40295919
Connor Mallon

CSC3065 Cloud Computing

Assessment: Student Engagement Monitoring (QUBengage) Microservice App Submission

General Instructions:

- First, students must include a "Cover Page" with "Student Number" and "Name".
- Second, student must include acknowledgement for any use of generative AI in their reports and must include this academic integrity statement in the second page of their report – ***"I certify that that the submission is my own work, all sources are correctly attributed, and the contribution of any AI technologies is fully acknowledged."***
- In addition, students must include table of contents and list of figures with page numbers. The report must include references at the end, and the use of footnotes must be avoided.
- The report must be saved as <your student_number>.pdf, and the report shouldn't be more than 100 pages. Any report exceeding the 100-page maximum limit will be penalised.

Important Note: The "anything else" is related to Task H (that is the use of Generative AI Tools), and also, in case something didn't fit into implementation or testing.

40295919
Connor Mallon

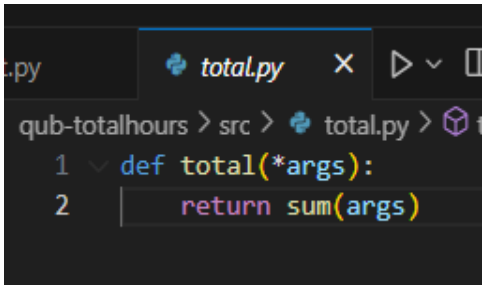
The structure of the report should be as follows:

Task A: Functions

Use the following structure for each of the functions you have implemented, if necessary, please feel free to include additional information such as code, screenshots, or anything else you want to maximise demonstrating how you meet/exceed marking criteria. Please note we will mark just the FOUR functions given here (i.e. it's better to have four more complete services than submit six less, so, only four will marked!).

- **FUNCTION ONE**

- Function (What Operation): **qub-engage-totalHours**
- Repo URL: https://repository.hal.davecutting.uk/ConnorM70/qubtotal_hours
- Live Service URL: http://localhost:83 (Containerised)
- Description of Implementation (language, methods, paradigm, etc):

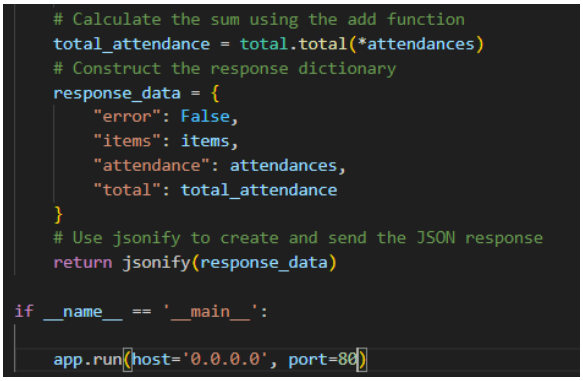


I have firstly created a simple python script which has a function called total. Where it returns the sum of any given argument inputs. Which will be called by my python flask application called app.py.

```
def addnumbers():
    items = []
    attendances = []
    # Retrieve items and attendances from the query string
    for i in range(1, 5):
        item = request.args.get(f'item_{i}')
        attendance = request.args.get(f'attendance_{i}')
        if not item or not attendance:
            return Response(
                response=json.dumps({
                    "error": True,
                    "message": f"Item or attendance for index {i} is missing."
                }),
                status=400,
                mimetype='application/json'

            )
        try:
            attendances.append(int(attendance))
        except ValueError:
            return Response(
                response=json.dumps({
                    "error": True,
                    "message": f"Invalid integer for attendance at index {i}."
                }),
                status=400,
                mimetype='application/json'
            )
        items.append(item)
    # Calculate the sum using the add function
    total_attendance = total.total(*attendances)
    # Construct the response dictionary
```

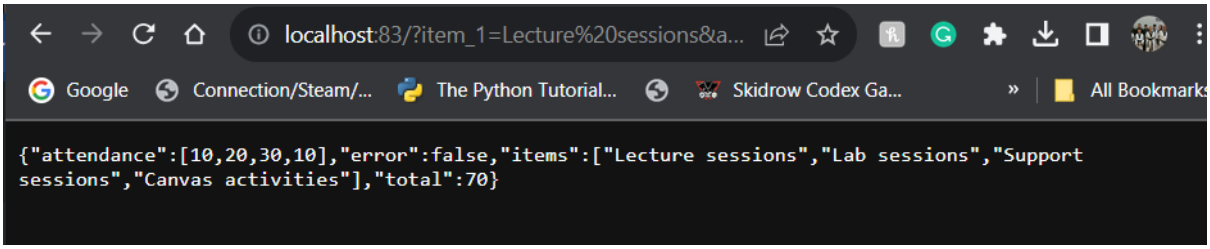
Within the app.py flask application,
It will first check for error inputs by the user and return a json error response which says "error: True message: Item or attendance for index _ is missing" on the web server if any incorrect input has been input by the user. However, if correct

it will then initialise a variable called total attendance and call the function total within total.py and return the correct json response as expected.
The code example is shown in the third screenshot.



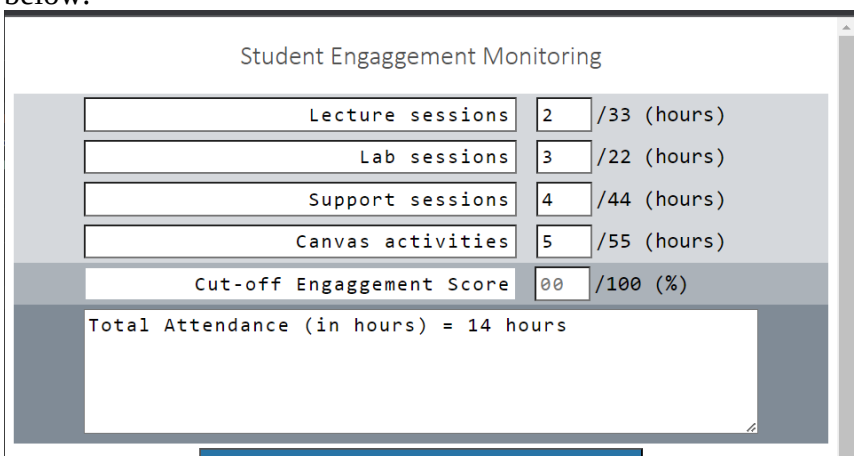
```
    # Calculate the sum using the add function
    total_attendance = total.total(*attendances)
    # Construct the response dictionary
    response_data = {
        "error": False,
        "items": items,
        "attendance": attendances,
        "total": total_attendance
    }
    # Use jsonify to create and send the JSON response
    return jsonify(response_data)

if __name__ == '__main__':

    app.run(host='0.0.0.0', port=80)
```

This will be printed to the user if a correct input will be validated.



It will look like this on the web server.
Shown here I have the localhost:83 port running and returns an example of the total hours calculated using the python flask application. The json response will be called and allowing to print to the web service.

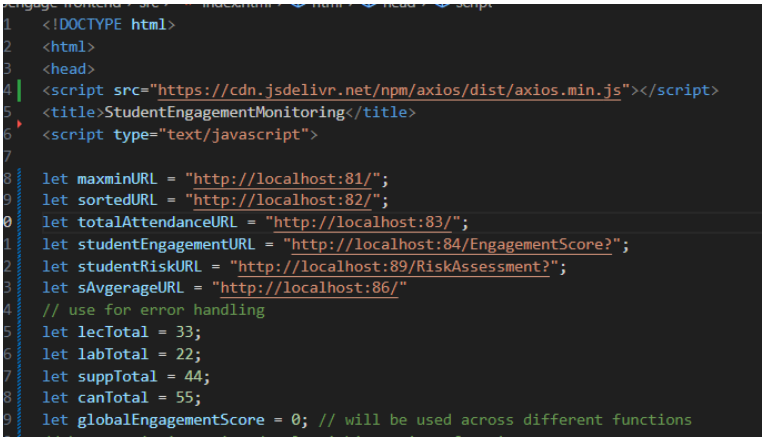
Where total hours will be calculated by summing up the attendances of Lectures, labs, Support, Canvas. Now that the web server is up and running,
It will now be accessible through the index.html front end section as shown in the image below.



Here you can see the physical frontend implementation which runs of http:localhost:90, where it is called within the backend implementation of the index.html as shown here.

The image shows the backend implementation of index.html, were, on line 10: totalHoursURL variable links to base http address of localhost:83.

This variable is then used within the getTotal() function that I have personally created.



```
<!DOCTYPE html>
<html>
<head>
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<title>StudentEngagementMonitoring</title>
<script type="text/javascript">

let maxminURL = "http://localhost:81/";
let sortedURL = "http://localhost:82/";
let totalAttendanceURL = "http://localhost:83/";
let studentEngagementURL = "http://localhost:84/EngagementScore?";
let studentRiskURL = "http://localhost:89/RiskAssessment?";
let sAvgerageURL = "http://localhost:86/"
// use for error handling
let lecTotal = 33;
let labTotal = 22;
let suppTotal = 44;
let canTotal = 55;
let globalEngagementScore = 0; // will be used across different functions
```

Here I have defined the functions which are called via localhost address.

