 0 ensuring that the scoring function works even without input. These tests will be written with Python Unittest Framework inside python files and be run with DroneCI.

7.3. Integration Testing

Integration testing is a black-box testing technique. This verifies whether multiple components being used together are working appropriately and producing the correct result. It examines the interaction between subsystems and identifies any issues that may arise from these interactions. Integration tests will be written to guide how features will be designed as a whole. An example of an integration test is appropriately handling user submitted strings in a textbox and ensuring that SQL code is then querying the database appropriately. These tests will be written with Python Unittest Framework inside python files and be run with DroneCI.

7.4. Performance Testing

Performance testing is essential to assess the system's ability to scale appropriately. One of the requirements of the system is to function concurrently between hundreds of students and a multitude of classes. As a result, the system will need to be stress tested to determine whether performance dips below acceptable values. We will progressively increase the submission load on the system and measure the response time. In doing so, we may be able to determine bottlenecks and optimize resource allocation. This will be tested on UBC-O's proper PrairieLearn server by logging the time it takes for questions to load, and for submitted answers to be autograded. We will simulate this under load to mimic our non-functional requirement of approximately 200 concurrent students.

7.5. UI Testing

UI testing allows us to ensure that various components of the software's UI, both new and old, are working as required. In this project some of the new UI components will be the button for running queries without submitting, displaying results of those queries in tables, an input field for Relational Algebra, and an input field for SQL/DDL. This will be tested with Selenium.

7.6. Software/Technologies

7.6.1. Python unittest

Python code for: the rendering of questions, automatic marking, and automatic question generation will be tested with python unittest framework. We will be using this for our Unit Tests, Integration Tests, and to automate our UI tests.

7.6.2. Selenium

To ensure that our UI components, both old and new are working as required, we will be using the Selenium WebDriver for our UI tests.

7.6.3. Drone CI

Our continuous integration will be completed through Drone CI. All tests will be run and passed before merging and again when integrating new features to the main branch

7.7. Continuous Integration/Deployment

Our continuous integration will be completed through DroneCI. All tests will be run and passed before merging and again when integrating new features to the main branch

Requirements	**Type of Testing**	**Status**
Functional		
System will allow for relational algebra statements to be entered.	UI Testing Integration Testing	Fail Fail
System will show visualizations of the resulting entered statement prior to submission.	UI Testing Integration Testing	Fail Fail
System will automatically mark the relational algebra questions once submitted.	Unit Testing	Fail
System will allow for DDL/SQL code to be entered. System will show resulting tables of queries prior to submission.	UI Testing Integration Testing	Fail Fail
System will automatically mark the DDL/SQL questions once submitted.	Unit Testing	Fail
Student will be able to see the correct answer if the professor has allowed for the	Unit Testing Integration Testing UI Testing	Fail Fail Fail

correct answer to be displayed after the question is submitted.		
Professor will be able to set whether the correct answer will be displayed after the question is submitted.	Unit Testing UI Testing	Fail Fail
Professor will be able to see the correct answer.	UI Testing	Fail
Non-Functional		
The system will support all COSC 304 users simultaneously – about 200 students.	Performance Testing	Fail
The system will ensure data integrity and preservation so that no data is lost upon submission.	Performance Testing	Fail
The system will display entered queries within 3 seconds at scale and under optimal conditions.	Performance Testing	Fail
The system will return automarked submissions within 5 seconds at scale and under optimal conditions.	Performance Testing	Fail
The user interface will match existing software used for COSC 304.	UI Testing	Fail
Technical Requirements		
Rebuild RelaX editor and calculator into PrairieLearn	UI Testing Integration Testing	Fail Fail

Frontend: JavaScript, HTML, CSS	UI Testing	Fail
Backend: Python, Node.JS	Unit Testing	Fail
Write JavaScript code that takes in SQL/DDL statements and displays appropriate table results	Integration Testing	Fail
Write Python code that automatically marks submitted data and returns the students grade	Unit Testing	Fail