Capstone Portfolio

Survey Analytics Dashboard for the Empower Student Center
ESOF 423 - Spring 2026

Chrys Nelson, Ari Osmun

ESOF 423 Group 5 Portfolio

Section 1: Program

When we were first looking for a project idea, our team's objective was to involve both of our minors, psychology and data science. That's when we visited the Empower Student Center for inspiration, and Susan Kelly, the Director of the Empower Program, had a task for us. At the end of the semester, the students who participate in the Empower Center are asked to fill out an optional survey that gathers information about identity and the effectiveness of the Center. Susan then plans to perform analytics on those responses. Our end goal was to simplify that process. The project is a web application where Susan can upload a CSV file of responses, and the app iterates through each question.

It could automatically detect the question type, and using the question responses, calculate statistics and generate a graph. Susan can then download the graph for use in her research for the Empower Center, saving her time and effort. Overall our goal was to create an application that was simple, effective, and user-friendly.

Deployed Project Link on Render
Github Link

Section 2: Teamwork

Team Member 1

Team Member One's contributions included working mostly on the architecture and backend of the project, as well as helping organize the group's Agile practices and organization with GitHub. Generally, they focused on the logic of the dashboard and how it processed questions, calculated statistics, and generated visualizations. They also focused on implementing modern programming solutions, for example, setting up automated testing, CI/CD pipelines, and ensuring that the project structure followed modern best practices.

Throughout the semester, there were various blockers and obstacles. Despite these, Team Member One maintained communication with the client and still completed most sprints on time, focusing on the robustness of the overall structure and project health with long-term applications for the client in mind. Over the course of this project, Team Member One worked about 26 hours outside of class on this project during sprints.

Team Member 2

Team Member Two's contributions included setting up the user interface interactions using a mix of Django and Flask, as well as certain backend interactions. They helped with the user interface and organization at the start of the semester, focusing on the general structure of the web application. They also gained new knowledge on GitHub features and automated testing to enhance the coding process overall.

Later in the semester, they used their knowledge of user interface design to form a more user-friendly web application that was easier to use and had clear instructions. They changed the UI of the question organization, as well as added filtering options for the user to find specific questions. Over the course of this project, Team Member Two worked about 23 hours outside of class on this project during sprints.

Section 3: Design Pattern

For our design pattern, we used multiple different patterns depending on the requirements of the problem. We used Python for the backend to handle uploading the CSV, automatically detecting the question type, and generating a dictionary with the question, question type, and responses. To generate some of the visuals, such as the question display, we used JavaScript elements to display and filter the data. On top of that, we used make to run locally, and pytest for automated testing on GitHub.

However a specific design pattern we used is the Strategy pattern, implemented in `backend/main.py` in the `auto_detect_type()` function (lines 18-74).

The Strategy pattern defines a family of algorithms, encapsulates each one, and selects between them at runtime based on context. In our case, the "algorithms" are the different detection and classification strategies for each question type. Rather than one tangled block of logic, `auto_detect_type()` executes two ordered strategies in sequence:

- **Keyword matching on the question text** - the function checks the question string against our keyword lists for likert, ranking, and multiple choice types.
- **Response value analysis** - if keyword matching doesn't work, the function falls back to analyzing the actual response values, checking for yes/no sets, numeric ranges, and repeated string patterns.

Each question type (`likert, ranking, multiple_choice, yes_no, short_answer`) has its own detection logic, and the correct one is selected at runtime based on whatever the CSV column looks like.