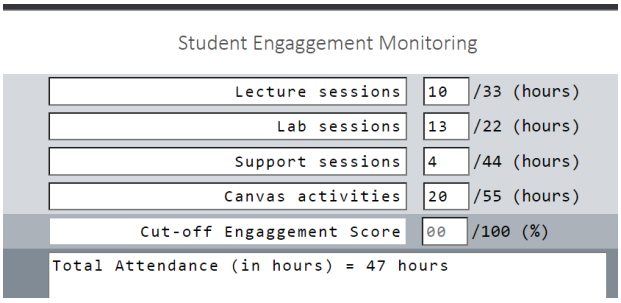
They will also be called and initialised within the respective functions they are used in. Additionally, I have used the total limit variables to ensure any error handling is also addressed in the case a user were to have an erroneous data input;

40295919
Connor Mallon

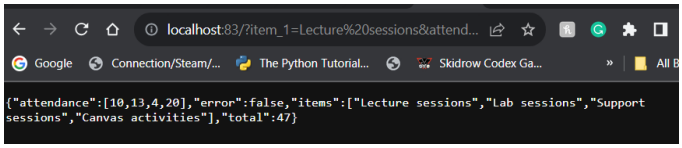
```
function getTotal()
{
    let item_1 = document.getElementById('item_1').value
    let item_2 = document.getElementById('item_2').value
    let item_3 = document.getElementById('item_3').value
    let item_4 = document.getElementById('item_4').value

    let attendance_1 = document.getElementById('attendance_1').value
    let attendance_2 = document.getElementById('attendance_2').value
    let attendance_3 = document.getElementById('attendance_3').value
    let attendance_4 = document.getElementById('attendance_4').value

    if (attendance_1 > lecTotal || attendance_2 > labTotal || attendance_3 > suppTotal || attendance_4 > canTotal) {
        alert("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
  let xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      var j = JSON.parse(this.response);
      let total = j.total;
      console.log('HELLO:' + j.total);
      displayTotal(total); // edit
    }
  };
  xhttp.open("GET",totalAttendanceURL +"?item_1=" + item_1 + "&attendance_1=" + attendance_1 + "&item_2=" + item_2 + "&attendance_2=" + attendance_2
  + "&item_3=" + item_3 + "&attendance_3=" + attendance_3 + "&item_4=" + item_4 + "&attendance_4=" + attendance_4);
  xhttp.send();
  return;
}

function getEngagementScore() {
```

Shown above is the getTotal() function which is the implementation on index.html, where it called the app.py flask application and prints the response to the localhost:90 front end.

This code which firstly gets the attendances of each user input from the lectures, labs, canvas activities, and support sessions. It will then error handle to ensure the user has not put in a value above the allowed total and pass the Json response from the web application of total hours. And display it within the localhost:90 in a nice, visible format, as shown in the first image. As displayed "Total Attendance (in hours) = 14". Dependant on the user input, the total would also increase/ decrease of course, we can see another example of this as shown:



Live Service URL example:

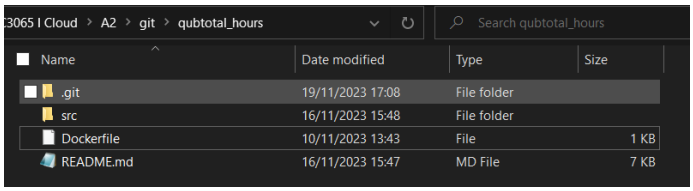


http://localhost:83/?item_1=Lecture%20sessions&attendance_1=10&item_2=Lab%20sessions&attendance_2=20&item_3=Support%20sessions&attendance_3=30&item_4=Canvas%20activities&attendance_4=20

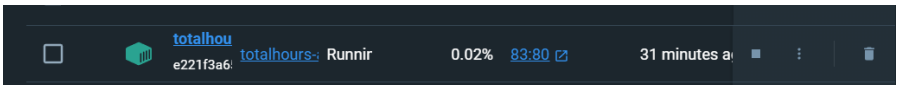
This application will now be accessible to the front-end application.

Next for containerisation the application, I created a Dockerfile which looks as follows:

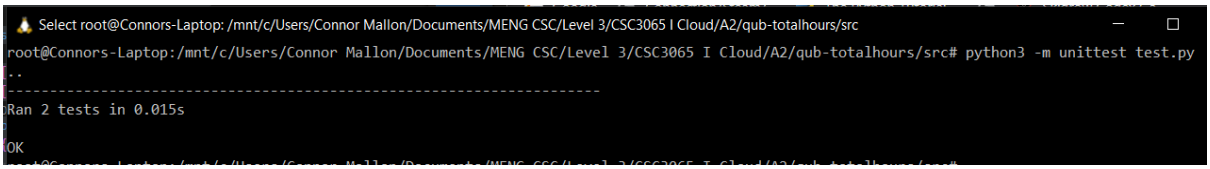
```
FROM ubuntu:18.04

RUN apt-get update -y  && apt-get install -y python3 python3-pip

COPY ./requirement.txt /app/requirement.txt

WORKDIR /app

RUN apt-get install -y python3-markupsafe

RUN pip3 install -r requirement.txt

COPY ./src /app

EXPOSE 5000

ENTRYPOINT ["python3"]

CMD ["app.py"]
```



This file is stored outside the src directory and in the root folder of qubtotal_hours.
Having this sort of code structure within the qubtotal_hours allow for a simple, straightforward structure which does not complicate the code.



The Docker container here will allow us to deploy the qubtotal_hours python code and connect it to the front-end application. This is shown in the live service URL example as above.
Testing case:



I have implemented the test using Python once again, I created two functions:
Description of Testing:

```
import unittest
from app import *

class AddNumbersTestCase(unittest.TestCase):
    def setUp(self) :
        self.app = app.test_client()
        self.app.testing = True

    def test_add_numbers(self):
        response = self.app.get(
        '/?item_1=Lecture%20sessions&attendance_1=10'
        '&item_2=Lab%20sessions&attendance_2=20'
        '&item_3=Support%20sessions&attendance_3=30'
        '&item_4=Canvas%20activities&attendance_4=10')
        # Check if the response status code is 200
        self.assertEqual(response.status_code, 200)

        # Load the response data
        data = json.loads(response.data)
        # Check the response content
        self.assertEqual(data['error'], False)
        print("No error in response.")
        self.assertEqual(data['items'],
                         ['Lecture sessions', 'Lab sessions', 'Support sessions', 'Canvas activities'])
        print("Labels are correct.")
        self.assertEqual(data['attendance'], [10, 20, 30, 10])
        print("Attendance values are correct.")
        self.assertEqual(data['total'], 70) # Assuming the total.total function sums the numbers
        print("Total value is correct.")
```

```
    def test_missing_parameters(self):
        # Missing parameters in the query string
        response = self.app.get('/?item_1=book&attendance_1=10')

        # Check if the response status code is 400
        self.assertEqual(response.status_code, 400)

        # Load the response data
        data = json.loads(response.data)

        # Check the error message
        self.assertEqual(data['error'], True)

        self.assertEqual(data['message'], "Item or attendance for index 2 is missing.")


if __name__ == '__main__':
    unittest.main()
```

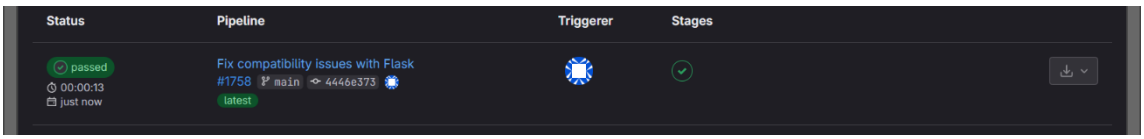
I have created two test cases, one test case to ensure it is equal, and one to ensure the results are not equal.

Shown below is the testing case code which I have completed in python.

I have a test_add_number() function and test_missing parameters() function which allow to test the user input of the given elements.
This will be displayed in a web service application as shown in the previous screenshot at the top of the page. Shown in “test_add_numbers()

I have used an example correct response and additionally ensured that the assertEquals() response will be respected.

Below is the Working test pipeline shown. We can see the CI has passed with the following test cases on Gitlab. This is also built using a .git-ci.yml file – shown in the next page.



This .gitlab.yml file only has one stage which is the test stage, which allows to pass the CI pipeline test.
It will firstly install (flask), (Jinja) and (flask cors) which are essential modules that are needed for the CI pipelining.

```yaml
image: python:3.8-slim

stages:
  - test

before_script:
  - python -m venv venv
  - source venv/bin/activate
  - pip install -r requirements.txt

test_app:
  stage: test
  script:
    - export PYTHONPATH=./src:$PYTHONPATH
    - python -m unittest discover -s src -p 'test*.py'
```

- ## FUNCTION TWO

- Function (What Operation):
  qub-engagementScore
- Repository URL: https://repository.hal.davecutting.uk/ConnorM70/qub-engagementscore
- Live Service URL: http://localhost:84 (Containerised)
- Description of Implementation (language, methods, paradigm, etc):

Firstly, I created a C# file with used the API.Controller to convert to a web service. Which is called “stEngagement.cs”
I have initialised variables which are needed and used within index.html, These variables will be used in the calculation of the engagement Score.

```
using Microsoft.AspNetCore.Mvc;

namespace StudentEngagementApi.Controllers
{
    [ApiController]
    [Route("[controller]")]
    0 references
    public class EngagementScoreController : ControllerBase
    {
        // Fixed totals for each category
        2 references
        private const double LectureTotal = 33;
        2 references
        private const double LabTotal = 22;
        2 references
        private const double SupportTotal = 44;
        2 references
        private const double CanvasTotal = 55;

        // Fixed weights for each category
        1 reference
        private const double LectureWeight = 0.3; // Weight for lecture sessions
        1 reference
        private const double LabWeight = 0.4; // Weight for lab sessions
        1 reference
        private const double SupportWeight = 0.15; // Weight for support sessions
        1 reference
        private const double CanvasWeight = 0.15; // Weight for canvas activities
```

I have used the formula as below:

II. **Student Engagement Score** – calculate the engagement score using the following equation inside a function:

$$StudentEngagementScore = \frac{lec * lec_w}{lec_{total}} + \frac{lab * lab_w}{lab_{total}} + \frac{supp * supp_w}{Supp_{total}} + \frac{can * can_w}{can_{total}}$$

My code will call the CalcuateEngagementScore() function which is the calculation of the given formula shown in the Assignment brief.

