Canvas LTI Student Climate Dashboard

Final Report

Team: sddec21-19
Client: Henry Duwe
Adviser: Nick Fila
Team Members
Andrew Dort
Emma Paskey
Joshua Slagle
Zach Borchard
Kira Pierce
Team Email: sddec21-19@iastate.edu
Team Website: http://sddec21-19.sd.ece.iastate.edu/

Fall 2021

Table of Contents

List of Figures

Acknowledgements

A very special thanks - just as at the end of our first semester -to our advisor Nick Fila and client Henry Duwe for giving us technical assistance in regards to Canvas data curating and project requirements, in addition to being extremely helpful in the entire design process. Throughout the entire implementation of the design both Nick and Henry have been nothing but supportive, flexible, and encouraging and has made the entire experience of senior design even more enjoyable than it normally would have been.

Project Design

Introduction

Terms and Definitions

Persona - An overarching behavioral type driven by a defining set of characteristics. Used to generalize a group of similar students and model their responses to crafted scenarios.
Journey Map - A visual representation of a student's experience throughout the duration of the class. The x-axis represents time, discretized as a series of events. The y-axis represents the resonance of the class with the student.
Resonance - The level of impact the class has on a student and their academic or professional career. Note that this can be positive or negative.

Problem and Project Statement

The problem the project addressed was Canvas's lack of extensive data analysis tools for measuring and visualizing the impact courses have on students. Specifically, Canvas lacks the support to automatically collect student input from climate surveys and show trends in resonance throughout a semester. Our client wanted one place to easily track the success of his students to help him improve his current and future classes.
Our solution to this was to implement a web-based journey map that will analyze feedback given by students, alongside their grades and the time it takes them to turn in assignments to display how well they are progressing and emphasize trends that make the class better or worse for different types of students. This will also include the ability to do automatic resonance prediction and rudimentary data analysis for the client's use.

Intended Users and Uses

The most important end user(s) are the professors that will use this to reflect on their class delivery, both as a whole and as separate components. For example, a professor could tell by the journey map that, although the personas of the students may have varying degrees of success in

the class, no one is struggling or holds bad sentiment towards it and its delivery. Or on the other hand, say one persona appears as having low - or negative - resonance towards the class. The instructor can now click on the specific events that drove this persona's resonance down to get a breakdown of what went wrong. This can be thought of as similar to the debugging of a classroom experience.

Operational Environment

This product will operate in an online environment and need to be built securely to reduce risk of data breaches. Specifically, the computer software will be hosted on a VM provisioned to us by ETG running Ubuntu 20.04. Inside of the Linux VM, we will be running the container orchestrator *Kubernetes* to spin up and scale our application's pods (containers). The VM will operate inside of the ISU VPN so that no traffic external to the school's network can reach the page.

Evolution of Project Design

Much of the evolution of our product design was driven either by 1) limitations imposed by the system we were working *with* (Canvas) and not the system we were working *on* or 2) by a reduction of stretch-goal scope to focus on more critical portions of the application. This being said however, there was not a whole lot that changed in our project design from the end of our design phase to the end of our project's completion. This sentiment can be backed up by comparing the two diagrams below.

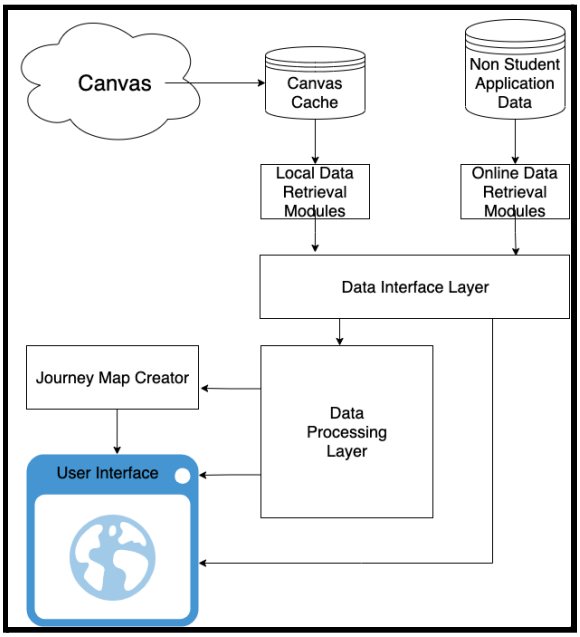


Figure 1.1 – Original Block Diagram

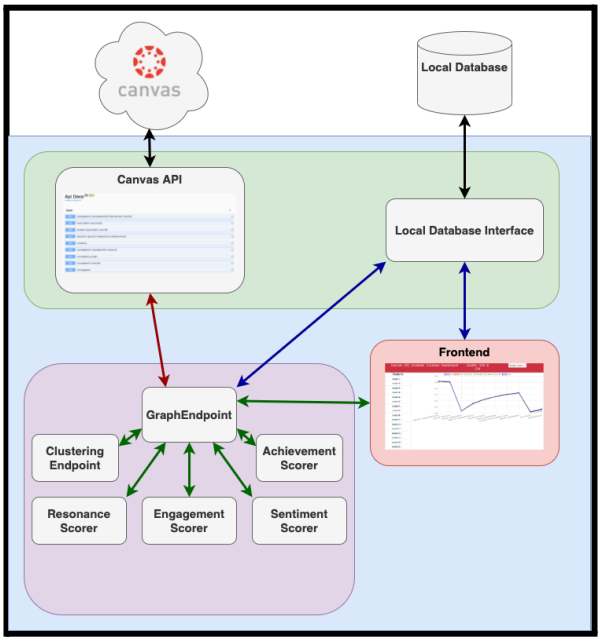


Fig 1.2 – Application Block Diagram

The diagram on the left shows our final design diagram while the figure on the right side shows the breakdown of our final product. The diagrams flip the position of the frontend UI and the data

processing layer (now known as the data analysis pipeline), but are relatively unchanged otherwise. The only large difference between our design and final product is that, originally, it was assumed that the user would need to have Docker installed on their computer to be able to run the local web page / database cache. This assumption was conservative and is no longer needed because we were able to push all containers and data processing pipelines to the VM. This allows the webpage to be accessed by means of a web browser such as Chrome and does not require the user to download and run the containers on their local machine. This is not something that we strove for - we were completely open and expecting to change our design - but rather is more likely a reflection of the care put into designing the application last semester. Our team spent all last semester designing (and redesigning) our application amongst the team with different architectures, receiving feedback from our client and advisor along the way. We believe that the meticulous designing and exploring of our client's desires allowed us to progress steadily with the implementation of our application without too many roadblocks.

One functional change to our project's requirements came once we started working with the Canvas API and realized what limitations it had on serving data quickly. These problems specifically were related to the time it takes to retrieve data from the Canvas API. Because the requests were so slow, in order to fulfill our cold-start and warm-start time requirements, we needed to implement a cache for the Canvas data. This cache would make the real-time requirement of our application hard to satisfy. Luckily, with the blessing of our client, we were able to modify the requirement from a real-time update to a requirement to include a button that would force a cache refresh. This was okay with our client because they preferred a quick refresh over real time updates.

The requirement "*The system should be able to predict how a group of students with given personas will react to an existing class outline."* was reduced in focus as to complete the more vital portions of the application and are in the scope for future work after handing off the application to (possibly) another senior design team as has been discussed with the client.

Finally, the following user-interface requirements were not implemented due to time constraints and more weight being given to the functionality of the graph and the data analysis pipeline.

- The user interface color should be customizable
- Design Layout to be accessible (Ex:color blindness)
- View multiple Journey Maps at once
- The UI should be customizable ^
- The UI should be able to zoom in/out.
- The UI should show an appropriate level of information with respect to the zoom level.
    - The zoomed in view should display data at a more granular level
    - The zoomed out view displaying things should be more abstract

Requirements

We have three sections of requirements here: the traditional Functional Requirements, the traditional non-functional requirements, and then a section dedicated specifically to the user interface requirements since that portion is a large area of focus for our application.

Functional Requirements

- Professor should be able to view class Journey Map
- Professor should be able to view individual students' Journey Maps
- The system should be able to pull data directly from the Canvas API
- The system should be able to accept and use data directly provided to it from a user
- Professors should be able to create their own data/feedback metrics
- Professors should be able to create a new Journey map
- Professor should be able to create groups of data together
- Professors should be able to view journey maps with only a specific set of variables taken into account
- The system should create personas on a per-course basis
- The system should be able to convert data into Journey Maps
- The professor should be able to interact with the Journey Maps
- The professor should be able to inspect data at a more granular level through some action

Non-Functional Requirements

- Data integration should be modular for future extensions
- Student's data should not be accessible by other students
- All TAs, Professors, and students should be able to access the application with no crashes
- The system should be modular
- The system should be easily extensible
- Data Storage should not violate FERPA
- The system should use open source datasets in the absence of actual data for training (i.e. sentiment analysis)
- The system should be able to use mock data
- The system should be accessible at all times
- A journey map should be cold-constructed within 60 seconds.
- A journey map should be warm-constructed within 5 seconds.

User Interface Requirements

- The user interface needs to utilize color
- The UI needs to be interactive (toggle items on/off)
- The UI needs to be resizable and accommodate multiple screen sizes
- The UI should be able to show particular student subsets

Standards & Practices Used

Below is a list of all the standards that we used in the development of our application. These are very common standards in today's industry for creating software. Following these standards closely gave us the ability to write software that was easy to maintain, update, and make constant changes quickly across all platforms. There were numerous times this semester that the ability to simply make a change to our code, build a new container and push it up to our cluster allowed us to debug our application faster by orders of magnitude.

- Agile
- Acceptance / Integration testing
- CI/CD
- Docker
- SOLID Principles
- Kubernetes

Engineering and Technical Constraints

The client imposed no programming language, operating system, libraries, or frameworks that we had to use. Instead, our technical constraints were derived from the non-functional requirements that we discovered while talking to the client. Specifically, in our non-functional requirements we had six key items that drove our constraints:

- Data integration should be modular for future extensions
- The system should be modular
- The system should be easily extendible
- Data Storage should not violate FERPA
- A journey map should be cold-constructed within 60 seconds.
- A journey map should be warm-constructed within 5 seconds.

In addition, we took into account the programming language experience of the team for speed of execution. The requirements that our system be modular and extendible naturally drove us to choose microservices for our architecture; although, in our decision making process, we heavily considered MVC and Microkernel as well. In addition, our team is familiar with a wide range of languages: C, C++, C#, Python, Javascript, Java, the .NET framework just to name a few. Using microservices allowed us to take advantage of this, since each microservice can be written in a different language.

The other big driver was the security concern for FERPA. This means that we cannot just pull all of the student data and store it in our online server with our application data. However, the bottom two of our listed non-functional requirements deal with speed, so we cannot pull from Canvas every time we wish to update or perform a calculation. Instead, we opt to do a full pull of the data we will use at the start and cache it for later.

Security Concerns and Countermeasures

Because we are dealing with student data we had to be extra cautious with how we are handling the data our application touches. We took great care in our decisions while designing and implementing our app in order to ensure no student data is leaked. Specifically, we made sure that no data was kept on machines unless absolutely necessary. Thus, any data that was quick to recall from canvas when needed was not cached and thus could only be accessed with a canvas access token. In the cases where it was necessary that data be cached for speed-of-retrieval-purposes, the data is either stored on the instructor's computer running the browser or it is cached for a brief period of time in one of our microservices behind a firewall and obfuscated by kubernetes. The data cached on the instructor's laptop contains nothing they shouldn't have access to and goes away as soon as the browser is closed. For data that is cached is all stored in memory on the server:

- The data is also not personally identifiable data.
- All other data that is personally identifiable can only be obtained from accessing the Canvas API.

This is mainly to stop the violation of FERPA as well as maintain overall security of our data by having most of it handled directly by Canvas. It is a requirement that the instructor log in before accessing the data.

In addition, our application is being given access to the professor's canvas access token. This is needed in the application set up when spinning up the Canvas API Wrapper microservice. The process we use to supply the canvas access token to the Canvas API Wrapper microservice while allowing for it to change later takes very careful care to:

- Ensure that the token is not hard-coded anywhere inside the code.
- Keep the token out of any bash history that would be present if supplied as a command line argument.
- Keep the token encrypted in the one place it must live (since we do in fact still need access to it).

To accomplish these goals, we use *Kubernetes Secrets* paired with container environment variables. The detailed process for supplying the bearer token to the application is outlin ed in the *Deploy Application* portion of Appendix I, however in general we solve this issue by:

- Only temporarily storing the canvas access token inside a file (avoids the bash history containing the access token).
- Use a deployment script that overwrites a placeholder for the canvas api token (using sed) inside the kubernetes secret file (avoids hardcoding of the token).
- Kubernetes Secrets are automatically encrypted (based 64) when reading the secret out to the CLI, so in addition to needing access to the kubernetes config file, a malicious actor would also need to know the name of the stored secret (not provided in a 'list of keys' anywhere) and would need to decrypt the canvas api token to make use of it.