We used this pattern rather than coding directly because a single flat if/else block would become increasingly brittle as new question types are added. With the Strategy approach, adding support for a new type only requires adding a new keyword list or response check without touching the existing logic. It also made the code significantly easier to test, since each detection branch can be verified independently, which is reflected in our automated testing. Frankly we thought the process of autodetecting questions was going to be really difficult at first, but it ended up being a lot more simple this way.

```
      ef auto_detect_type(question_text, responses):
         """
         Attempts to infer the question type from its text and response
values.
         Returns one of: 'likert', 'ranking', 'multiple_choice', 'yes_no',
                         'rating', 'short_answer', or None if indeterminate.

         Strategy:
           1. Keyword matching on the question text (most reliable for
Qualtrics exports).
           2. Response value analysis as a fallback.
         """
         text = question_text.lower()

         # --- Text-based detection ---
         likert_keywords = [
             "strongly disagree",
             "strongly agree",
             "scale: 1-5",
             "1-strongly",
             "1 = strongly",
             "very dissatisfied",
             "very satisfied",
             "level of agreement",
         ]
         ranking_keywords = [
             "rank",
             "order of importance",
             "most important",
             "least important",
             "drag and drop",
         ]
         multichoice_keywords = ["select all that apply"]

         if any(k in text for k in likert_keywords):
             return "likert"
         if any(k in text for k in ranking_keywords):
             return "ranking"
         if any(k in text for k in multichoice_keywords):
             return "multiple_choice"
```

```
        # --- Response value analysis ---
        non_empty = [r.strip() for r in responses if r.strip()]
        if not non_empty:
            return None

        unique_lower = set(v.lower() for v in non_empty)

        # Yes / No
        if unique_lower <= {"yes", "no"}:
            return "yes_no"

        # All numeric
        try:
            nums = [float(r) for r in non_empty]
            # Check if every value is a whole number
            if all(n == int(n) for n in nums):
                int_set = set(int(n) for n in nums)
                if int_set <= {1, 2, 3, 4, 5}:
                    return "likert"
                if int_set <= set(range(1, 8)):
                    return "ranking"
            return "likert"
        except ValueError:
            pass

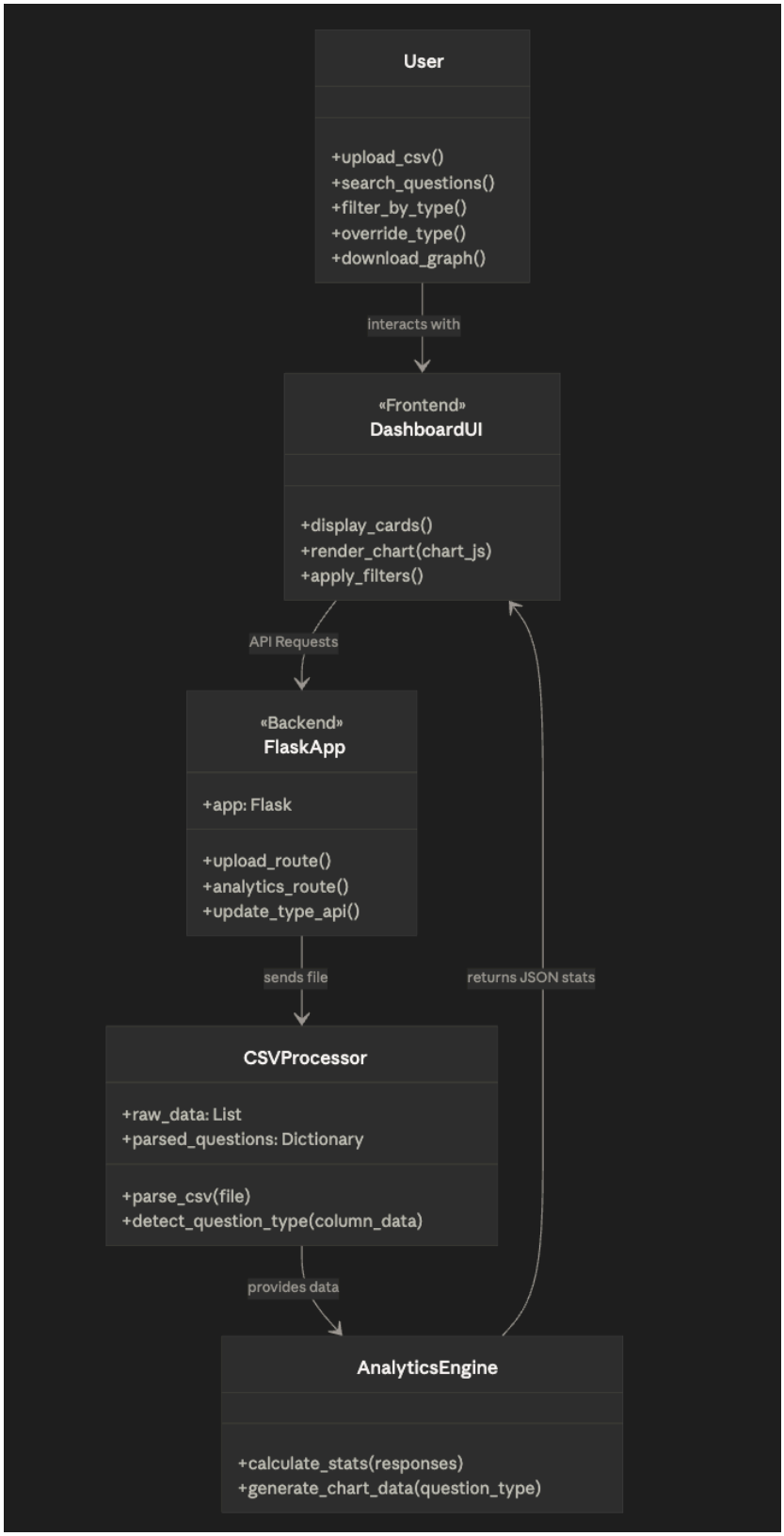
        # Small fixed set of repeated strings → multiple choice
        if len(unique_lower) <= 8 and len(non_empty) > len(unique_lower):
            return "multiple_choice"

        # Default: free-text
        return "short_answer"
```

Section 4: Technical Writing

User Documentation and Developer Documentation can be found at the end of the document.

Section 5: UML



Section 6: Design Trade-offs

One design trade-off we faced was picking an appropriate backend tech stack. Originally our idea was to host and store question and response data on a hosted database, but eventually we realized that it wasn't necessary for our design flow, and would add unnecessary steps to the process. Instead we focused on making an app that was easy to use and ran in real-time. One of the downsides to not implementing a database to store our data was that our client had to upload the CSV each time she intended to use our application, as it was unable to save the data on the cloud. However, choosing not to add a database saved us from complications in our design system, as well as avoided any future issues that our client might have run into with a database.