40295919
Connor Mallon

```
// GET action method to calculate the engagement score
[HttpGet]
0 references
public ActionResult<double> GetEngagementScore(
    [FromQuery] double lec, [FromQuery] double lab,
    [FromQuery] double supp, [FromQuery] double can)
{
    // Validate that the attendance values do not exceed the total values
    if (lec > LectureTotal || lab > LabTotal || supp > SupportTotal || can > CanvasTotal)
    {
        return BadRequest("Attendance values must not exceed total values.");
    }

    // Calculate the engagement score
    double engagementScore = CalculateEngagementScore(lec, lab, supp, can);

    // Cap the engagement score at 1.0 (100%)
    engagementScore = Math.Min(engagementScore, 1.0);

    // Return the score as a fraction of 1
    return Ok(engagementScore);
}

1 reference
private double CalculateEngagementScore(
    double lec, double lab, double supp, double can)
{
    // Calculate the weighted score for each category
    double lecScore = (lec / LectureTotal) * LectureWeight;
    double labScore = (lab / LabTotal) * LabWeight;
    double suppScore = (supp / SupportTotal) * SupportWeight;
    double canScore = (can / CanvasTotal) * CanvasWeight;

    // Sum the weighted scores to get the total engagement score
    double totalScore = lecScore + labScore + suppScore + canScore;

    // The score is already a fraction of 1, so we don't need to divide by 100
    return totalScore;
}
}
}
```

Firstly, I have an error handling check to ensure any input above the given total will return with a error message “Attendance must not exceed total values” This will stop any values being entered, locally on the web page, and any inputs which fall outside the total range will not be accepted.

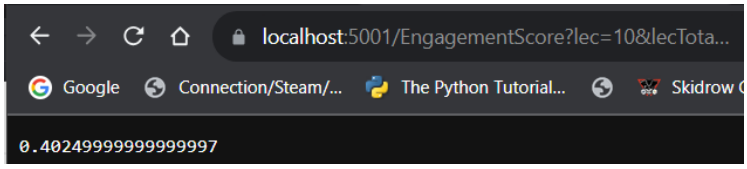
The CalculateEngagementScore() will take in the input lectures, labs, support, canvas variables which are assigned on the index.html, or this can be done manually as the as the screenshot here shows the result output on the working web service.

Specific to C# I need to create a stEngage.csproj file which looks as follows:
This uses the required SDK version 6 which I have installed, additionally uses the TestAdapter, used for testing, as well as the NUnit Version to execute.

qub-engagement > stEngage.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <!-- Test-specific package references -->
    <PackageReference Include="NUnit" Version="3.14.0" />
    <PackageReference Include="NUnit3TestAdapter" Version="4.5.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc.Testing" Version="6.0.0" />
  </ItemGroup>

  <ItemGroup>
    <!-- Reference to the main project -->
    <ProjectReference Include="..\src\stEngage.csproj" />
  </ItemGroup>

</Project>
```

Moving back to the web service, we can run it locally will look like this as follows:
Using the following correct URL example:



https://localhost:5001/EngagementScore?lec=10&lecTotal=20&lecW=0.3&lab=15&labTotal=20&labW=0.4&supp=5&suppTotal=20&suppW=0.15&can=8&canTotal=10&canW=0.15.

Shown here is the C# console working response 200. So, we know everything is working as expected for the engagement Score.

```
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished HTTP/2 GET https://localhost:5001/EngagementScore?lec=10
&lecTotal=20&lecW=0.3&lab=15&labTotal=20&labW=0.4&supp=5&suppTotal=20&suppW=0.1
5&can=8&canTotal=10&canW=0.15 - - - 200 - application/json;+charset=utf-8 177.9
634ms
```

The total inputs have been ensured that the total lecture., lab, support amount must add up to 100 as well, in the case it does not remain consistent, this following error will prompt to the user if the total attendance exceeds 100:

```
Attendance values must not exceed total values.
```

The user must then change the lecture, lab, support, or canvas attendance hours to get a valid response.

Since the program is now running locally, we can then transfer to the frontend code of the index.html, Firstly creating the getEngagementScore() function.

```
function getEngagementScore() {

    // Actual attendance values from the input fields
    let lec = parseFloat(document.getElementById('attendance_1').value);
    let lab = parseFloat(document.getElementById('attendance_2').value);
    let supp = parseFloat(document.getElementById('attendance_3').value);
    let can = parseFloat(document.getElementById('attendance_4').value);

    // Check if any of the parsed numbers are NaN and set them to 0 if true
    lec = isNaN(lec) ? 0 : lec;
    lab = isNaN(lab) ? 0 : lab;
    supp = isNaN(supp) ? 0 : supp;
    can = isNaN(can) ? 0 : can;
    let lecW = 0.3;
    let labW = 0.4;
    let suppW = 0.15;
    let canW = 0.15;

    // Error handling for input values greater than the totals
    if (lec > lecTotal || lab > labTotal || supp > suppTotal || can > canTotal) {
        alert("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
```

```
    // Use XMLHttpRequest to send the GET request
    let queryString = `lec=${lec}&lecTotal=${lecTotal}&lecW=${lecW}`
    '&lab=${lab}&labTotal=${labTotal}&labW=${labW}&supp=${supp}&suppTotal=
    ${suppTotal}&suppW=${suppW}&can=${can}&canTotal=${canTotal}&canW=${canW}`;
    let xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            let engagementScore = parseFloat(this.responseText);
            if (!isNaN(engagementScore)) {
                globalEngagementScore = engagementScore; // assign here
                displayEngagementScore(engagementScore);
            } else {
                console.error('Invalid response from the server:', this.responseText);
            }
        } else if (this.readyState == 4) {
            console.error('Error: ' + this.statusText);
        }
    };
    xhttp.open("GET", studentEngagementURL + queryString, true);
    xhttp.send();
    return;
}
```

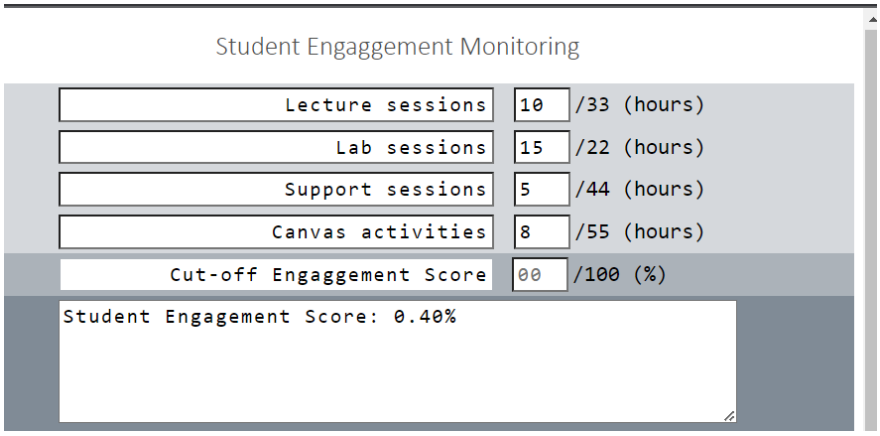
The function gets all the user inputs, and checks if the values are ‘NaN’ or 0, which will assign the respective variable to 0. It will additionally check if the input is larger than the total amount, this returns a window prompt to the user saying “input values cannot be higher than the total hours” which prevents the user from continuing. They must input a value, lower than the given total amount. In the second image, we can catch the URL link and get a respective Json response, ensuring it will be valid and return a response of 200. If so, this allows us to proceed to the displayEngagementScore() else return an error response.

In the displayEngagementScore() function, it looks as follows:

```
function displayEngagementScore(engagementScore) {
    let scoreToDisplay = (engagementScore).toFixed(2);
    document.getElementById('output-text').value = 'Student Engagement Score: ' + scoreToDisplay + '%';
}
```

This passes the engagement score value which is calculated prior and prints this output to the user “Student Engagement Score: (score) %” which is calculated to two decimal places. The output example is shown in the front-end Student Engagement example above.

The result will look like such:



This is used from the displayEngagement() function as shown in the screenshot above.

We can see that the front-end web service is working so we can move onto the testing segment.

40295919
Connor Mallon

Description of Testing:

```
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
Expected Score: 0.6659090909090909
Actual Score: 0.6659090909090909

Expected Score: 0.4870454545454546
Actual Score: 0.327954545454545


Passed!  - Failed:     0, Passed:     2, Skipped:     0, Total:     2, Duration: 28 ms - testProject.Tests.dll (net8.0)
```

I have created two functions, one to test if the engagement score is equal, shown in the top example, and the other to check if the engagement score is not equal, shown in the screenshot here, we can see both tests have passed locally.
The following code looks as follows.