Implementation Details

Our application has four main components as depicted in the figure below:

- Frontend User Interface
- Data Analysis Pipeline (DAP)
- API Wrapper
- Data Storage

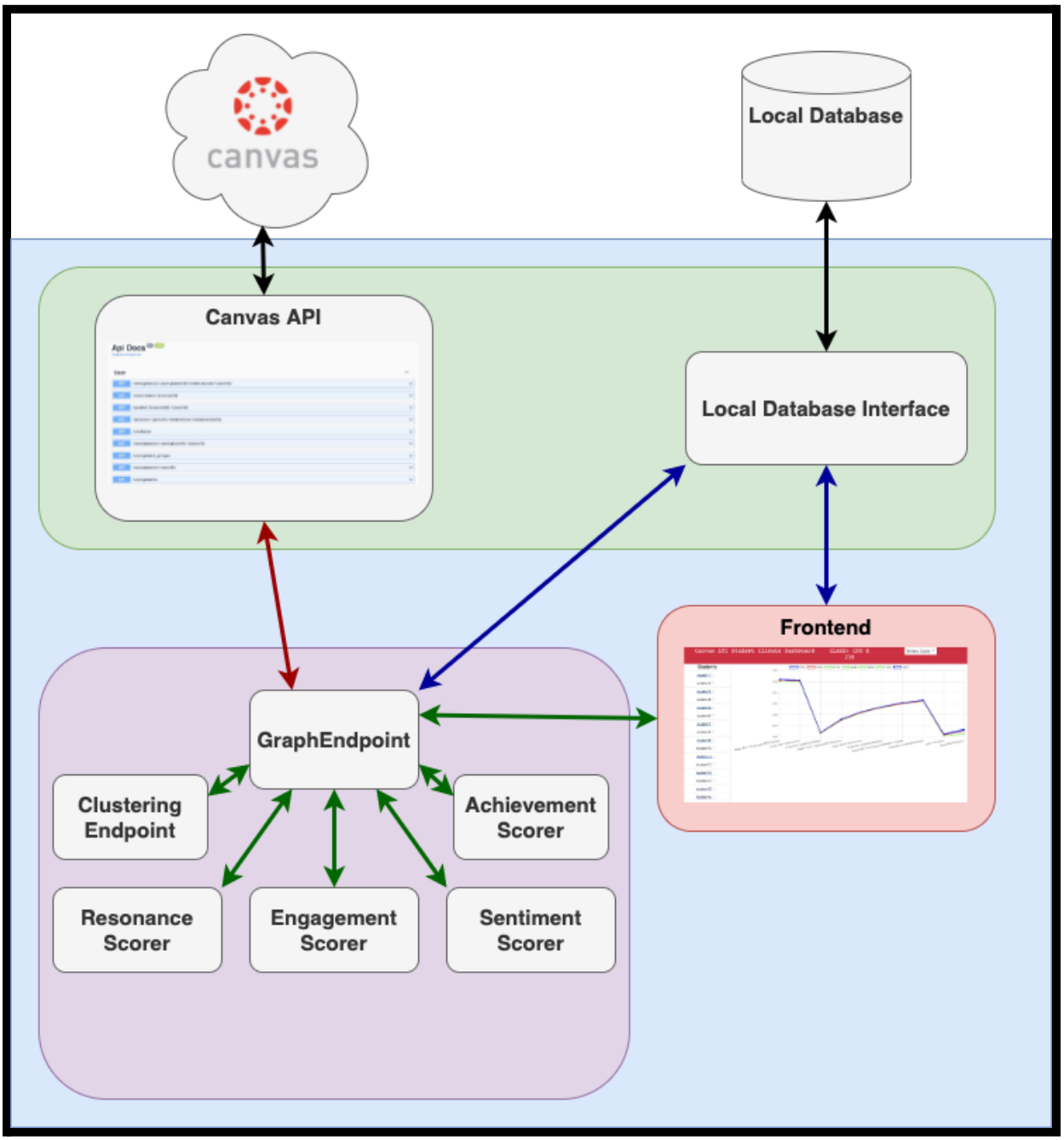


Figure 2.0 – Application Block Diagram Enlarged

Each component of the application has its own set of distinct responsibilities and is deployed as either its own microservice or its own set of microservices. Specifically, the responsibilities of each component of our application is as follows:

- **Front End (Red)**: Requests data from the data analysis pipeline and displays it in an interactive fashion for the user.
- **Data Analysis Pipeline (Purple)**: Serves the front end's requests by fetching the necessary data from the API wrapper and processing it based on user preferences specified on the GUI.
- **API Wrapper (Green- Left):** Hosts endpoints for the data analysis pipeline. Requests data from Canvas, which returns both relevant and extraneous data. The API Wrapper filters out unnecessary data and sends back only what the pipeline needs.
- **Data Storage (Green - Right)**: Data caching on API endpoints for speed / minimizing API calls to Canvas as well as storage and endpoints for all non-canvas data that the application needs to run.

Between the different components we use three primary forms of communication. The first form of communication is Google Protobufs (represented in the diagram above as green arrows). Google Protobufs are a way of abstracting away, to the highest level, the necessities of a communication protocol without having to worry about the low-level implementation. Protobufs are supported by a host of different programming languages and thus aid in our microservice-based approach. The second form of communication between our components is JSON (represented in the diagram by red arrows). This form is found primarily in applications dealing with our canvas API wrapper microservice. The third and final form of communication within our application is the standard TCP/IP communication (represented by blue arrows in the diagram) and appears whenever we are accessing our Data Storage component (using SQL).

In general, all of our requests initially begin by accessing the frontend of our application to log in and then view the page associated with your course. As soon as this page is accessed, a protobuf is sent to the graph endpoint portion of the Data Analysis Pipeline which in turn requests data from the canvas api and begins the entire data analysis process before returning the results to the frontend.

Frontend

Overview

The Frontend service is a node.js application that utilizes Express, Chart.js, Bootstrap 5, and Okta to deliver its core functionalities:

- Providing a web application
- Charting the journey map
- Authentication and Authorization

These functionalities are expanded upon in the sections below.

Web Application

The Express web application handles routing and integration of additional node.js packages; this is the master hub of tools used by the web application. Settings for all middleware (Okta authentication controls is an example of middleware) are defined and used by the express application, which enables control over routing access and app endpoint responses.

Charting the Journey Map

The journey map utilizes information calculated resonance from the Data Analysis Pipeline to populate and display the graph. Below is an overview of the communication and graph-creation process.

- Frontend calls the API and translates the JSON response to graph data points.
    - Data points get translated to EventList, StudentList, and GroupList within session storage for use throughout the application.
- Arrays of the student and group data are formatted to work with Chart.js by separating their elements into data and labels.
    - The skeleton of the chart calls the function with the formatted elements to populate the graph.
    - The y scale is limited to the space between -1 and 1 as resonance will never exceed those values.
    - Apply the graph title and disable the legend.
    - The onClick plugin is modified to activate only if a node is selected and pass the students and groups active at that node to sessionStorage for a secondary highlight page, as well as the value and event label of the node.

Charting the Journey Map

Also relevant to journeymap creation are the inputs for weight fields provided on the user interface. These include weights for assignment type and the three criteria our application uses to draw from student feedback (achievement, sentiment, and engagement). The user inputs numerical values to increase or decrease the “weight” of each criteria when calculating student resonance. These values are used by the Data Analysis Pipeline to provide custom interpretations of the data.

- The resonance fields form is hard coded because these 3 criteria are always present.
- Assignment weight fields are created dynamically for each course; the system pulls these values from the Canvas API, and creates input fields for each assignment type.
- Weight fields are targeted by the protobuf when redrawing the graph. Values are initially loaded with the course’s default assignment grading weights, but can be set to load using a user-defined default filter. Values can be loaded into the weight fields based upon any filters the user has defined and saved.
- The application utilizes session storage to manage which filter is currently selected.

Filters

Filters store user-defined settings for a journey map: assignment and resonance weights and group settings used to load lines on the journey map that represent an averaged resonance for multiple students. These filters are saved to the SQL Database and accessed upon load. The user is able to define a default filter that loads when they access their application.

Authorization and Authentication

The web application implements Okta authorization and authentication controls, as well as Okta sessions, to provide user security. The user is responsible for configuring the application to run and connect to their desired Okta organization. Once this connection is established, the user enters their credentials into the Okta login widget, and the application redirects the user to the main user interface upon successful login. This page is configured to be secure and users must be logged in to access it. Logging out will redirect the user back to the login page.

Canvas API Wrapper

Overview

The Canvas API wrapper was a .NET core application that was used to pull data from the Canvas API. This application is then called from our Data Analysis Pipeline that will do data processing and sentiment analysis. Essentially this is just a wrapper for the canvas api.
Added functionality on top of existing Canvas API:

- Caching on certain Canvas endpoints to speed up the retrieval process on endpoints that need to be called often.
- Automated pagination handling. For many endpoints on Canvas the responses are returned in a paginated form and there is no way to specify the retrieval of all data that comes from these endpoints even though it is needed. So we handled that with a custom

method that will read the link headers and iterate through the pages of data that the canvas api will return.

- Aggregate objects. In many cases we needed data from the canvas API that did not exist in a single endpoint, for instance grabbing all assignments along with their submissions. To solve this we created aggregate objects that would be constructed from calls from multiple different endpoints and serialized into one object. Essentially adding functionality that did already exist in the canvas API.

In addition, all endpoints that exist for the Canvas API wrapper are documented on the Swagger endpoint for our application. This is all automated as well so when changes are made to the applications code, such as adding new endpoints or modifying objects, those changes will be shown on our Swagger doc. Giving the end user or developer the ability to have a source for all information about the wrapper and usable endpoints.

Micro-Services.

Overview

All applications are split into their own Docker container. The reason to take this microservice approach is to thinly slice our application to prevent a monolith software from being formed. This allows us to easily troubleshoot certain services as well as keeping our overall application highly scalable and decoupled from just one application. This is a large industry standard and after weighing out the pros and cons of implementation details we decided that this would by far be the best solution. Mainly because this application will need to be maintained by others after we are gone.

Data Analysis Pipeline

Overview

The data analysis pipeline (DAP) is written entirely in python and religiously follows the microservice architecture. Like all portions of our application, the microservices that make up the DAP are containerized using docker and deployed/orchestrated using kubernetes on a cluster. Unlike other portions of the application, the DAP utilizes Google Protobufs as its primary form of communication for both requests that stay inside of the DAP and for requests to the DAP itself. There are six main components of the data analysis pipeline:

- Graph Endpoint
- Clustering Endpoint
- Achievement Scorer
- Sentiment Scorer
- Engagement Scorer
- Resonance Scorer

As can be seen in the figure below, the main endpoint that communicates with the external world is the Graph Endpoint while the other components of the DAP are subservient to it.

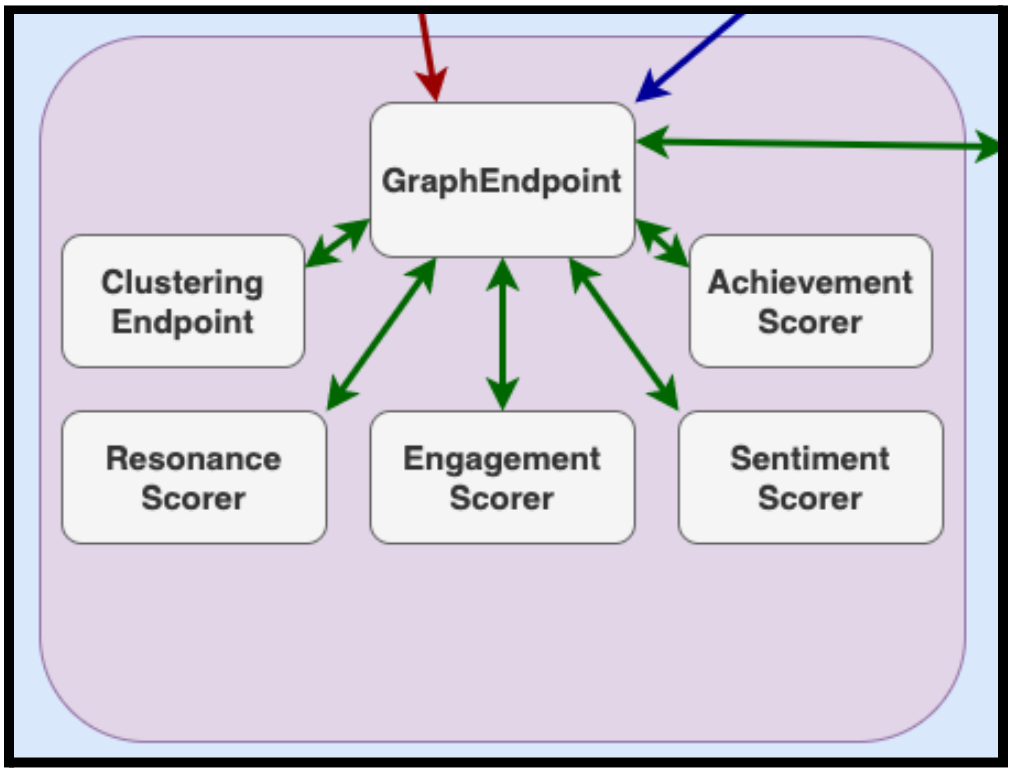


Figure 3.0 – Data Analysis Pipeline

Specifically, once a request is made to the GraphEndpoint microservice, the following steps are completed *in parallel* using multi-threading in order to reduce the runtime of the application:

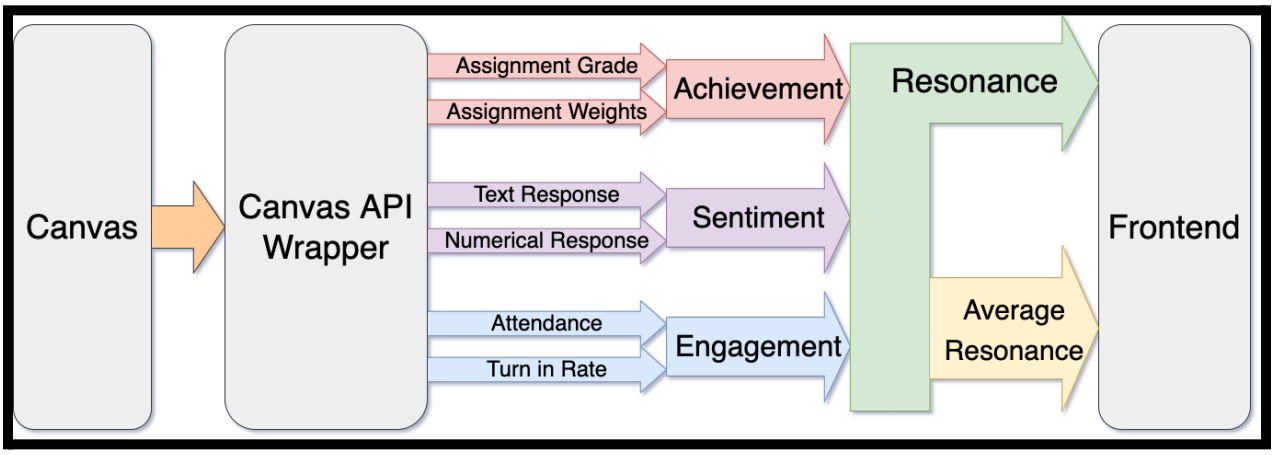


Figure 4.0 – Application Use-Case Flow

1. A call is made to the backend database to get the list of students in a class.
2. A call is made to the Canvas API Wrapper to get the student data for each student within the list just recieved.
3. The resonance is calculated for each "event' - i.e. homework, test, quiz, lab - for every student, taking into consideration the assignment weights provided from the UI. The resonance is calculated from the three components *Achievement, Sentiment, and Engagement*:
    a. The Assignment Grade is sent to the Achievement Scorer microservice to calculate the achievement score of the event.
    b. The Text Response and Numerical Responses that students provided in Canvas Quizzes (if they exist) are sent to the Sentiment Scorer microservice to calculate the sentiment score of the event.
    c. The Attendance (if exists) and Turn-in-Rate are sent to the Engagement Scorer microservice to calculate the engagement score of the event.
4. The average resonance is calculated for each student taking into consideration the resonance weights provided from the UI.The average resonance is calculated from three components *Average Achievement, Average Sentiment, and Average Engagement*:
5.
    a. A list of achievement scores, assignment types, and assignment weights are sent to the Achievement Scorer microservice to calculate the average achievement score of the student.
    b. A list of sentiment scores that students provided in Canvas Quizzes (if they exist) are sent to the Sentiment Scorer microservice to calculate the average sentiment score of the student
    c. A list of engagement scores are sent to the Engagement Scorer microservice to calculate the average engagement score of the student.

6. The clustering microservice is provided with a number of groups to create and a list of students with their most recent average achievement, sentiment, and engagement to categorize the students according to a 3-Dimensional K-means clustering algorithm. A depiction of the group formation in 3D is given below:

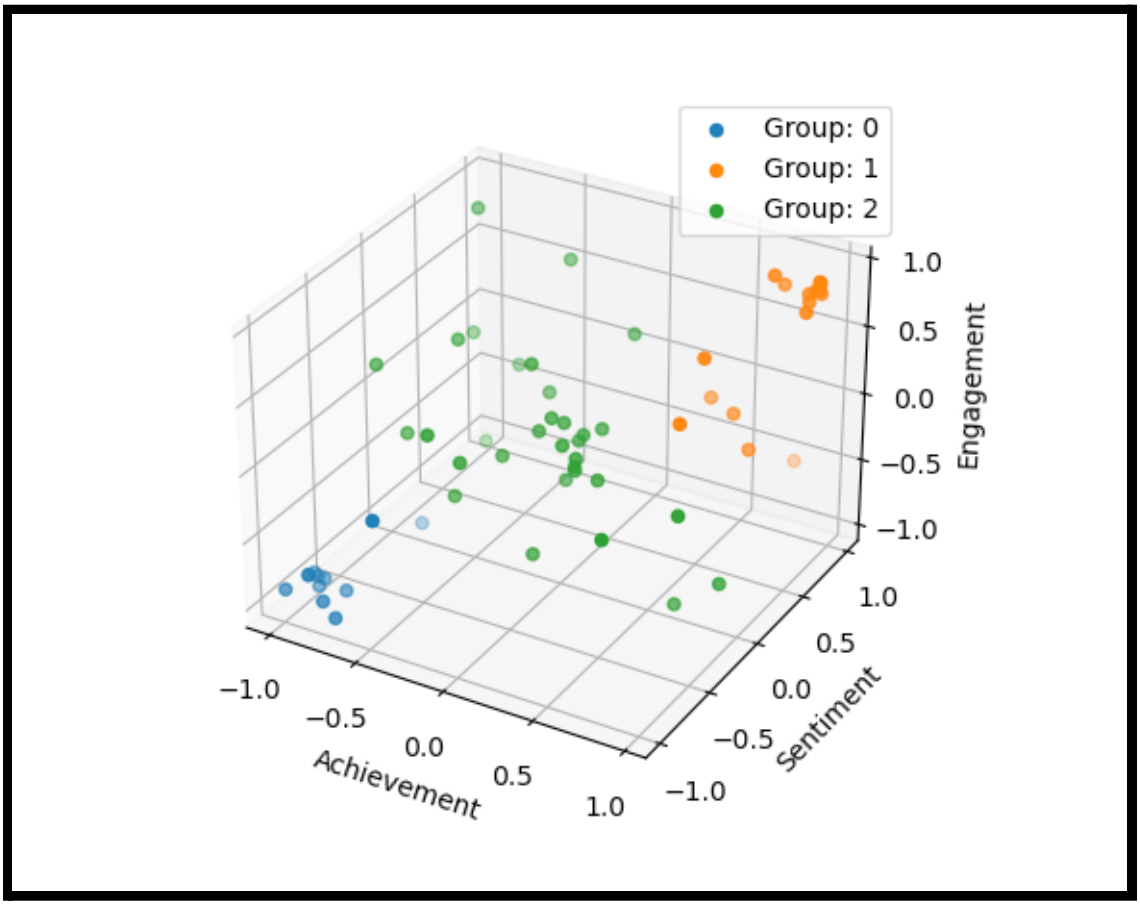


Figure 5.0 – 3D Grouping Visualization

7. Finally the data required to display our journey map is passed back to the frontend for display on the graph.

Protobuf and gRPC Diagrams

All of the microservices that make up the DAP implement the Google Remote Procedure Call (gRPC) framework. That is, each component is implemented as a microservice that acts as a gRPC server that sits idly and waits for requests. These requests/response pairs are represented by the green arrows in figure 3.0. Each request and response take the form of protobufs. The UML diagrams and definitions for each of the gRPC and Protobuf services are included in the Appendix III.

Canvas API Wrapper

Overview

The Canvas API wrapper was a .NET core application that was used to pull data from the Canvas API. This application is then called from our Data Analysis Pipeline that will do data processing and sentiment analysis. Essentially this is just a wrapper for the canvas api.
Key features:

- Caching on certain Canvas endpoints to speed up the retrieval process on endpoints that need to be called often.
- Automated pagination handling. For many endpoints on Canvas the responses are returned in a paginated form and there is no way to specify the retrieval of all data that comes from these endpoints even though it is needed. So we handled that with a custom method that will read the link headers and iterate through the pages of data that the canvas api will return.
- Aggregate objects. In many cases we needed data from the canvas API that did not exist in a single endpoint, for instance grabbing all assignments along with their submissions. To solve this we created aggregate objects that would be constructed from calls from multiple different endpoints and serialized into one object. Essentially adding functionality that did already exist in the canvas API.

In addition, all endpoints that exist for the Canvas API wrapper are documented on the Swagger endpoint for our application. This is all automated as well so when changes are made to the applications code, such as adding new endpoints or modifying objects, those changes will be shown on our Swagger doc. Giving the end user or developer the ability to have a source for all information about the wrapper and usable endpoints.

Data Storage / Access - FLASK APPLICATION ENDPOINTS

The data storage / access portion of our application was created by spinning up a MySQL database on the VM. In order to access this VM however, we needed to provide an API to create, read, update, and delete table entries. To do this we created a containerized REST API that allowed for the CRUD operations written in Python using the Flask framework. The following are a list of the

endpoints (relative to the base URL: http://sddec21-19.ece.iastate.edu:30031) along with their required parameters and example requests:

Adding a Filter

/addFilter
POST
Adds filters to the database using provided JSON object
Example object:

```
{
   "courseId": 1,
   "userId": 1,
   "filterName": "My filter",
   "assignmentWeights": {
      "assignment": 1,
      "exams": 2,
      "quizzes": 3,
      "labs": 4,
      "other": 5
   },
   "achievementWeight": 7,
   "sentimentWeight": 8,
   "engagementWeight": 9
}
```

Getting a List of Filters

/getFilterList
GET
Returns a list of all filters based on userID and courseID
Example request:
http://sddec21-19.ece.iastate.edu:30031/getFilterList?courseID=1&userID=1

Getting a List of Filter IDs

/getFilterIDList
GET
Returns a list of all filters based on userID and courseID
Example request:
http://sddec21-19.ece.iastate.edu:30031/getFilterIDList?courseID=1&userID=1

Getting Filter Data

/getFilterData
GET
Returns a JSON object containing all of the data for a specified filter
Example request:
http://sddec21-19.ece.iastate.edu:30031/getFilterData?courseID=1&userID=1&filterName="My filter"

Deleting a Filter

/deleteFilter
GET
Deletes a filter from the database
Example request:
http://sddec21-19.ece.iastate.edu:30031/deleteFilter?courseID=1&userID=1&filterName="My filter"

Adding a User

/addUser
POST
Adds a user to the database using provided JSON object
Example object:

```
{
   "userId": 1
}
```

Adding Students

/addStudents
POST
Adds student(s) to the database using a provided JSON array
Example array:
[{ "studentID": 1, "courseID": 1}, { "studentID": 2, "courseID": 1 }]

Adding Students to a Group

/addStudentsToGroup
POST
Adds students to a group using provided JSON array
Example array:
[{ "studentID": 1, "groupID": 1}, { "studentID": 2, "groupID": 1 }]

Getting Student Data

/getStudentData
GET
Returns a JSON object containing all of the data for a specified student
Example request:
http://sddec21-19.ece.iastate.edu:30031/getFilterData?courseID=1&studentID=1

Adding a Group

/addGroup
POST
Adds a group to the database based on the provided JSON object
Example object:

```
{
   "groupNum": 1,
        "groupName": "My group",
        "filterID": 1,
        "courseID": 1,
        "userID": 1
}
```

Getting a List of Groups

/getGroupList
GET
Returns a list of groups based on course and user
Example request:
http://sddec21-19.ece.iastate.edu:30031/getGroupList?courseID=1&userID=1

Getting a List of Groups by Filter

/getGroupListByFilter
GET
Returns a list of groups based on course and user
Example request:
http://sddec21-19.ece.iastate.edu:30031/getGroupListByFilter?courseID=1&userID=1&filterID=14

Getting Student List by Group

/getStudentListByGroup

GET
Returns a list of students associated with a group by ID
Example request:
http://sddec21-19.ece.iastate.edu:30031/getStudentListByGroup?groupID=1

Updating Filter Name

/changeFilterName
GET
Changes the name of a specified filter
Example request:
http://sddec21-19.ece.iastate.edu:30031/changeFilterName?current="Current name"&new="New name"

Updating Group Name

/changeGroupName
GET
Changes the name of a specified group
Example request:
http://sddec21-19.ece.iastate.edu:30031/changeGroupName?current="Current name"&new="New name"

Deleting a Group

/deleteGroup
GET
Deletes a group from the database based on ID
http://sddec21-19.ece.iastate.edu:30031/deleteGroup?groupID=1

Deleting a Student

/deleteStudent
GET
Deletes a student from the database based on student ID and course ID
http://sddec21-19.ece.iastate.edu:30031/deleteStudent?courseID=1&studentID=1

Testing Process and Test Results

Canvas API wrapper / Databases.

The following is a list of all the tests ran against our API Wrapper and Databases:

- Verification against calling the Canvas API and data that exists on our backend.
- Backend speed testing to determine API call times and loads.
- Testing on our local application against the external Iowa State Canvas API.
- Calling the endpoints from the Data Analysis Pipeline and sending it over to the front-end.
- SQL testing

Each header in the remainder of this section provides details about the test (if not self explanatory).

Verification against calling the Canvas API and data that exists on our backend

Backend speed testing to determine API call times and loads

The main reason we took this approach was because the Canvas API is already heavily tested, and even if there were issues with it there are no changes that we would be able to make. All validation was just making sure the canvas API returned the same responses when called from our dotnet wrapper, and the objects were aggregated correctly.

Results: The runtime for some endpoints that collected data from multiple endpoints and made 20+ api calls took around 50+ seconds. After changing our serialization library we were able to bring it down to 20 or so. On top of this caching was added and tested manually. This brings down the time to under a second if the data is cached.

Testing on our local application against the external Iowa State Canvas API

Shown in the picture below. Data was called using a HTTP tool such as insomnia or postman. This data retrieved from the canvas api was matched against the endpoints we wrapped and called from within our application. We then compared them side by side and made sure they had similar runtimes and the same data (unless response objects were modified to shorten the amount of data returned). This was to make sure our endpoints functioned as expected and the serialization to our response objects was populated in the correct format.