Another design trade-off was our hosting solution. We looked at a couple different options for hosting our website, but eventually we stuck with hosting on a free tier of Render. This allowed us to keep our application cost-effective by saving money on a hosting service. Unfortunately, it came with one big trade-off. When initially loading our web application, it often takes 10 to 15 seconds for the server to start up. Once the project is up, the delay is minimal, and our application runs efficiently; it only has issues with the initial bootup. We felt like this was fine for our project, as our client wouldn't be needing to load the server frequently, and because the application ran efficiently once it was loaded, we figured it was worth the trade-off.

Section 7: Software Development Life Cycle Model

For our software development model, we used Agile with the Scrum methodology. Each day, we started with a daily standup meeting that lasted about 3 to 5 minutes. Here, we would discuss what progress we have made on the project, and what we plan to work on over the next sprint. Our design process was split into six two-week-long sprints, with a feature-complete design at Sprint 4. This helped us break up our semester-long project into six chunks, giving us a clearer idea of what we needed to accomplish by the end of the sprint.

We found the structure really helpful for the design process, because it allowed us to organize all the work needed to accomplish different sections. When looking at a project as large as this, it's easy to get overwhelmed, but the Agile process helped prevent that feeling. We also liked how the Work Unit system worked, allowing us to break different features into work units to estimate how long they would take. Although our estimations weren't always accurate, it gave us a rough idea of how long we would be working on a specific feature.

A slight downside with this model is the assumption that each team member is able to do a consistent amount of work each sprint. We could see how this could work better in a professional environment, but as college students, we often had exams or busy weeks that slowed down the design process which we had to catch up on later, although having a burn-down chart helped with this, to keep track of which sprints we were ahead on and which sprints we were behind in. It was a new process for us, but we were eventually able to adjust to the model, and we found it incredibly beneficial.

User Documentation

Analytics Dashboard for the Empower Student Center

Chrys Nelson, Ari Osmun

A web application that transforms Qualtrics survey CSV exports into interactive visualizations and statistics - reducing hours of manual Excel work to minutes.

How to Use

Step 1: Export from Qualtrics

1. Log into Qualtrics
2. Open your survey and go to the Data and Analytics tab.
3. Click Export & Import, then Export Data.
4. Select CSV format and download the file

Step 2: Upload your CSV

On the **Upload** page, click **Choose File**, select your exported CSV, then click **Upload File**. If you want to just explore the app, click **Use Example File** to load the built-in coffee shop survey instead.

Step 3: Analyze Questions

Go to the **Analytics** page. All survey questions will appear as clickable cards. Use the search bar or type filter to narrow them down.

Step 4: View Statistics and Charts

Click any question card to see a response preview and auto-detected question type. If the type is wrong, use the **Question Type** dropdown to correct it and click **Save Type**. The statistics and chart will update automatically. Click **Download Graph** to save the chart as a PNG.

Developer Documentation

Analytics Dashboard for the Empower Student Center

Chrys Nelson, Ari Osmun

Running Locally

Requirements

- Python 3.9+
- A modern web browser (Chrome, FIrefox, Safari, Edge)

1. **Set up the virtual environment**

```
python -m venv .venv
source .venv/bin/activate          # macOS/Linux
# .venv\Scripts\activate           # Windows
pip install -r requirements.txt
```

2. **Start the Flask Backend**

```
python backend/main.py
```

The server will start on http://localhost:5001

macOS note: Port 5000 is reserved by AirPlay Receiver, so we use 5001. If you see a 403 error, make sure you're hitting port 5001, not 5000.

3. **Open the App**

Navigate to **http://localhost:5001** in your browser. Do not open the HTML files directly - they need to be served through Flask for uploads to work

Running Tests

```
pytest backend/test_main.py -v
```

36 tests covering upload parsing, auto detection, and all API endpoints.

Tech Stack

- **Frontend** - HTML, CSS, JavaScript
- **Charts** - Chart.js
- **Backend** - Python 3.9+, Flask
- **Deployment** - Render (free tier)
- **CI** - Github Actions (lint + tests on every push)

Project Structure

```
project5/
    frontend/       # HTML, CSS, and JavaScript
    backend/        # Flask app, tests, and example CSV
    .githib/        # CI workflows
    index.html      # Github Pages Landing page
```

Makefile Commands

```
make run            # Start the Flask dev server
make test           # Run the test suite
make lint           # Run all linters (Python, HTML, JS)
make format         # Run all formatters
make check          # Format then lint
```

Contact and Support

Project Team

- **Ari** - Computer Science / Data Science - osmunaj@gmail.com
- **Chrys** - Computer Science / Psychology - howlhund@gmail.com

Client

- **Susan Kelly** - Empower Student Center
- **Organization:** Montana State University

Report an Issue

For End Users

Email the team with a description of what you tried to do, what happened, any error messages, and screenshots if possible

For Developers

Open an issue on GitHub.