```
using System;
using NUnit.Framework;
using StudentEngagementApi.Controllers;
using Microsoft.AspNetCore.Mvc;

namespace StudentEngagementApi.Tests
{
    [TestFixture]
    0 references
    public class EngagementScoreControllerTests
    {
        #pragma warning disable CS8618 // Non-nullable field is uninitialized. Consider declaring as nullable.
        3 references
        private EngagementScoreController _controller; // No need for the nullable '?'
        #pragma warning restore CS8618
        [SetUp]
        0 references
        public void SetUp()
        {
            _controller = new EngagementScoreController(); // This ensures _controller is never null in your tests
        }
```

```
[Test]
0 references
public void GetEngagementScore_WithValidAttendanceValues_ReturnsCorrectScore() {
    // Arrange
    double lectureAttendance = 20;
    double labAttendance = 15;
    double supportAttendance = 30;
    double canvasAttendance = 40;

    // Act
    double expectedScore = CalculateExpectedScore(lectureAttendance, labAttendance, supportAttendance, canvasAttendance);
    ActionResult<double> result = _controller.GetEngagementScore(lectureAttendance, labAttendance, supportAttendance, canvasAttendance);

    // Assert
    Assert.IsNotNull(result.Result, "Expected an ActionResult.");
    var okResult = result.Result as OkObjectResult;
    Assert.IsNotNull(okResult, "Expected an OkObjectResult.");
    var score = okResult.Value as double?;

    Assert.IsTrue(score.HasValue, "Expected score value");
    Assert.AreEqual(expectedScore, score.Value, "The calculated score does match the expected score.");

    // Print to console
    TestContext.WriteLine($"Expected Score: {expectedScore}");
    TestContext.WriteLine($"Actual Score: {score.Value}");
```

The three following code snippets presents four total functions used which help validate the student engagement score that will be implemented to the front-end service user. Validation checking such as the setup() function are critical for the C# program to begin in the first place, where it needs a given controller instance to return a given value from the web service. The given test cases complete a assertEquals to ensure the expected result matches with the actual result. Additionally, I have created another case where the actual result is intentionally not equal to the expected result, assuming a AreNotEquals() function. Shown on the bottom right function, within - getEngagementScoreWithInvalidValues ()

```
[Test]
0 references
public void GetEngagementScoreWithInvalidValues() {
    // Arrange
    double lectureAttendance = 6;
    double labAttendance = 20;
    double supportAttendance = 17;
    double canvasAttendance = 4;
    // wrong input values
    double wrongLecture =2;
    double wrongLab = 14;
    double wrongSupp = 13;
    double wrongCan = 4;
    // Act
    double expectedScore = CalculateExpectedScore(lectureAttendance, labAttendance, supportAttendance, canvasAttendance);
    ActionResult<double> result = _controller.GetEngagementScore(wrongLecture, wrongLab, wrongSupp, wrongCan);

    // Assert
    Assert.IsNotNull(result.Result, "Expected an ActionResult but was null.");
    var okResult = result.Result as OkObjectResult;
    Assert.IsNotNull(okResult, "Expected an OkObjectResult but was null.");
    var score = okResult.Value as double?;

    Assert.IsTrue(score.HasValue, "Expected a score value.");
    Assert.AreNotEqual(expectedScore, score.Value, "The calculated score does not match the expected score.");

    // Print to console
    TestContext.WriteLine($"Expected Score: {expectedScore}");
    TestContext.WriteLine($"Actual Score: {score.Value}");

}
```

The test case result example is shown at the top of the page. It additionally shows two passing test cases.

```
}
    2 references
    private double CalculateExpectedScore(double lec, double lab, double supp, double can)
    {
        // Mock the calculation logic here or provide a static expected score
        double lecScore = (lec / 33) * 0.3;
        double labScore = (lab / 22) * 0.4;
        double suppScore = (supp / 44) * 0.15;
        double canScore = (can / 55) * 0.15;
        double totalScore = lecScore + labScore + suppScore + canScore;
        // Cap the total score at 1.0 as the method does
        return Math.Min(totalScore, 1.0);
    }
}
```

Next, since the local test has passed, we can push this onto GitLab and pipeline the project.

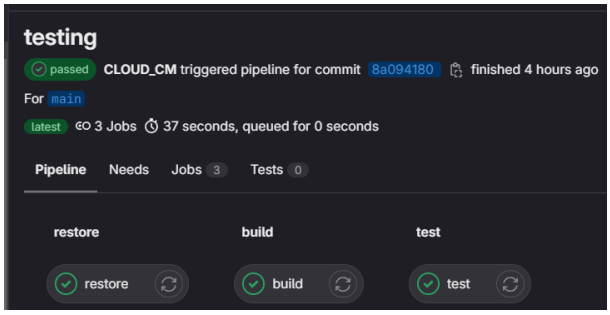
Firstly, by creating a .gitlab-ci.yml file as following:

40295919
Connor Mallon

```yaml
stages:
  - restore
  - build
  - test

variables:
  ENGAGEMENTSCORE_DIR: "src/"
  ENGAGEMENTSCORE_PROJECT_NAME: "stEngage"
  ENGAGEMENTSCORE_TEST_DIR: "Tests/"
  ENGAGEMENTSCORE_TEST_PROJECT_NAME: "testProject.Tests"

image: mcr.microsoft.com/dotnet/sdk:6.0

before_script:
  - 'dotnet nuget locals all --clear'

restore:
  stage: restore
  script:
    - 'echo "Restoring packages for $ENGAGEMENTSCORE_PROJECT_NAME..."'
    - 'dotnet restore $ENGAGEMENTSCORE_DIR$ENGAGEMENTSCORE_PROJECT_NAME.csproj --verbosity detailed'
    - 'echo "Restoring packages for $ENGAGEMENTSCORE_TEST_PROJECT_NAME..."'
    - 'dotnet restore $ENGAGEMENTSCORE_TEST_DIR$ENGAGEMENTSCORE_TEST_PROJECT_NAME.csproj --verbosity detailed'
  cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths:
      - .nuget/packages
      - $ENGAGEMENTSCORE_DIR**/obj/
      - $ENGAGEMENTSCORE_TEST_DIR**/obj/
```

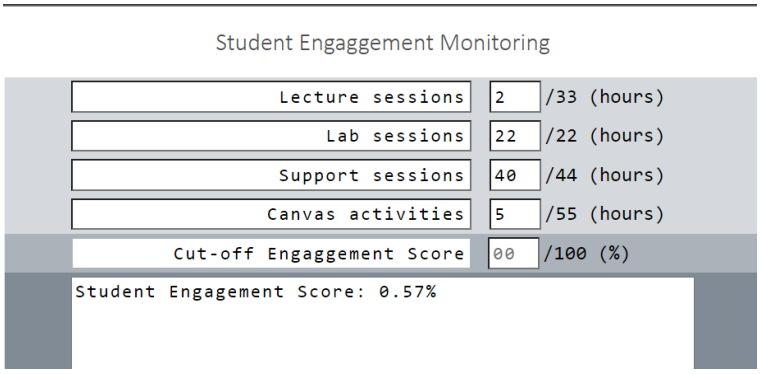
```yaml
build:
  stage: build
  script:
    - 'dotnet build $ENGAGEMENTSCORE_DIR$ENGAGEMENTSCORE_PROJECT_NAME.csproj --no-restore --configuration Release'
  artifacts:
    paths:
      - $ENGAGEMENTSCORE_DIR/bin/Release/
      - $ENGAGEMENTSCORE_TEST_DIR/bin/Release/
    expire_in: 1 hour
  only:
    - main

test:
  stage: test
  script:
    - 'dotnet test $ENGAGEMENTSCORE_TEST_DIR$ENGAGEMENTSCORE_TEST_PROJECT_NAME.csproj --no-build --verbosity normal'
  artifacts:
    when: always
    paths:
      - $ENGAGEMENTSCORE_TEST_DIR/bin/Release/
    reports:
      junit:
        - $ENGAGEMENTSCORE_TEST_DIR/TestResults/*.xml
    expire_in: 1 hour
  only:
    - main
```

There are three stages in this pipeline, restore, build, and test. Restore and Build are essential for the C# code to be up and running. After building the project. We can execute the program normally by dotnet run, or test by dotnet test. I have needed to connect the stEngage.csproj file to the test.cs to build and test the project properly.



And this is the pipeline for
the three stages: restore, build, and test.
We can see here, we have successfully passed the pipeline test.

We can see the working result on the localhost:90 page.
By inputting each session amount, we get the respective engagement score.



The front-end code has been changed to allow the engagement score to be displayed.
As follows within the index.html.

```dockerfile
# Use Microsoft's official .NET 6 SDK image for building the app.
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build-env
WORKDIR /app

# Copy csproj and restore as distinct layers.
COPY src/stEngage.csproj ./
RUN dotnet restore

# Copy everything else from the src directory and build.
COPY src/ ./
RUN dotnet publish -c Release -o out

# Build runtime image using the official .NET 6 runtime.
FROM mcr.microsoft.com/dotnet/aspnet:6.0
WORKDIR /app
COPY --from=build-env /app/out .
EXPOSE 80

ENTRYPOINT ["dotnet", "stEngage.dll"]
```

I have also created the following Dockerfile to port this into a container, to be used for the web service.

It will focus on the stEngage.dll application, using container port 80.

The process consists of a dotnet restore, and a dotnet publish.

This Dockerfile is a lot more complex in comparison to the python total hours application.

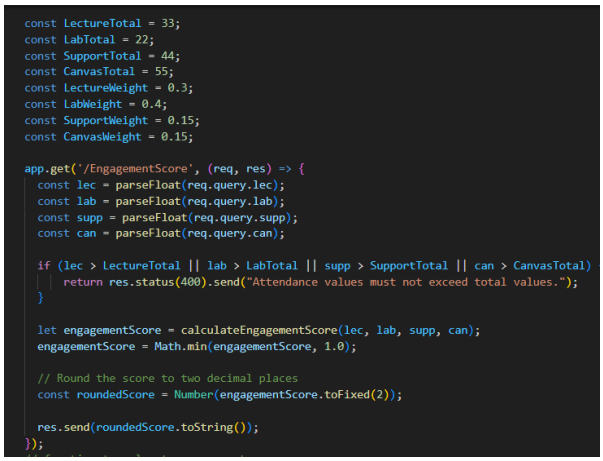
40295919
Connor Mallon

FUNCTION THREE

- Function (What Operation): **qub-engagementScore-failure**
- Repository URL: https://repository.hal.davecutting.uk/ConnorM70/qubengagement-failure
- Live Service URL: http://localhost:5000/RiskAssessment?engagementScore=0.44&cutoff=0.5

I have created a Node.js file which is called 'totalRiskApp.js' which looks as follows:'

```
const path = require('path');
const express = require('express');
const cors = require('cors');
const app = express();
const port = 89;  // run test on port 89
app.use(cors());
module.exports = app;

app.get('/', (req, res) => {
  const indexPath = path.join('/mnt/c', 'Users', 'Connor Mallon',
  'Documents', 'MENG CSC', 'Level 3', 'CSC3065 I Cloud', 'A2',
  'qubengage-frontend', 'src', 'index.html');

  res.sendFile(indexPath, (err) => {
    if (err) {
      console.error('Error sending index.html:', err);
      res.status(500).send('Internal Server Error');
    }
  });
});
app.get('/health', (req, res) => {
  res.send('Server is running!');
});
```