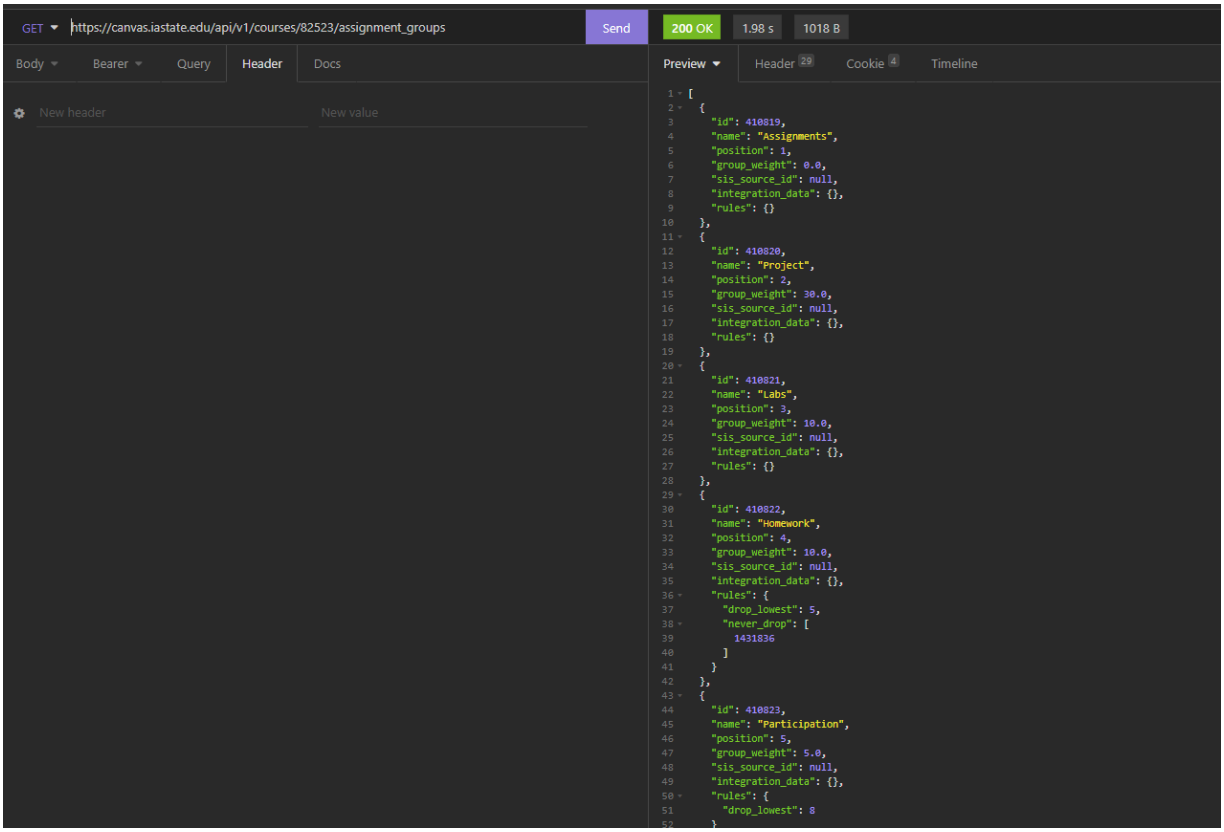


Figure 6.0 – Insomnia Testing

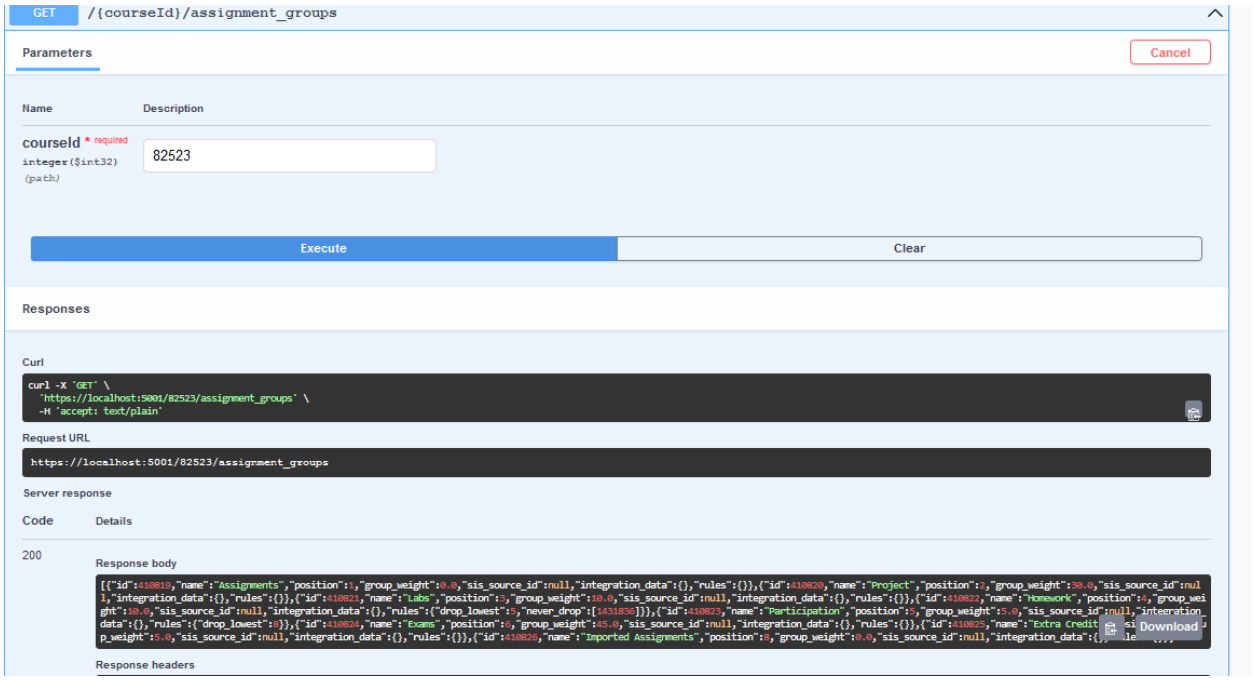


Figure 7.0 – Swagger Testing

Calling the endpoints from the Data Analysis Pipeline and sending it over to the front-end. Essentially running a full end to end integration test on many different occasions.

SQL testing

Due to the complexity of sql testing, the majority of testing that was completed was by running temporary transactions to run select and modify statements. This was done to verify that the tables we created are correctly storing our data that the front end uses without any issues. From that we made sure the same data is what is being displayed on the front-end with the correct values mapped on our graphs.

Data Analysis Pipeline

For DAP, each microservice was tested independently by creating a client that sends different expected and unexpected values for each protobuf message. Once a response was received it was evaluated visually by the tester. All tests were required to both:

- Not crash the application and to have received a response.
- Have the response contain the expected values.

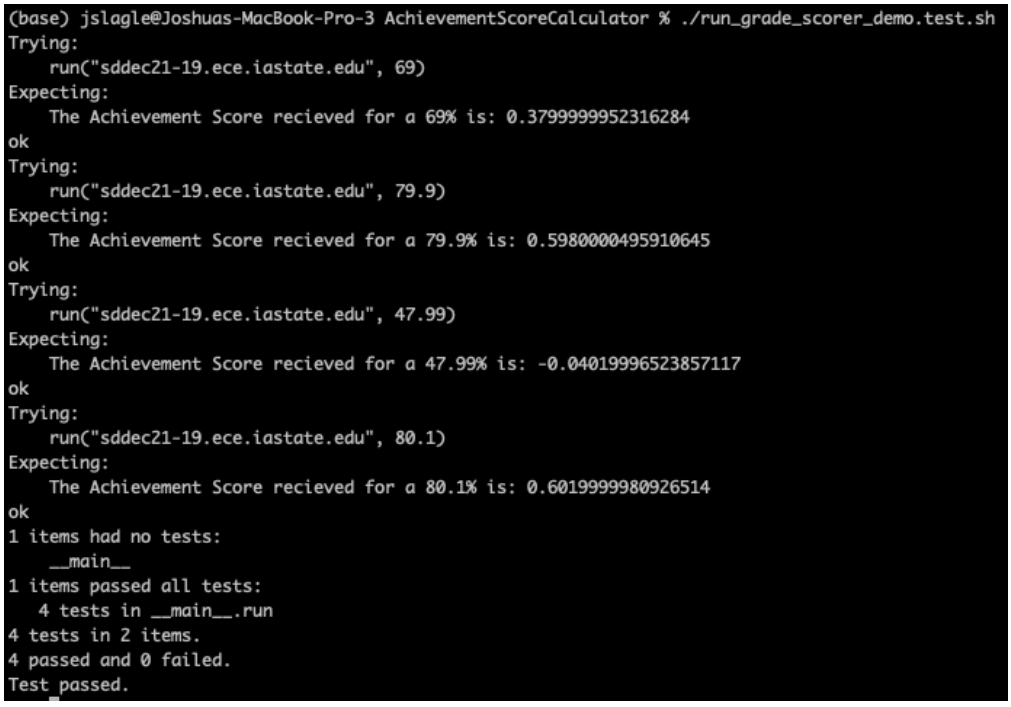


Figure 8.0 – Data Analysis Pipeline Unit Testing

The above image contains one of many unit tests that we wrote. In order to decouple the dependency on the canvas api wrapper and make this a true unit test, we to mocked out the expected responses. From there we made sure that data calculations ran as expected.

Frontend

For our frontend testing we performed multiple testing styles.

Use-case Test with two different paths:

- Happy path: Have a user with full level permissions (mock being a teacher) go through the application and verify speeds and filters on the front end. As well go through and verify the data that was collected on the students.
- Malicious path: Due to this application needing to only be able to be accessed by professors and TAs that the professor grants access to. We tried running the application from the vpn and accessing it as a separate user. This worked as expected and the user was unable to successfully login to the application and view any data that is collected.

Local Differences

- To ensure usability and expected display behaviors across browsers, we performed use-case testing using recent versions of Google Chrome, Firefox, and Safari with passing results and marked no decrease in usability.
- Tested across various display sizes to ensure proper resizing and display behaviors. This was performed using screen resizability tools provided by Chrome Developer Tools and by our teammates accessing and performing use-case testing through their own personal devices. Our team reported expected behaviors and marked these as passing results.

Related Projects and Literature

There are similar products that exist in the market, largely specializing in business and marketing needs. UXPressia is a significant company in this market; their product allows users create multiple, phased journeys and descriptive personas, assign personas to one or more journeys, collaborate with other team members in real-time, create impact maps, and generate high-resolution graphics of their work (UXPressia, 2020).

The key difference between this implemented project and similar tools in the marketplace is that this project is specific to an academic setting and utilizes student feedback. Many of the collaborative features utilized in software on the market are not relevant to this project and may pose security risks regarding student data. Additionally, this project was intended to help professors organize and identify patterns in student feedback. This is done with the goal of improving the student experience in a course, especially in terms of course resonance.

In the academic world, there is another analysis tool created by Michigan called MyLA - or My Learning Analytics (My Learning Analytics Support, University of Michigan) - that is currently being advertised by Unizin, an organization that deals with the digitization of the university experience. Our tool provides two key features that MyLA does not:

- Our tool provides predictive analytics instead of just statistical analysis of Canvas Data
- Our visualization is much more rich and inviting than MyLA's. Where they have numerical values and bar charts, we will have an interactive journey map that displays trends much more easily. The three figures below show their three visualizations.

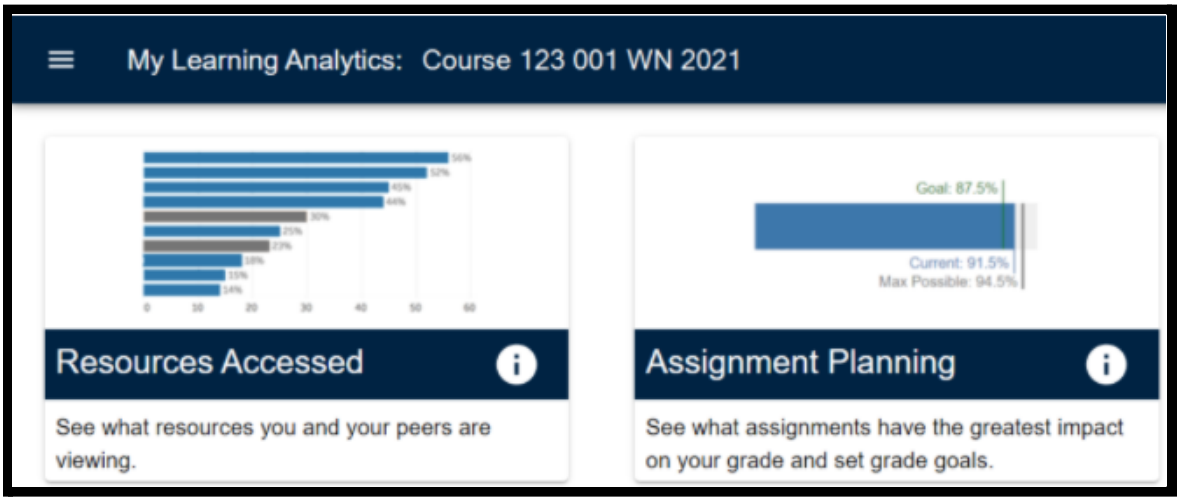


Figure 9.1 - MyLA Resources Accessed

Figure 9.2 - MyLA Assignment Planning

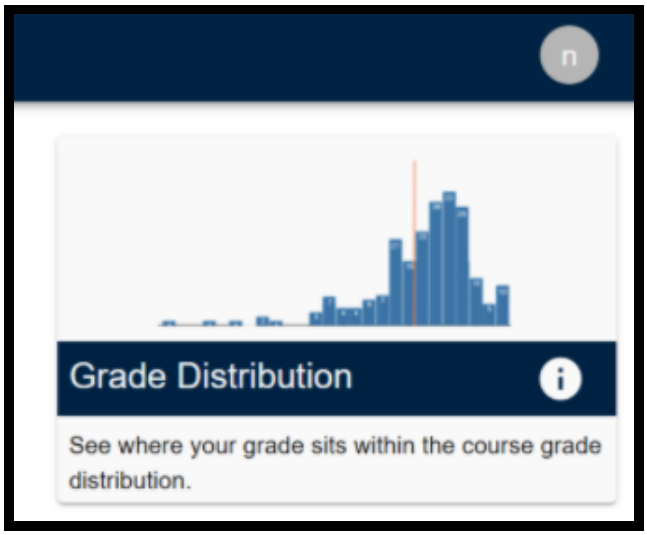


Figure 9.3 - MyLA Grade Distribution

Our UI and graphics, depicted below, provide a completely different, and more in depth view of the class progression.