```
const LectureTotal = 33;
const LabTotal = 22;
const SupportTotal = 44;
const CanvasTotal = 55;
const LectureWeight = 0.3;
const LabWeight = 0.4;
const SupportWeight = 0.15;
const CanvasWeight = 0.15;

app.get('/EngagementScore', (req, res) => {
  const lec = parseFloat(req.query.lec);
  const lab = parseFloat(req.query.lab);
  const supp = parseFloat(req.query.supp);
  const can = parseFloat(req.query.can);

  if (lec > LectureTotal || lab > LabTotal || supp > SupportTotal || can > CanvasTotal)
      return res.status(400).send("Attendance values must not exceed total values.");
  }

  let engagementScore = calculateEngagementScore(lec, lab, supp, can);
  engagementScore = Math.min(engagementScore, 1.0);

  // Round the score to two decimal places
  const roundedScore = Number(engagementScore.toFixed(2));

  res.send(roundedScore.toString());
});
```

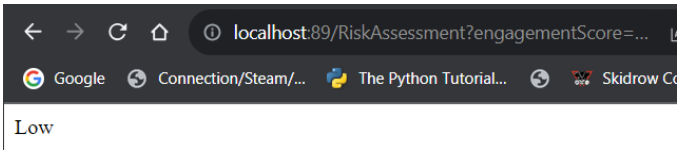
```
// function to calcuate engagement score
function calculateEngagementScore(lec, lab, supp, can) {
  const lecScore = (lec / LectureTotal) * LectureWeight;
  const labScore = (lab / LabTotal) * LabWeight;
  const suppScore = (supp / SupportTotal) * SupportWeight;
  const canScore = (can / CanvasTotal) * CanvasWeight;
  return lecScore + labScore + suppScore + canScore;
}
  // check url for /RiskAssessment
app.get('/RiskAssessment', (req, res) => {
  const engagementScore = parseFloat(req.query.engagementScore);
  const cutoff = parseFloat(req.query.cutoff);

if(isNaN(engagementScore) || isNaN(cutoff) || engagementScore < 0
  || engagementScore > 1 || cutoff < 0 || cutoff > 100) {
    return res.status(400).send("Invalid engagement score or cutoff values.");
  }

  const risk = calculateRisk(engagementScore, cutoff);
  res.send(risk);
});

function calculateRisk(engagementScore, cutoff) {
  const normalizedCutoff = cutoff / 100;
  if (engagementScore >= normalizedCutoff) {
    return "Low";
  } else if (engagementScore >= normalizedCutoff * 0.5) {
    return "Moderate";
  } else {
    return "High";
  }
}
```

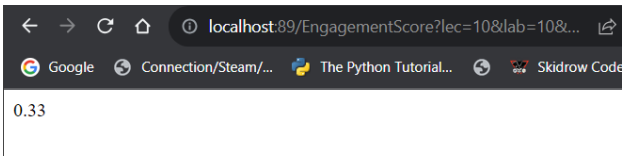
```
}
if (require.main === module) {
app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
}
```

Across the four screenshots, from top left, to bottom right, is the full totalRiskApp.js file.

I have assigned to get the index.html address. IT will also send an error message to the user if the webpage is not working. This code allows for the engagement score to be calculated, and the risk of student failure. As shown in the first image, I have a variable: "const port = 89" which will host the application port 89. I will be able to /getEngagementScore and /getRiskAssessment within the following web page. We can see this application running on below.

We have a /getRiskAssessment example shown in this left screenshot below:



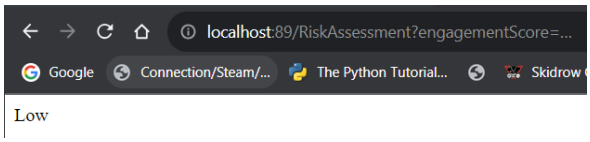


Similar we can access the same website and view the engagement score with the /getEngagementScore URL link, shown in the screenshot on the right.

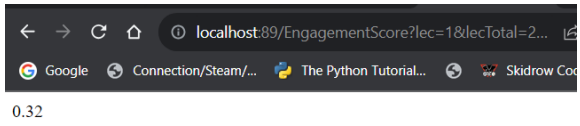
This qubengagement-failure microservice can access both engagement score and risk assessment. This is possible due to the app.get() on line 61, for /RiskAssessment, and line 34 /EngagementScore. In both scenarios, they take the input required variables such as lectures,

labs, for the engagement score feature. Whereas the /RiskAssessment would need an engagement score value, as well as a cut off.
For example: a /RiskAssessment URL would look like:
http://localhost:89/RiskAssessment?engagementScore=0.44&cutoff=0.75.

This displays a “Low” to the user within the web service.



Likewise, using the same port=89,
For the Engagement Score feature, the URL would be:



http://localhost:89/EngagementScore?lec=1&lecTotal=20&lecW=0.3&lab=15&labTotal=20&labW=0.4&supp=5&suppTotal=10&suppW=0.15&can=8&canTotal=10&canW=0.15

The failure microservice can do a lot more in comparison to the other microservices. It contains two calculating functions.

```
77  function calculateRisk(engagementScore, cutoff) {
78    const normalizedCutoff = cutoff / 100;
79    if (engagementScore >= normalizedCutoff) {
80      return "Low";
81    } else if (engagementScore >= normalizedCutoff * 0.5) {
82      return "Moderate";
83    } else {
84      return "High";
85    }
```

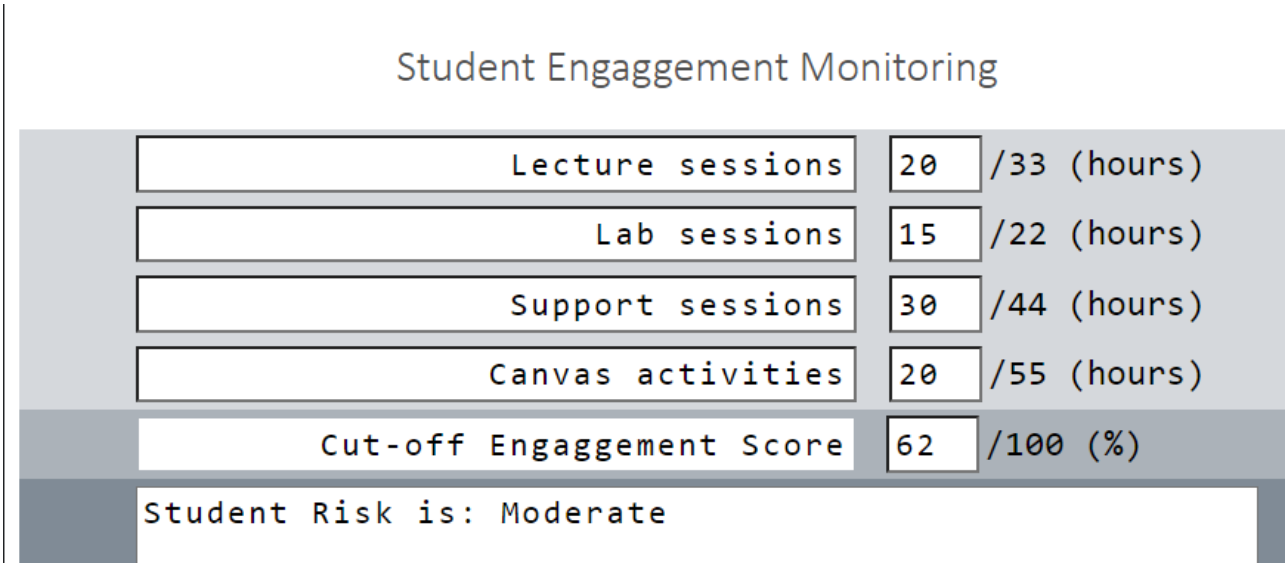
```
55  // function to calcuate engagement score
56  function calculateEngagementScore(lec, lab, supp, can) {
57    const lecScore = (lec / LectureTotal) * LectureWeight;
58    const labScore = (lab / LabTotal) * LabWeight;
59    const suppScore = (supp / SupportTotal) * SupportWeight;
60    const canScore = (can / CanvasTotal) * CanvasWeight;
61    return lecScore + labScore + suppScore + canScore;
62  }
```

These functions can be called with the app.get() respective sections as well to perform their specific calculations. Within line 77, calculateRisk() in particular. I have an if statement, checking if the current engagement is above the cut off, which automatically assumes the risk if “low” In the case the current cut off is above the engagement score, it will assume either “moderate” or “high” risk.
In the second function calcuateEngagement() it is similar again to the previous microservice three implemented.

```
219 function getRisk() {
220     let engagementScore = globalEngagementScore;
221     let lec = parseFloat(document.getElementById('attendance_1').value);
222     let lab = parseFloat(document.getElementById('attendance_2').value);
223     let supp = parseFloat(document.getElementById('attendance_3').value);
224     let can = parseFloat(document.getElementById('attendance_4').value);
225     // error handling section
226     if (lec > lecTotal || lab > labTotal || supp > suppTotal || can > canTotal) {
227         alert("Input values cannot be higher than the total hours.");
228         return; // Stop execution if any value is too high
229     }
230     let cutoff = parseFloat(document.getElementById('cut-off').value);
231     cutoff = isNaN(cutoff) ? 0 : cutoff; // Default to 0 if not a number
232
233     // Use XMLHttpRequest to send the GET request
234     let xhttp = new XMLHttpRequest();
235     xhttp.onreadystatechange = function() {
236         if (this.readyState == 4 && this.status == 200) {
237             // The response from the server should be the risk assessment
238             displayRisk(this.responseText);
239         } else if (this.readyState == 4) {
240             console.error('Error in getRisk request:', this.statusText);
241         }
242     };
243
244     let portnum = "http://localhost:89?"
245     let riskQueryString = portnum + `&engagementScore=${engagementScore}&cutoff=${cutoff}`;
246     // Construct the URL with the engagement score and cut-off
247     xhttp.open("GET", portnum + riskQueryString, true);
248     xhttp.send();
249     return;
250 }
```

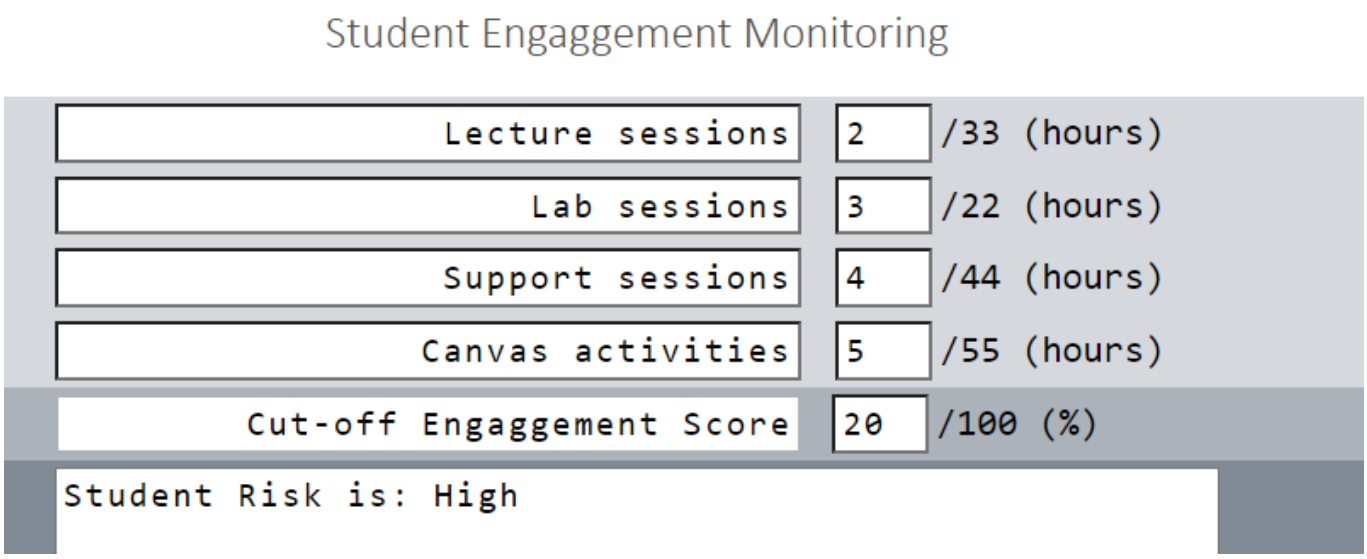
When running the application on to the front-end Student Engagement service.
I have created this getRisk() function.

On the webservice it looks as follows:
In this example: the Engagement Score = 0.61%, where the cut off = 62%
We can see the student is at risk = moderate which is a correct result



In conclusion we can see that the microservice is in full service for the front-end user. We can input different lecture, lab, support, canvas activates, additionally a separate cut off point.

As shown here, the engagement score= 10%, cut off = 20%



We can see in this scenario that the student is at a high risk of failure due to the fact they fall 10% below the cut-off engagement of 20%.
We can now proceed to the testing stage.

40295919
Connor Mallon

I have created a test.js file which looks as follows, I could not display it one screenshot, so I have split it across two.

```
const request = require('supertest');
const express = require('express');
const app = require('./totalRiskApp');

describe('Express Server Tests', () => {
  // Test for the health check endpoint
  it('GET /health - responds with server status', async () => {
    const response = await request(app).get('/health');
    expect(response.statusCode).toBe(200);
    expect(response.text).toContain('Server is running!');
  });

  // Test for the EngagementScore endpoint
  describe('GET /EngagementScore', () => {
    it('responds with calculated engagement score', async () => {
      const response = await request(app)
        .get('/EngagementScore?lec=10&lab=10&supp=10&can=10');
      expect(response.statusCode).toBe(200);
      expect(response.text).toMatch(/^\d+(\.\d{1,2})?$/); // checks if response is a numb
    });

    it('responds with error if attendance values exceed total values', async () => {
      const response = await request(app)
        .get('/EngagementScore?lec=34&lab=23&supp=45&can=56');
      expect(response.statusCode).toBe(400);
      expect(response.text).toContain("Attendance values must not exceed total values.");
    });
  });
```

```
  // Test for the RiskAssessment endpoint
  describe('GET /RiskAssessment', () => {
    it('responds with risk level based on engagement score and cutoff', async () => {
      const response = await request(app)
        .get('/RiskAssessment?engagementScore=0.6&cutoff=50');
      expect(response.statusCode).toBe(200);
      expect(response.text).toMatch(/Low|Moderate|High/);
    });

    it('responds with error for invalid engagement score or cutoff values', async () => {
      const response = await request(app)
        .get('/RiskAssessment?engagementScore=-1&cutoff=101');
      expect(response.statusCode).toBe(400);
      expect(response.text).toContain("Invalid engagement score or cutoff values.");
    });
  });

});
```

The following code ensure the expected result matches the actual result.
I have one test case to validate the working health, to ensure the web service is up and running. Additionally, two functions for the engagements score, the first ensure the returned result is a number returned to the user. The second, to ensure the total elements added must not exceed the total engagement score of 1. In the case of running the application locally, If the input values exceed the total amount, an error response will be returned “Attendance values must not exceed total values” And return a 400 response.

Within the screenshot shown on the right, I have a function which returns a respective low, moderate, high-risk probability if the engagement score is above or below the cut off range. A 200 response should respond back to ensure it is working as expected.

Additionally, if the application is run locally, I ensured the weights do not exceed 1%, as well as ensuring the cut off % does not exceeding 100%.
If the weights do exceed 1% or otherwise the cut-off% exceeding 100%. An error response of “Invalid engagement score” will be presented to the end user. The test cases in this sense ensure the end user does not interfere with incorrect data input.

```
PASS ./test.js (32.407 s)
  Express Server Tests
    ✓ GET /health - responds with server status (63 ms)
    GET /EngagementScore
      ✓ responds with calculated engagement score (8 ms)
      ✓ responds with error if attendance values exceed total values (4 ms)
    GET /RiskAssessment
      ✓ responds with risk level based on engagement score and cutoff (4 ms)
      ✓ responds with error for invalid engagement score or cutoff values (5 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        34.512 s
Ran all test suites.
```

We can see the results as shown, I have created multiple test cases as shown in the image for test.js:
We can see here that the five test cases have passed, and a following description of each is also displayed.

This result proves that the correct expected response matches the actual return values.
For test cases were cut off exceeds 100% for example, the 400 responses will be displayed to the user and prevent them from proceeding.
They will be forced to input a data value which fits the engagement score of 100% limit.
For example, if they must have a 0.90% cut-off which allows them to proceed.
Now finally proceeding to push the code onto GitLab and complete the CI pipeline testing,

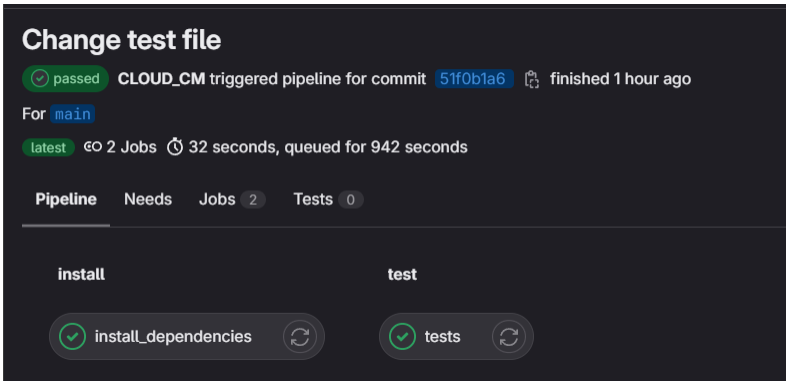
40295919
Connor Mallon

I firstly created a .gitlab-ci.yml file for the engagement score failure application.

This code is shown on the right ->

There are only two stages, an install and test.
Node.js is a lot simpler when creating a .gitlab-ci.yml file.

It will only install the necessary dependencies on npm and continue to test.
Shown here is the working pipeline, showing the two stages as mentioned earlier.




```yaml
image: node:16

stage:
  - install
  - test

cache:
  paths:
    - node_modules/

before_script:
  - npm install

install_dependencies:
  stage: install
  script:
    - npm install
  artifacts:
    paths:
      - node_modules/

tests:
  stage: test
  script:
    - npm test
```

This is the docker file which I have used for the application:

This allows for the application to be containerised and separately in a container and not through a local instance.