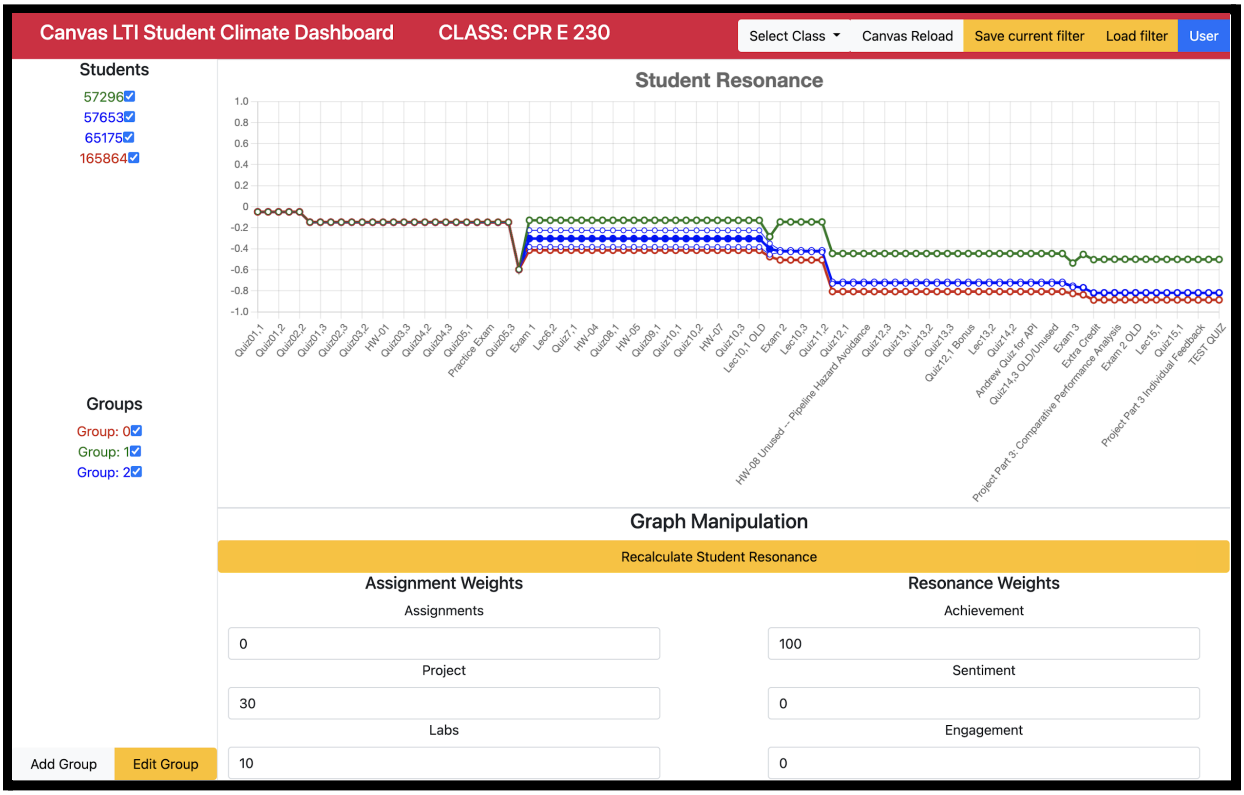


Figure 10.0 - Example Journey Map

These differences give us our niche between the academic product and the corporate product that we feel very comfortable operating in.

Appendices

Appendix I - Operation Manual

Installation of Application

Anytime we reference the *git repository,* we refer to: https://git.ece.iastate.edu/sd/sddec21-19.

Environment Set Up

In the following section of the document we go through the process of setting up our application from scratch. Our application is Kubernetes based and thus we make two key assumptions in that section:

- You have access to a vanilla kubernetes cluster using the kubectl command line tool
- The kubernetes cluster and you alike have access to a container registry on which to store container images

Because kubernetes administration is not a commonly taught skill set at the time of writing, we will briefly go over how to install and set up a necessary cluster and container registry in the context of an Ubuntu 18.04/20.04 Virtual Machine.

In order to set up the cluster and container registry, the following steps will need to be followed:

1. Install Outside Tools
    a. Install Docker
    b. Install Docker-compose
    c. Install Kubernetes (Kubelet, Kubectl, Kubeadm)
2. Set Up Cluster Infrastructure
    a. Initialize the Kubernetes Cluster
    b. Setup Container Networking Interface
    c. Setup Container Registry

Install Outside Tools

Our infrastructure can be set up quite easily without too much understanding of what is going on under the hood. To accomplish this, we will rely on some external packages to do some of the heavy lifting for running the container registry (where we will house our application images - *think binaries*) and kubernetes cluster instantiation/administration. Thus we will need to install the following packages on the VM which the application will be deployed:

- Docker
- Docker-Compose
- Kubernetes (Kubelet, Kubectl, Kubeadm)

Installing Docker on Ubuntu 18.04/20.04

The first thing we want to do is install docker onto the machine which will be running our cluster. This will allow kubernetes to use docker as the container runtime and allow you to build the application from the VM if so desired. To install docker on ubuntu, follow the instructions at this link: https://docs.docker.com/engine/install/ubuntu/ . For linux distributions, it is recommended to add the user to the *docker* users group to manage docker as a non-root user account (link here: https://docs.docker.com/engine/install/linux-postinstall/).

Installing Docker-Compose on Ubuntu 18.04/20.04

The docker-compose toolset allows you to manage containerized applications at a small scale. In our specific usage however, we are simply going to use the command line tool to spin up a container registry that we can push and pull our container images to. To install docker compose, navigate to the link here (https://docs.docker.com/compose/install/), click on our operating system (for us, Linux) in the '*Install Compose*' section, and follow the instructions there.

Installing Kubernetes on Ubuntu 18.04/20.04

Kubernetes is the container management system that we will be using to manage the state of our application. We've provided scripts and YAML files that will set up the kubernetes cluster for you if not already done. Before using the scripts however, we need to install the command line tools that the script relies on. To install these tools, follow the instructions at the link provided here: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/.

Set Up Cluster Infrastructure

At this point you should have kubelet, kubeadm, and kubectl installed on your VM and we can now use the scripts in the git repository to set up the cluster and container registry.

Initialize the Kubernetes Cluster

To initialize the cluster, clone the git repository to the VM on which we are going to run our application. Inside the cloned repository, navigate to the *Infrastructure/Cluster_Specification/Kubeadm* directory. Run the *create-cluster.sh* file and wait for completion. This will create a kubernetes cluster for you with the help of kubeadm and then set the proper permissions and file locations for you to be able to control the cluster.

Setup Container Networking Interface

Next we need to set up the Container Networking interface so that our microservices have a way to communicate with each other. There are tons of different CNIs you could choose to use for different needs, however our application doesn't require anything fancy so we chose to use the most basic (and usually the default) CNI - flannel. To install flannel, navigate to CNI directory

parallel to the Kubeadm directory we were just in (*Infrastructure/Cluster_Specification/CNI*). Once there, run the command:

```
kubectl apply -f flannel.yaml
```

This will create the kubernetes objects necessary and install flannel onto the cluster.

Setup Container Registry

Finally we need to set up the container registry that will house the images that we create for our microservices. The container registry will simultaneously allow the kubernetes cluster to (pull and then run) our application as we specify.

To create the container registry on the VM, navigate to the *Infrastructure/Cluster_Specification/Container_Registry* directory and follow the steps below:

1. Create a directly named *data*
2. Run `docker-compose up`

This will open up a container registry on port 5000 for pushing and pulling to. In order to use this with the just-installed docker instance, edit the /etc/docker/daemon.json file as described in this link: https://docs.docker.com/registry/insecure/. This will allow you to push/pull images to the registry without needing to provide extra SSL layers (no sensitive data will be stored here). Make sure that you've restarted the docker daemon as specified in the link provided.

At this point, you have a Ubuntu 18.04 VM provisioned to you that houses a kubernetes cluster with an installed CNI and image registry. Now we can begin setting up the actual application.

Application Set Up

In order to set up the application, three steps are required:

1. Build the Application's Microservices and Push to Image Registry
2. Deploy the Application's Microservices and Kubernetes Objects
3. Setup MySQL Database

Build Application

To make building the application easy, there is a ' *build-application.sh*' file located in the root of the git repository once cloned. To use this script, simply run:

```
./build-application <REPOSITORY NAME>
```

And wait for the text "BUILD COMPLETE" to be displayed. On the first run, this may take a while. Afterwards, this will take substantially less time. It is important to note that you should *not* include the angle brackets around the repository name when running this script. For example:

```
./build-application sddec21-19.ece.iastate.edu:5000
```

Is correct while:

```
./build-application <sddec21-19.ece.iastate.edu:5000>
```

Is incorrect.

If you've already built most of the application but just want to build a portion, navigate to the directory containing the code for the portion of the application you wish to build and run the script that begins with the text *'image_registry_dockerize'*, passing in the image registry address as an argument just as in the *build-application.sh* script. For example, if we wish to just update the Graph Endpoint Service, we:

1. Navigate to the directory *Data_Analysis/GraphEndpoint/*
2. Run `./image_registry_dockerize_graph_endpoint.sh <REPOSITORY NAME>` with the image repository name you are using.

Deploy Application

At this point in the setup, we have a kubernetes cluster, a container network interface, and an image registry filled with container images that we just built in the last section. All that is left now is to:

1. Set proper Access Token from Canvas
2. Deploy Microservices to Kubernetes

Setting Proper Access Token from Canvas

In order to set the proper access token from canvas, we first need to retrieve the token from canvas. To do this, open up and log into canvas, then follow the steps below:

1. Generate your own Canvas Access Token
    a. Click on the *Account* button on the left side of the screen.
    b. Choose the *Settings* selection that appears on the pop-up menu
    c. Under *Approved Integrations* click the button that says + *New Access Token*
        i. In the *Purpose* Textbox, say "*Canvas LTI Student Climate Dashboard"*
        ii. In the *Expires* Box, choose when you want to need to refresh the token
    d. Generate this token and copy the value to the right of *Token*
2. Set the Access Token in Application
    a. On the VM, go to the *Infrastructure/Application_Infrastructure* directory
    b. Open up the *bearer_token.txt* file, delete all contents, and place only your token in the file. Save.

Deploying Microservices to Kubernetes

Now that the token has been saved to the file, we are ready to deploy the application by following the steps below:

1. Run the *deploy-application.sh* file, passing in the image repository address that you are using, like below:
    `./deploy-application sddec21-19.ece.iastate.edu:5000`

2. Delete your token from the *bearer_token.txt* file.

The writing your token to the *bearer_token.txt* file and then deleting it accomplishes a few things with respect to security:

- Keeps the plaintext representation of the token on the computer to a minimum.
- Hides the token from the bash_history file and anyone who can view your command history.

It is important to note that anyone who has the kubernete token can access this token, so it is important not to distribute this token without care.