```dockerfile
# Specify a base image
FROM node:16-alpine

# Set the working directory in the container
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies, including 'express'
RUN npm install --only=production

# Copy the rest of the application code
COPY . .

# Your app binds to port 89 so you'll use the
EXPOSE 89

# Define the command to run your app using CMD
CMD [ "node", "totalRiskApp.js" ]
```

FUNCTION FOUR

Function (What Operation): qub-average
Repository URL:

https://repository.hal.davecutting.uk/ConnorM70/qub-average
Live Service URL: http://localhost:86/average?lecture=10&lab=10&support=10&canvas=10

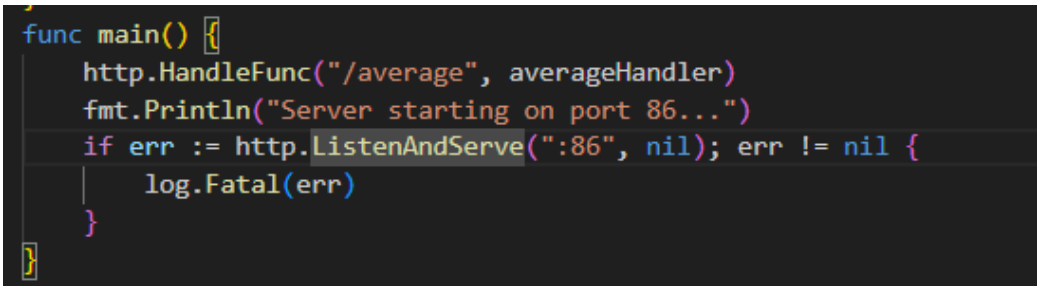
Description of Implementation (language, methods, paradigm, etc):

For the FaaS function, I have decided to get the student average engagement across all the Lectures, Labs, Support, Canvas to roughly gauge the average student engagement in hours. The code is done in Go. Which was a new language implementation.

40295919
Connor Mallon

Firstly, I have implemented this function in the Language Go. This following code called app.go:

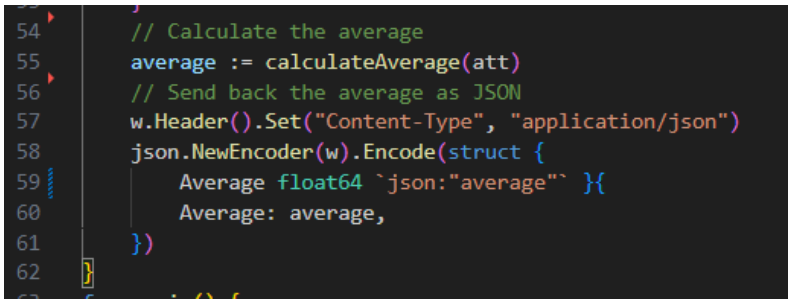
This following code contains the func calcuateAverage(attendance Attendance) which will perform the average calculation. It will get the sum of all hours, and divide by 4, (number of activities) to return a respective average.



We can see here in the main function that the server is running port 86, We can access the localhost:86 after executing the program.

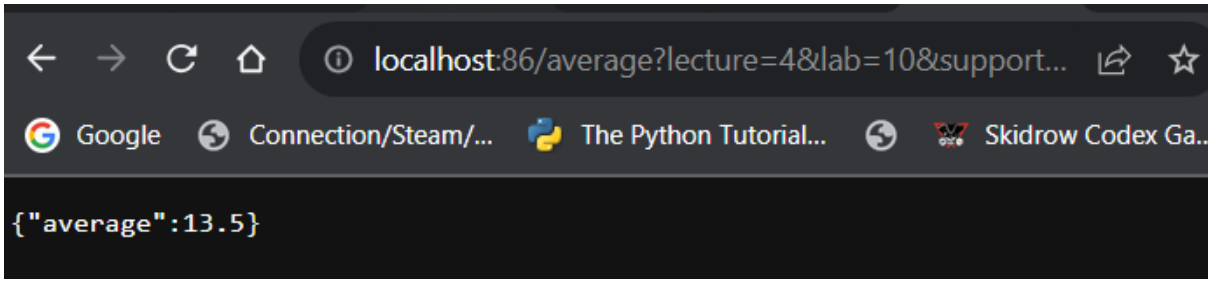
ON the image on the left,

The web page will require the parameters:
lecture, lab, support, and canvas
This will be essential to calculate the result. An example URL will be:



http://localhost:86/average?lecture=4&lab=10&support=15&canvas=25

The variable “average” will call upon the calculateAverage() function with the JSON at this parameter.
We can see a working example of this shown below:



We can see the result returns “average:13.5”, which is shown in the json.NewEncoder line, shown on line 58 in the screenshot above. Now that we can see the JSON response is working on port=86, We can port this over to the index.html.

Firstly, I have updated the front-end code on index.html in order to transfer to the working local Go code to be used for the index.html.

```
252 function getAverage() {
253     let lec = parseFloat(document.getElementById('attendance_1').value);
254     let lab = parseFloat(document.getElementById('attendance_2').value);
255     let supp = parseFloat(document.getElementById('attendance_3').value);
256     let can = parseFloat(document.getElementById('attendance_4').value);
257     // Error handling for input values
258     if (isNaN(lec) || isNaN(lab) || isNaN(supp) || isNaN(can)) {
259         alert("Please enter valid numeric values for attendance.");
260         return; // Stop execution if any value is invalid
261     }
262     // Use XMLHttpRequest to send the POST request
263     let xhttp = new XMLHttpRequest();
264     xhttp.onreadystatechange = function() {
265         if (this.readyState == 4 && this.status == 200) {
266             // Parse the JSON response from the server
267             let response = JSON.parse(this.responseText);
268             // Assuming the server sends back a JSON object with an "average" key
269             displayAverage(response.average);
270         } else if (this.readyState == 4) {
271             console.error('Error in getAverage request:', this.statusText);
272         }
273     };
274     let portnum = "http://localhost:86/average?"
275     let queryString = portnum + `&lecture=${lec}&lab=${lab}&support=${supp}&canvas=${can}`;
276     xhttp.open("GET", portnum + queryString , true);
277     xhttp.send();
278 }
```

I have created a getAverage() function which alike the other functions, parses the input values from the user, and checks if the inputs are a NaN input.
The input values will be assigned to their assigned variables, lecture, lab, support, and canvas.
The NaN values will be allowed for the “return;” to be executed on line 260. Thus prevent the program to process any further.

If the program continues, it will check the ready state, if the JSON response is 200, we have a success response and continue into the if statement.
Returning the response on the backend code running on port 86, as shown previously.
It will then execute the call to displayAverage() with the JSON response parameter.

Similarly, if the JSON response begins with a 4, for example if the response is 404 or 403, it will instead enter the else if branch and present an error message to the front-end user. “Error in getAverage request” with the following error 404 response for example.

In the case that it is a 200 response,
We can call to displayAverage() and branches to this given function as follows below:

```
function displayAverage(average)
{
    document.getElementById('output-text').value = "Student Average: " + average + " hours";
}
```

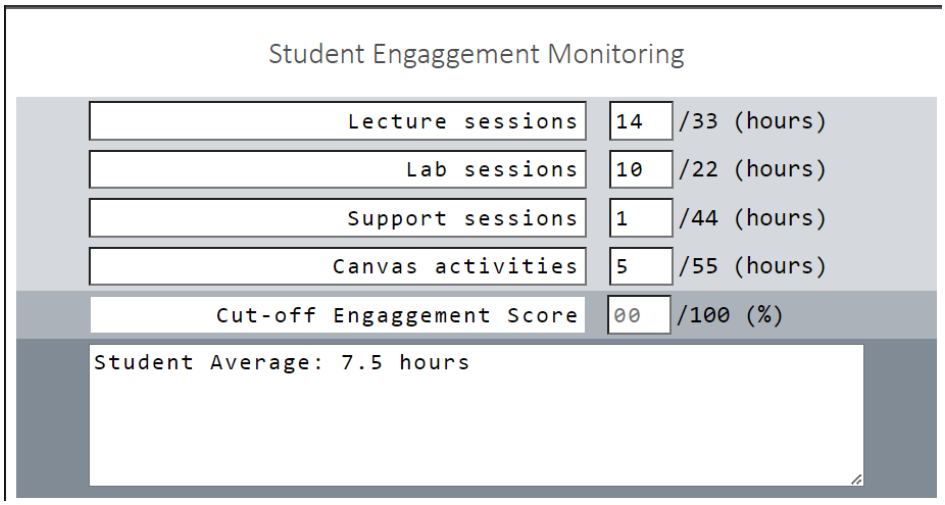
This result will be viewable to the user, on the front end service.
An example of this is shown in the previous page of the front-end service, where it returns the total average of the input values of the user.

The input parameter will be the average JSON response.

In the query String, It will be the connected URL which allows to print a given average.

```
274     let portnum = "http://localhost:86/average?"
275     let queryString = portnum + `&lecture=${lec}&lab=${lab}&support=${supp}&canvas=${can}`;
276     xhttp.open("GET", portnum + queryString , true);
277     xhttp.send();
```

An example of a working URL will be:
http://localhost:86/average?lecture=X&lab=X&support=X&canvas=X,
Where X is a user input value. This response will display such as follows:



We can see the working result as shown here:
Where the lec=14
Lab = 10, supp = 1,
Can=5

40295919
Connor Mallon

Description of Testing:
I have created the following test class for the App.go code, it is named “app_test.go”

I have two main test functions:
TestCalculateAverage()

In this function, I have checked 0 inputs, the same inputs, and different inputs, we would return an error if the expected result will be different than the actual output.

Similarly, I have a Test function called TestAverageHandler().

This function will check the URL response, ensuring the average return response on the web service, will be the same as the expected return. In this example, I have set all activities to 10. Therefore, the average should be 10. It will return an error otherwise.

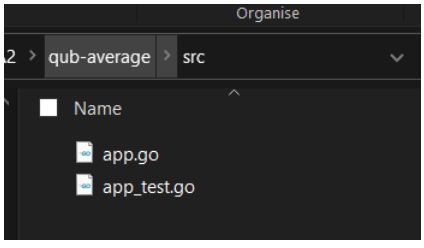
```go
package main
import (
    "net/http"
    "net/http/httptest"
    "testing"
)
func TestCalculateAverage(t *testing.T) {
    cases := []struct {
        name       string
        attendance Attendance
        want       float64
    }{
        {"all zeroes", Attendance{0, 0, 0, 0}, 0},
        {"same values", Attendance{10, 10, 10, 10}, 10},
        {"different values", Attendance{5, 15, 20, 10}, 12.5},
    }
    for _, c := range cases {
        got := calculateAverage(c.attendance)
        if got != c.want {
            t.Errorf("calculateAverage for %s: got %f, want %f", c.name, got, c.want)
        }
    }
}
func TestAverageHandler(t *testing.T) {
    req, err := http.NewRequest("GET", "/average?lecture=10&lab=10&support=10&canvas=10",
    nil)
    if err != nil {
        t.Fatal(err)
    }
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(averageHandler)

    handler.ServeHTTP(rr, req)

    if status := rr.Code; status != http.StatusOK {
        t.Errorf("handler returned wrong status code: got %v want %v",
        status, http.StatusOK)
    }
    expected := `{"average":10}` + "\n"
    if rr.Body.String() != expected {
        t.Errorf("handler returned unexpected body: got %v want %v",
        rr.Body.String(), expected)
    }
}
```

We can see the passing function of TestCalculateAverage(), TestAverageHandler