Set Up MySQL Database

1. Install MySQL Database
    a. apt install mysql-server
    b. apt install mysql-client
2. Ensure that the server has the correct IPTables/permissions for your specific machine
3. Install MySQL Workbench (optional but easier than command line client)
4. In either workbench or client, log in as root user (or set up a new user)
5. Additionally, add database information in Data_Analysis/SQLConnection/config.ini
6. Setup Database Schema:

```
create table if not exists users (
        userID int unique not null,
        defaultCourse int,
        bearerToken int not null,
        primary key (userID)
);

create table if not exists courses (
        courseID int not null,
        userID int not null,
        primary key (courseID),
        foreign key (userID) references users(userID) on delete cascade
);

create table if not exists categories (
        uniquePairID int not null auto_increment,
        groupNum int not null,
        groupName varchar(256) not null,
        filterID int not null,
        courseID int not null,
```

```
        userID int not null,
        primary key (uniquePairID),
        foreign key (filterID) references filters(filterID) on delete cascade,
        foreign key (courseID) references courses(courseID) on delete cascade,
        foreign key (userID) references users(userID) on delete cascade
);

create table if not exists students (
        uniquePairID int not null auto_increment,
        studentID int not null,
        courseID int not null,
        initialResonance double,
        primary key (uniquePairID),
        foreign key (courseID) references courses(courseID) on delete cascade
);

create table if not exists studentGroups (
        pairID int not null auto_increment,
        studentID int not null,
        groupID int not null,
        primary key (pairID),
        foreign key (studentID) references students(uniquePairID) on delete cascade,
        foreign key (groupID) references categories(uniquePairID) on delete cascade
);

create table if not exists filters (
        filterID int not null auto_increment,
        filterName varchar(256) not null,
        courseID int not null,
        userID int not null,
        assignmentWeights varchar(256) not null,
        achievementWeight int not null,
        sentimentWeight int not null,
        engagementWeight int not null,
        primary key (filterID),
        foreign key (courseID) references courses(courseID) on delete cascade,
        foreign key (userID) references users(userID) on delete cascade
);
```

User Manual

Updating Canvas Access Token

In order to update the canvas access token with a new token, simply:

1. Open up the *bearer_token.txt* file.
2. Delete all contents.
3. Place your new token inside of the file with no extra character.
4. Save.
5. Deploy the new K8s Secret by running:

```
cat bearer_secret.yaml | sed -e "s/<BEARER_TOKEN>/${BEARER_TOKEN}/" |
                          kubectl apply -f -
```

6. All as one line.
7. Delete your token from the *bearer_token.txt* file.

Frontend

1. How to access webpage
    a. On a modern web browser, enter the hostname of your cluster into the URL bar and append *:30011* to the end of the URL.
    b. Navigating to this page takes the user to the Okta-managed login page.
        i. NOTE: The Okta-managed login page must be configured so that login is redirected to their Okta Organization of choice. The user either needs to create and manage their own Okta organization, or set it up with the university's Okta credentials. See #2 for more details.
    c. Enter user credentials and click "Log In". If login is successful, Okta will return a login cookie and the frontend application will automatically redirect the user to *:30011/graph*. This page is inaccessible without the Okta authentication cookie.
    d. The user is now logged in and viewing the main page of the application.
2. How to setup an Okta Login
    a. This process requires making edits to the program's files. Settings for Okta authentication are located in the file "/FrontendApplication/server.js".
    b. The user must be an administrator for the Okta organization they would like to connect to the application. Users are allowed to create and manage their own Okta organization and manage this.
    c. For instructions on how to connect an Okta organization to an Express application, please refer to Okta's thorough documentation on the process.
        i. https://developer.okta.com/docs/guides/sign-into-web-app/nodeexpress/main/#require-authentication-for-a-specific-route

3. How to Save New Filters
    a. In the "Graph Manipulation" section, there are two forms for *Assignment Weights* and *Resonance Weights*. These values saved will correspond to the values in these fields.
    b. Once *Assignment Weights* and *Resonance Weights* are ready to be saved, click the *Save Current Filter* button at the top of the webpage.
    c. A window will appear in the center of the screen, prompting a name for the filter. Enter your new filter's name here.
    d. Click the *save* button to save the filter. The user will see a message indicating that the filter was saved successfully, or that there was an error and the filter was not saved.
    e. If the user changes their mind, they can dismiss the modal without saving by clicking *close*, the *X* in the upper right corner, or anywhere outside the modal window.
4. How to See Saved Filters
    a. Click the *Load Filter* button displays a modal window that contains a list of all the user's saved filters. You can use this window just to view filters and are not required to load one.
    b. The user may close the modal by clicking *close*, or anywhere outside the modal window.
5. Loading a filter
    a. To load a filter, click the *Load Filter* button on the webpage banner.
    b. In the newly opened modal window, click the name of the filter you would like to load.
    c. The interface will display a message that the filter was successfully loaded or that an error occurred.
    d. Exit the modal window, and if successful, the loaded filter name should be visible on the left column of the screen, and the *Assignment Weight* and *Resonance Weight* forms updated to match the filter values.
    e. After loading a filter, see #11 to update the graph.
6. How to Create New Group
    a. Click the *Add group* button. A modal window will appear and provide an interface for saving a group.
    b. Enter a group name and click save. The user will be notified upon successful request or if an error occurred. Upon successful save, the user may now close the modal window.
7. How to Hide/Show Students and Groups on graph
    a. The left hand side of the webpage holds a column of students and groups.
    b. To hide a student/group, uncheck the box next to the name.
    c. To show a student/group, check the box next to the name.
    d. The graph will automatically update to reflect these changes.
8. How to Manually Refresh Cache

   a. Click the *Canvas Reload* button at the top of the page. This sends a request to the Canvas API Wrapper to recollect information from Canvas and recache the data.
   b. The user will be notified of an error or successful request.
   c. Once recaching is successful, the user may redraw the graph to see changes. See #10 for more details.
9. How to Redraw graph data
   a. Under *Graph Manipulation*, click the large button titled *Recalculate Student Resonance*. This sends the request to the data analysis pipeline, which may take some time depending on the size of the data set.
   b. You are free to navigate the page and inspect the current graph while waiting for the new graph to be ready.
   c. The page will automatically load the new graph when complete. Refreshing the page will not cancel the request to redraw the graph.
10. How to change Weights associated with assignments
   a. Under *Graph Manipulation*, enter new values into the *Assignment Weights* form.
   b. To see changes updated on the graph, see #11.
11. How to change Weights associated with resonance components
   a. Under *Graph Manipulation*, enter new values into the *Resonance Weights* form.
   b. To see changes updated on the graph, see #11.

Canvas API wrapper

1. How do you view the endpoints that exist?
   - To view the endpoints that are used to interact with the canvas api in our application, navigate to the swagger documentation at http://sddec21-19.ece.iastate.edu:30010/index.html
   - Alternatively, if you have set up your own instance of this application, navigate to the page: *http://<ADDRESS OF CLUSTER MACHINE>:30010/index.html* as the port the page is kept constant as we move across machines and only the hostname of the machine running your kubernetes cluster will change.
2. How do I set my bearer token?
   - On initial startup, refer to the *Setting Proper Access Token from Canvas* section of the *Deploy Application* process.
   - After initial setup, follow the *Updating Canvas Access Token* inside the *User Manual* section.
3. How do I set url parameters / see which ones are available?
   - The URL parameters that can be set will be listed as an input on the swagger doc.
   - There are also required URL parameters used for routing that will be marked red.
4. How do I make manual requests to the API?
   - 4.1. You can use the swagger doc and view the endpoints and hit those from postman.

- 4.2 The most straightforward way is to press try it out and enter the required fields as shown below.
- Step 1 press try it out

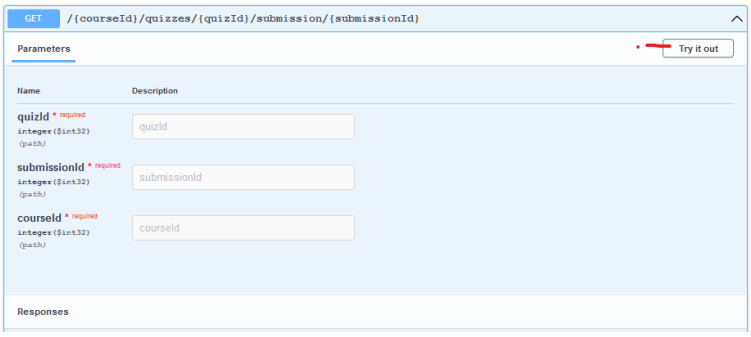


Figure 11.0 - Beginning Swagger Manual Requests

- Step 2. Enter required information and press execute. The JSON response will be given below:

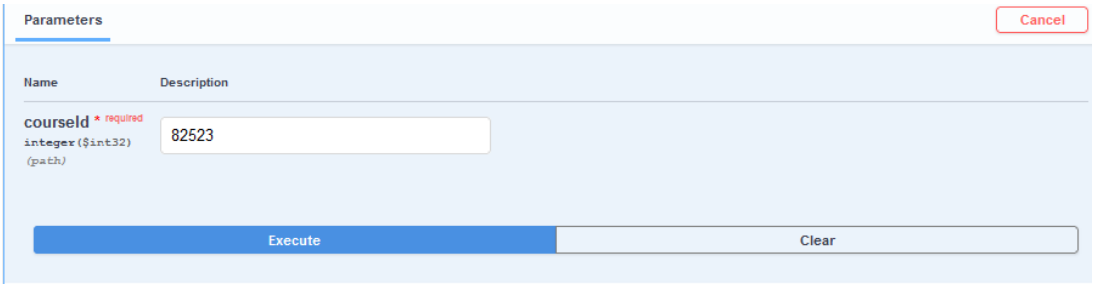


Figure 12.0 - Executing Swagger Manual Request

5. How do I see what data an endpoint will give back?
    - If it is a custom object the swagger doc will show the object documentation as shown below. If this isn't the case and it just says "string" you will need to call the endpoint to get back an object.

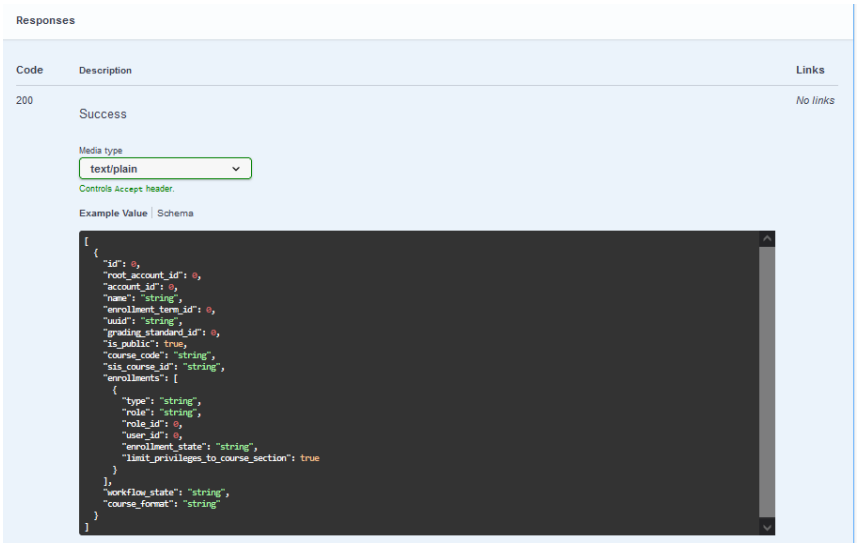


Figure 13.0 – Swagger Manual Response

Relevant Standards and Outside Resources
The endpoints that can be interacted with from the Canvas API and the documentation behind them can be found here: https://canvas.instructure.com/doc/api/.
The form for many common questions that are unanswered from the documentation can be found on the canvas api community board here:
https://community.canvaslms.com/t5/Canvas-Developers-Group/bd-p/developersforum-board.

For documentation on our API endpoints that exist we created Swagger documentation on our web application. This uses the OpenAPI standard 3.0 which is the standard for documentation on RESTful APIS.

Appendix II - Original / Alternate Designs

The first change that has come since our conception of this product was that of originally having the plan to use a python wrapper that already exists for the Canvas API. The reason why we chose to not take this approach were do to the following reasons

1. The python wrapper was mainly to wrap the functionality that already exists on the Canvas API. The issue with this is that the majority of the data we need can't simply just be retrieved from a Canvas endpoint unless the data is aggregated. Due to this and the fact that the team member working on the wrapper had little experience with python, it was not worth it to use an open source solution that already existed. There was too much customization that needed to be done, and it wasn't an out of the box solution for our team. Therefore we decided to create a custom .NET core application that did the operations we needed.
2. Another reason we did this was because if we used the code that this project used, it was licensed in a way that we would also need to make our project completely open source. This we determined was too risky due to the fact that we have little security knowledge as a team, and this would expose a lot of our endpoints and security to the outside world. Specifically Iowa State students.

Appendix III - Protobuf Specifications

Protobuf Diagrams

Clustering Endpoint

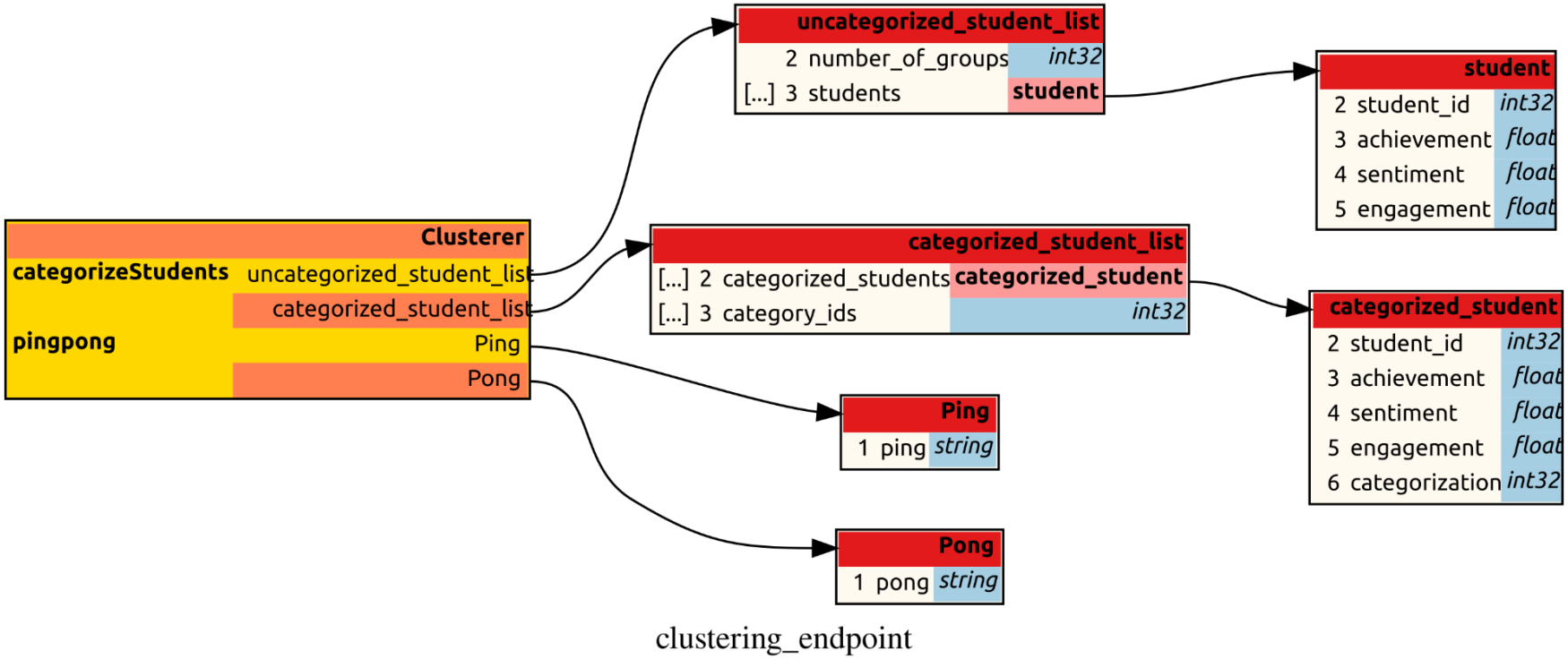
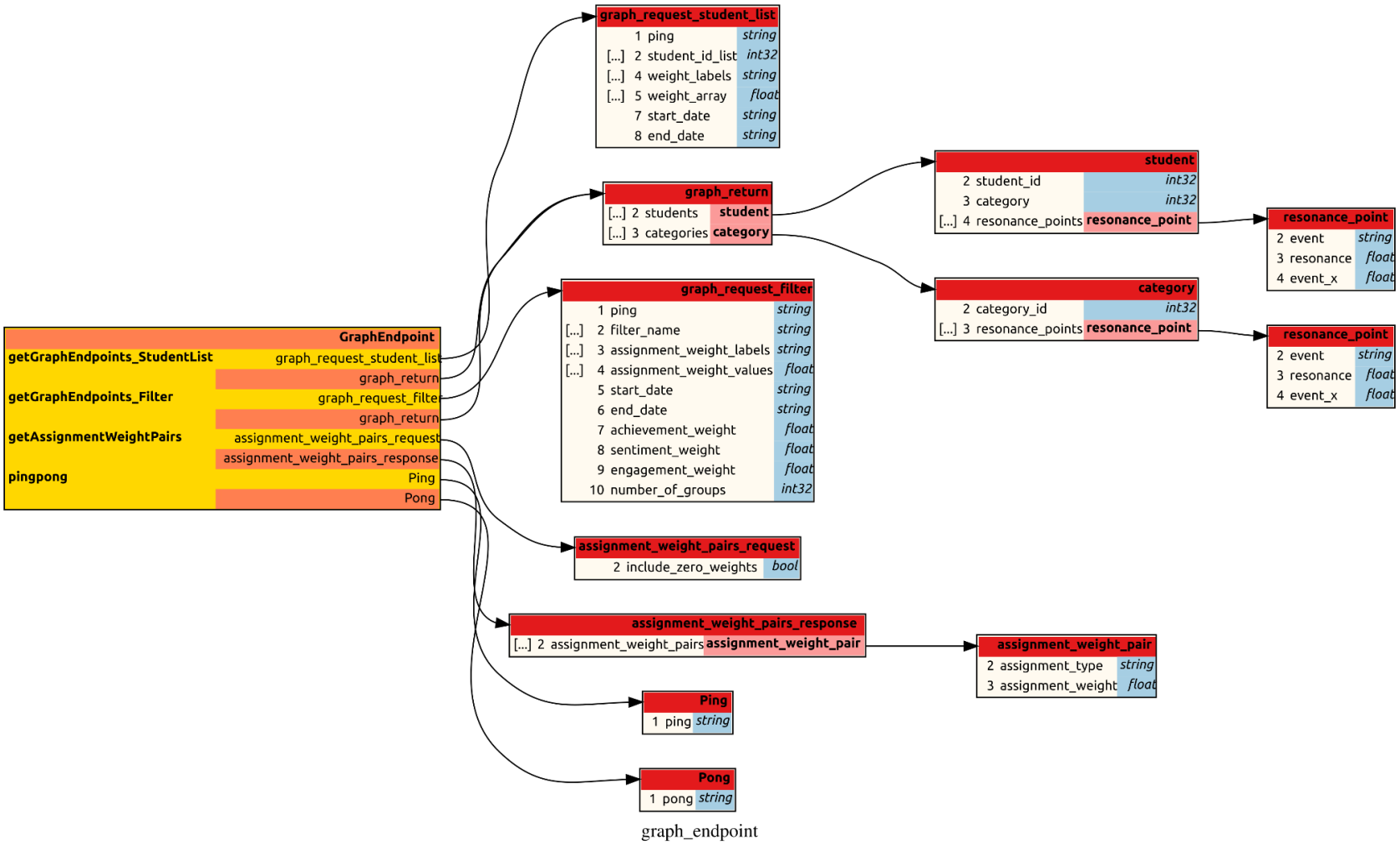


Figure 14.0 – Clustering Protobuf Diagram

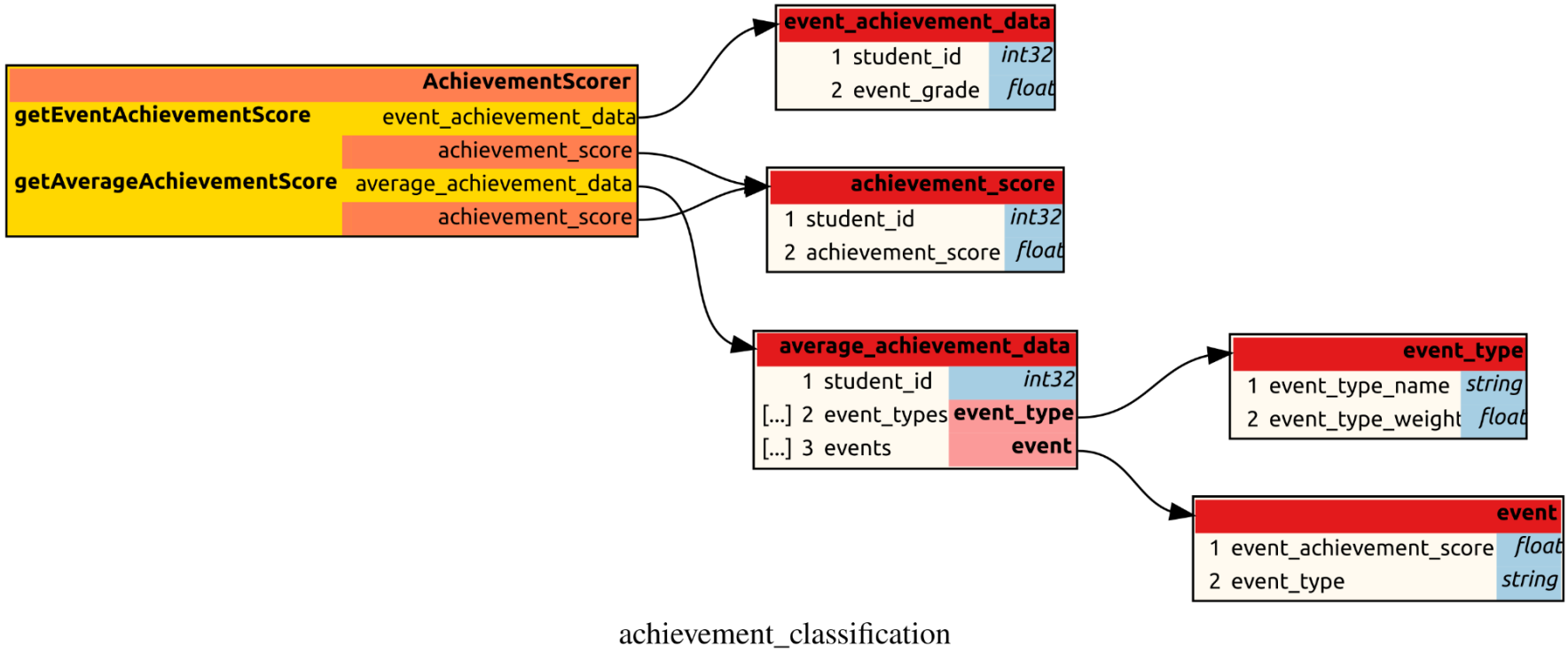
Graph Endpoint



graph_endpoint

Figure 15.0 – Graph Endpoint Protobuf Diagram

Achievement Scorer



achievement_classification

Figure 16.0 – Achievement Classification Protobuf Diagram

Sentiment Scorer

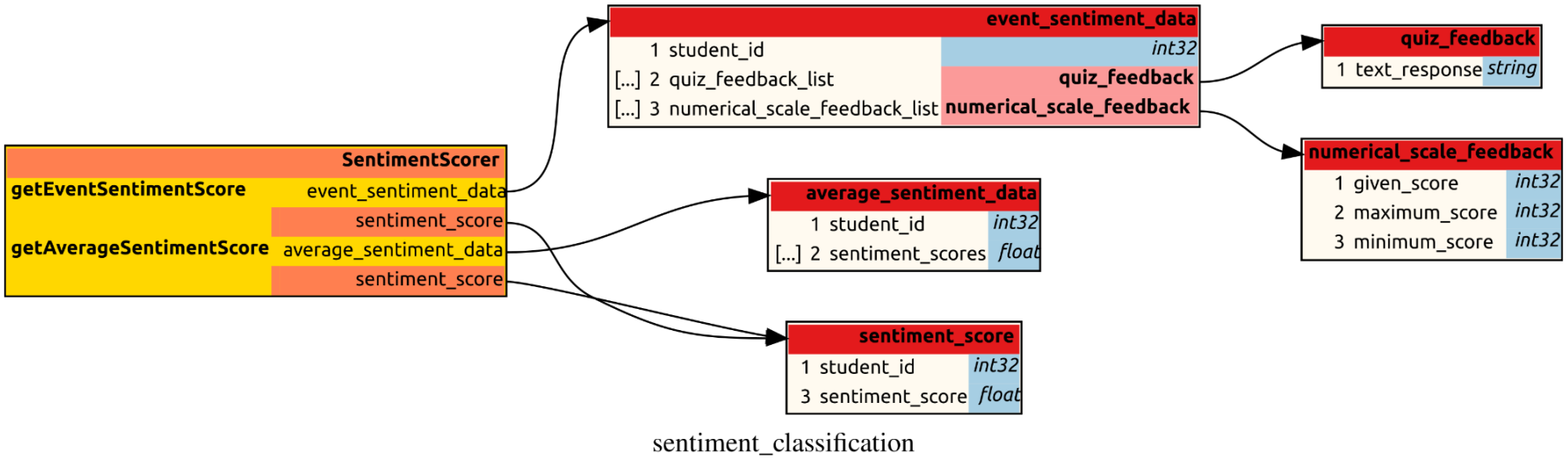


Figure 17.0 – Sentiment Classification Protobuf Diagram

Engagement Scorer

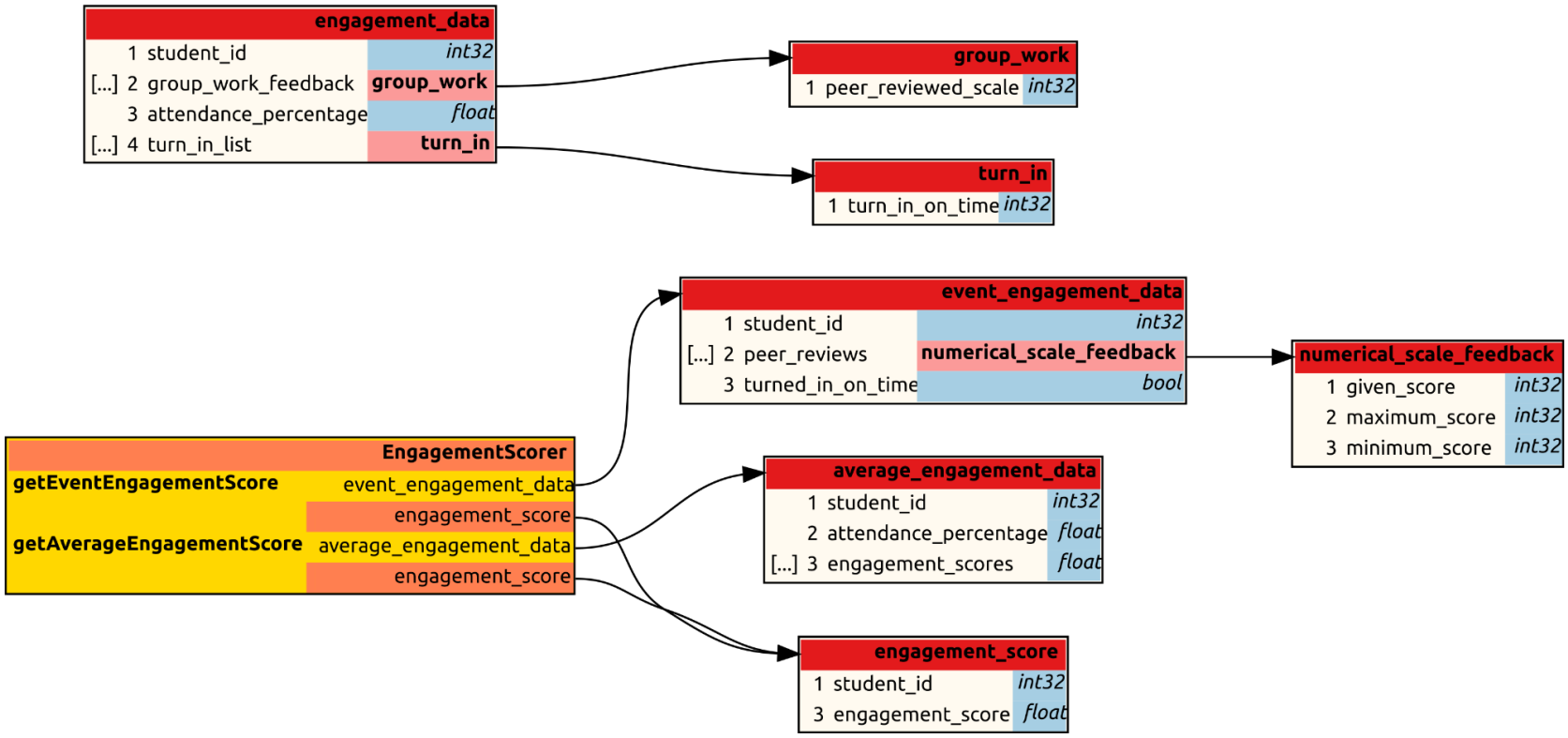


Figure 18.0 – Engagement Classification Protobuf Diagram

Resonance Scorer

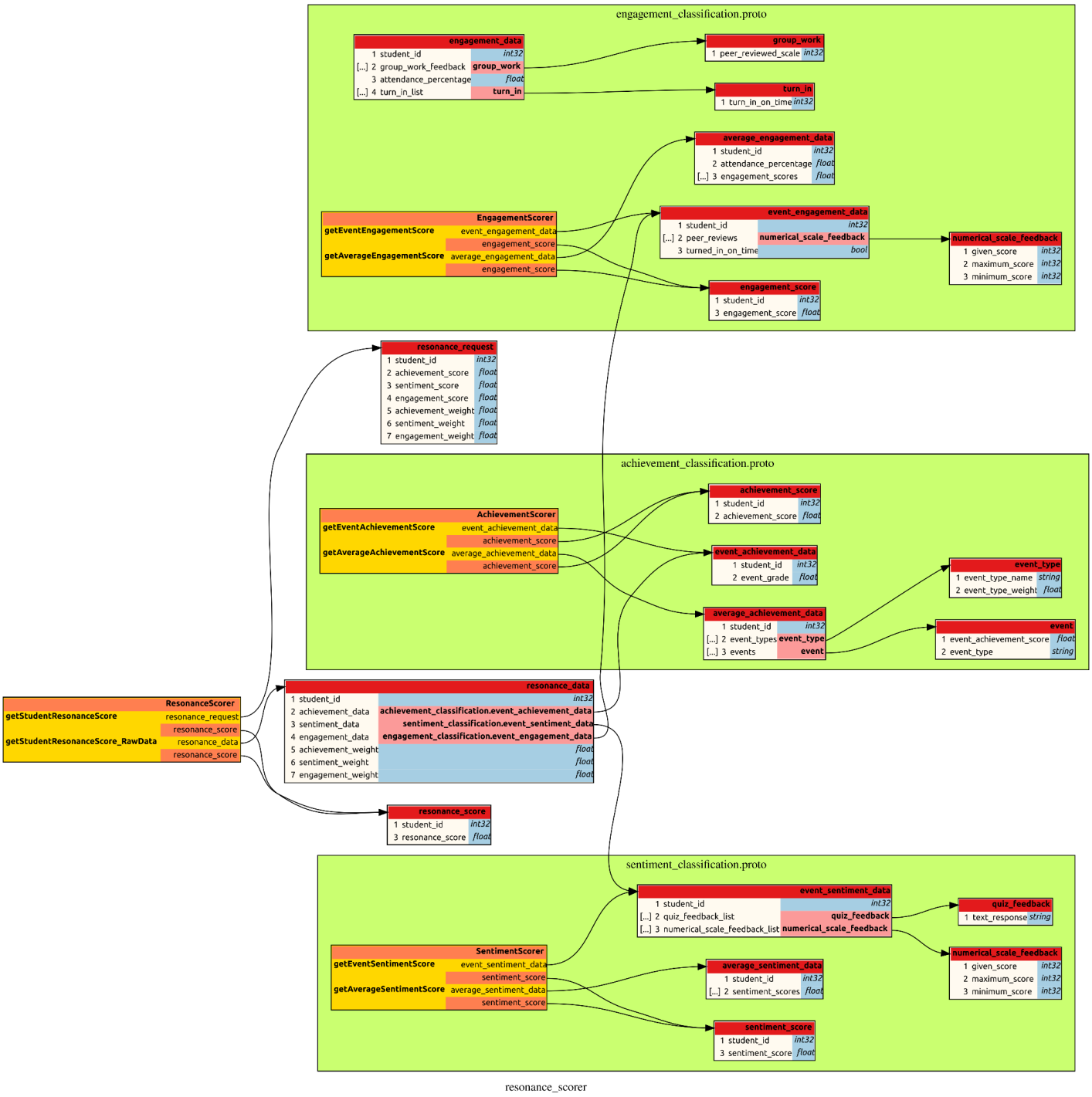


Figure 19.0 – Resonance Scorer Protobuf Diagram

Protobuf Definitions

Clustering Endpoint

```
syntax = "proto2";
package clustering_endpoint;

// SENTIMENT VALUE CALCULATION
message uncategorized_student_list{
  message student {
    optional int32 student_id = 2;
    optional float achievement = 3;
    optional float sentiment = 4;
    optional float engagement = 5;
  }
  optional int32 number_of_groups = 2;
  repeated student students = 3;
}

message categorized_student_list {
  message categorized_student {
    optional int32 student_id = 2;
    optional float achievement = 3;
    optional float sentiment = 4;
    optional float engagement = 5;
    optional int32 categorization = 6;
  }
  repeated categorized_student categorized_students = 2;
  repeated int32 category_ids = 3;
}

message Ping {
  optional string ping = 1;
}

message Pong {
  optional string pong = 1;
}

service Clusterer{
  // Obtains the persona classification given a score tuple
  rpc categorizeStudents(uncategorized_student_list) returns
(categorized_student_list) {}
  rpc pingpong(Ping) returns (Pong) {}
}
```

Graph Endpoint

```
syntax = "proto2";

package graph_endpoint;

// SENTIMENT VALUE CALCULATION
message graph_request_student_list {
  optional string ping = 1;
  repeated int32 student_id_list = 2;
  repeated string weight_labels = 4;
  repeated float weight_array = 5;
  optional string start_date = 7;
  optional string end_date = 8;
}

message graph_request_filter {
  optional string ping = 1;
  repeated string filter_name = 2;
  repeated string assignment_weight_labels = 3;
  repeated float assignment_weight_values = 4;
  optional string start_date = 5;
  optional string end_date = 6;
  optional float achievement_weight = 7;
  optional float sentiment_weight = 8;
  optional float engagement_weight = 9;
  optional int32 number_of_groups = 10;
}

message graph_return {
  message student {
    message resonance_point {
      optional string event = 2;
      optional float resonance = 3;
      optional float event_x = 4;
    }

    optional int32 student_id = 2;
    optional int32 category = 3;
    repeated resonance_point resonance_points = 4;
  }

  message category {
    message resonance_point {
      optional string event = 2;
```

```
      optional float resonance = 3;
      optional float event_x = 4;
    }

    optional int32 category_id = 2;
    repeated resonance_point resonance_points = 3;
  }

  repeated student students = 2;
  repeated category categories = 3;
}

message assignment_weight_pairs_request {
  optional bool include_zero_weights = 2;
}

message assignment_weight_pairs_response {
  message assignment_weight_pair {
    optional string assignment_type = 2;
    optional float assignment_weight = 3;
  }

  repeated assignment_weight_pair assignment_weight_pairs = 2;

}


message Ping {
  optional string ping = 1;
}

message Pong {
  optional string pong = 1;
}

service GraphEndpoint{
  // Obtains the persona classification given a score tuple
  rpc getGraphEndpoints_StudentList(graph_request_student_list)
returns (graph_return) {}
  rpc getGraphEndpoints_Filter(graph_request_filter) returns
(graph_return) {}
  rpc getAssignmentWeightPairs(assignment_weight_pairs_request)
returns (assignment_weight_pairs_response) {}
  rpc pingpong(Ping) returns (Pong) {}
}
```

Achievement Scorer

```
syntax = "proto2";

package achievement_classification;

// GRADE VALUE CALCULATION
message event_achievement_data {
  optional int32 student_id = 1;
  optional float event_grade = 2;
}

// GRADE VALUE CALCULATION
message average_achievement_data {
  message event_type {
    optional string event_type_name = 1;
    optional float event_type_weight = 2;
  }

  message event {
    optional float event_achievement_score = 1;
    optional string event_type = 2;
  }

  optional int32 student_id = 1;
  repeated event_type event_types = 2;
  repeated event events = 3;
}

message achievement_score {
  optional int32 student_id = 1;
  optional float achievement_score = 2;
}

service AchievementScorer {
  // Obtains the persona classification given a score tuple
  rpc getEventAchievementScore(event_achievement_data) returns
(achievement_score) {}
  rpc getAverageAchievementScore(average_achievement_data) returns
(achievement_score) {}
}
```

Sentiment Scorer

```
syntax = "proto2";

package sentiment_classification;

// SENTIMENT VALUE CALCULATION
message event_sentiment_data {
  message quiz_feedback {
    optional string text_response = 1;
  }

  message numerical_scale_feedback {
    optional int32 given_score = 1;
    optional int32 maximum_score = 2;
    optional int32 minimum_score = 3;
  }

  optional int32 student_id = 1;
  repeated quiz_feedback quiz_feedback_list = 2;
  repeated numerical_scale_feedback numerical_scale_feedback_list =
3;
}

message average_sentiment_data {
  optional int32 student_id = 1;
  repeated float sentiment_scores = 2;
}

message sentiment_score {
  optional int32 student_id = 1;
  optional float sentiment_score = 3;
}

service SentimentScorer {
  // Obtains the persona classification given a score tuple
  rpc getEventSentimentScore(event_sentiment_data) returns
(sentiment_score) {}
  rpc getAverageSentimentScore(average_sentiment_data) returns
(sentiment_score) {}
}
```

Engagement Scorer

```
syntax = "proto2";
package engagement_classification;

// ENGAGEMENT VALUE CALCULATION
message event_engagement_data {
  message numerical_scale_feedback {
    optional int32 given_score = 1;
    optional int32 maximum_score = 2;
    optional int32 minimum_score = 3;
  }

  optional int32 student_id = 1;
  repeated numerical_scale_feedback peer_reviews = 2;
  optional bool turned_in_on_time = 3;
}

message average_engagement_data {
  optional int32 student_id = 1;
  optional float attendance_percentage = 2;
  repeated float engagement_scores = 3;
}

message engagement_data {

  message group_work {
    optional int32 peer_reviewed_scale = 1;
  }

  message turn_in {
    optional int32 turn_in_on_time = 1;
  }

  optional int32 student_id = 1;
  repeated group_work group_work_feedback = 2;
  optional float attendance_percentage = 3;
  repeated turn_in turn_in_list = 4;
}

message engagement_score {
  optional int32 student_id = 1;
  optional float engagement_score = 3;
}
```

```
service EngagementScorer {
  // Obtains the engagement score given the necessary engagement data
  rpc getEventEngagementScore(event_engagement_data) returns
(engagement_score) {}
  rpc getAverageEngagementScore(average_engagement_data) returns
(engagement_score) {}
}
```

Resonance Scorer

```
syntax = "proto2";
package resonance_scorer;
import "achievement_classification.proto";
import "sentiment_classification.proto";
import "engagement_classification.proto";

// RESONANCE CALCULATION
message resonance_data {
  optional int32 student_id = 1;
  optional achievement_classification.event_achievement_data
achievement_data = 2;
  optional sentiment_classification.event_sentiment_data
sentiment_data = 3;
  optional engagement_classification.event_engagement_data
engagement_data = 4;

  optional float achievement_weight = 5;
  optional float sentiment_weight = 6;
  optional float engagement_weight = 7;
}

message resonance_request {
  optional int32 student_id = 1;
  optional float achievement_score = 2;
  optional float sentiment_score = 3;
  optional float engagement_score = 4;
  optional float achievement_weight = 5;
  optional float sentiment_weight = 6;
  optional float engagement_weight = 7;
}

message resonance_score {
  optional int32 student_id = 1;
  optional float resonance_score = 3;
}

service ResonanceScorer {
  // Obtains the resonance score given the student data
  rpc getStudentResonanceScore(resonance_request) returns
(resonance_score) {};
  rpc getStudentResonanceScore_RawData(resonance_data) returns
(resonance_score) {};
}
```