```
root@Connors-Laptop:/mnt/c/Users/Connor Mallon/Documents/MENG CSC/Level 3/CSC3065 I Cloud/A2/qub-average/src# go test -v
=== RUN   TestCalculateAverage
--- PASS: TestCalculateAverage (0.00s)
=== RUN   TestAverageHandler
--- PASS: TestAverageHandler (0.00s)
PASS
ok      qub-average/src 0.004s
```

As the console returns that the test cases have passed.
We can proceed to create the .gitlab-ci.yml file.

I have added the following .gitlab.yml file, which allows us to now complete the CI pipeline.
It only has one stage of test. It will use the current golang:1.18 image and test the app_test.go class, as well as involve the app.go class in this directory, Shown in the screenshot:



Here we can see the test stage has passed for the qub-average function.

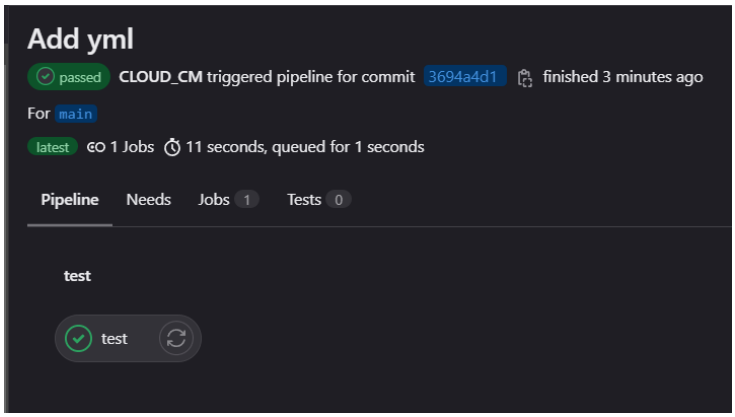
app_test.go | .gitlab-ci.yml | Dockerfile

qub-average > .gitlab-ci.yml

```yaml
stages:
  - test

variables:
  GO111MODULE: "on"

before_script:
  - 'go version'

test:
  image: golang:1.18  # Use the official Go image
  stage: test
  script:
    - go version
    - go mod tidy
    - go test -v ./...
  only:
    - main
```

40295919
Connor Mallon

Next shown here is the working CI pipeline consisting of only one test stage. Likewise, the Go. Application is straightforward and simple in comparison to the something like the C# .gitlab-ci.yml file.



Below shown here is each docker container build and run instance I have used:

For qub-maxmin:
docker build -t maxmin-app .
docker run -d --name maxmin -p 81:80 maxmin-app

For qub-sort:
docker build -t qubsort-app .
docker run -d --name sort -p 82:80 qubsort-app

For qub-totalhours:
docker build -t totalhours-app .
docker run -d --name totalhours -p 83:80 totalhours-app

For qub-engagementscore:
docker build -t engagement-score-app .
docker run -d --name engagement-score -p 84:80 engagement-score-app

For qub-risk:
docker build -t risk-app .
docker run -d --name failurerisk -p 89:80 risk-app

For qub-avg:
docker build -t avg-app .
docker run -d --name qub-avg -p 86:80 avg-app

Finally for qub-front-end:
docker build -t frontend-app .
docker run -d --name frontend -p 90:80 frontend-app

40295919
Connor Mallon

- *Task B: Improvements*

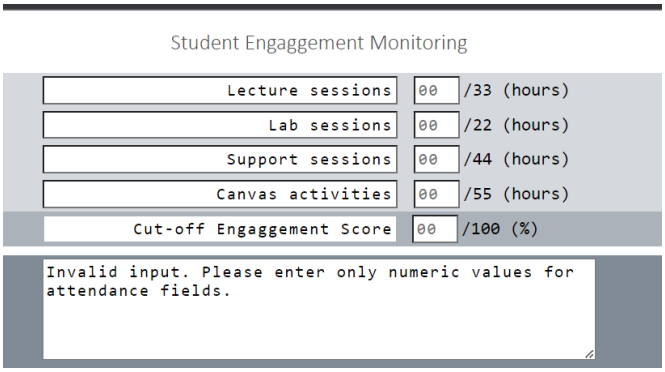
Firstly, I prevented the user to adjust the current titles by adding a 'readonly' within the CSS section of the code, for each attendance.
This adjustment looks like this following:
This now prevents the user to adjust or alter the front-end titles.

```
1" value="Lecture sessions" readonly>
" name="attendance_1" placeholder="00"><label>/33 (hours)</label>

2" value="Lab sessions" readonly>
" name="attendance_2" placeholder="00"><label>/22 (hours)</label>

3" value="Support sessions" readonly>
" name="attendance_3" placeholder="00"><label>/44 (hours)</label>

4" value="Canvas activities" readonly>
" name="attendance_4" placeholder="00"><label>/55 (hours)</label>
```

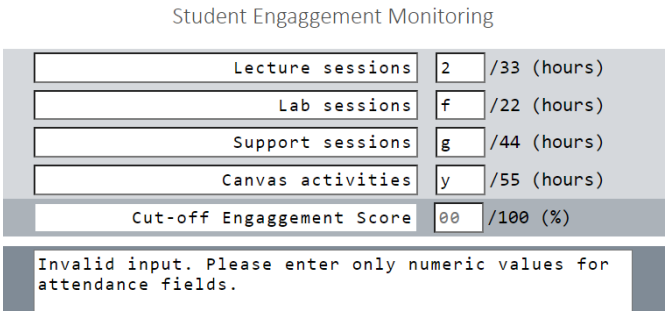
Ensuring additional error handling in the front-end code.

Max-Min
I have an error to check in the case that the user must input a value at each session, for example must input a value number or 0 for all lectures, labs, support, and canvas activities.

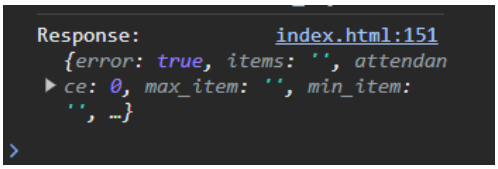
Additionally, if a user were to not enter an attendance and click the Maximum and Minimum Attendance, they would receive the following response:
This is a 400-error response which is handled by the back end php service.



This response will be caught in a try catch statement, where in the hopes a successful 200 response should appear, however in the case a user was to not input a value or input a value which is not an integer as such as a character input. The same error would also response back.



Shown here is the case where a user input as a value other than an integer.
This is the inspect element of the request shown:




```php
// Add Error handling to check if it is empty and an integer has been input
if (empty($attendance_1) || empty($attendance_2) || empty($attendance_3) || empty($attendance_4) ||
    !is_numeric($attendance_1) || !is_numeric($attendance_2) || !is_numeric($attendance_3) || !is_numeric($attendance_4)) {
    $output['error'] = true;
    $output['message'] = 'Invalid input. Please enter only numeric values for attendance fields.';
    echo json_encode($output);
    exit();
}
```

Within the PHP maximin backend source code, I have added the following exception handling – where if it is left black – output the message "Invalid input" as shown above here as well.

40295919
Connor Mallon

In the outcome of the code, I have not altered much else regarding the index.php within qubengage-maxmin.

This is the result of the max-min micro-service, shown in the next page.
The core premise remains, I have just altered the error handling instances to ensure the program knows how to response in such an exception outcome.

qubengage-maxmin > src > index.php

```php
<?php
header("Access-Control-Allow-Origin: *");
header("Content-type: application/json");
require('functions.inc.php');

$output = array(
    "error" => false,
    "items" => "",
    "attendance" => 0,
    "max_item" => "",
    "min_item" => ""
);

$item_1 = $_REQUEST['item_1'];
$item_2 = $_REQUEST['item_2'];
$item_3 = $_REQUEST['item_3'];
$item_4 = $_REQUEST['item_4'];
$attendance_1 = $_REQUEST['attendance_1'];
$attendance_2 = $_REQUEST['attendance_2'];
$attendance_3 = $_REQUEST['attendance_3'];
$attendance_4 = $_REQUEST['attendance_4'];
$items = array($item_1,$item_2,$item_3,$item_4);
$attendances = array($attendance_1,$attendance_2,$attendance_3,$attendance_4);
if (empty($attendance_1) || empty($attendance_2) ||
    empty($attendance_3) || empty($attendance_4) ||
    !is_numeric($attendance_1) || !is_numeric($attendance_2) ||
    !is_numeric($attendance_3) || !is_numeric($attendance_4)) {
    $output['error'] = true;
    $output['message'] = 'Invalid input.
    Please enter only numeric values for attendance fields.';
    echo json_encode($output);
    exit();
}
```

```php
}
// reassign into each array itemy
$items = array($item_1, $item_2, $item_3, $item_4);
$attendances = array($attendance_1, $attendance_2, $attendance_3, $attendance_4);

$max_min_items = getMaxMin($items, $attendances);

$output['items']=$items;
$output['attendance']=$attendances;
$output['max_item']=$max_min_items[0];
$output['min_item']=$max_min_items[1];

echo json_encode($output);
exit();
```

Shown on line 24, we can see the implementation of the exception handling, which ensures that any attendances that are left empty, or is not numeric, it will return the error response.

For testing purposes, I have created this following MaxMinTest.php code which displays such as:

```php
<?php
require_once __DIR__ . '/functions.inc.php';
require_once __DIR__ . '/vendor/autoload.php';

use PHPUnit\Framework\TestCase;

0 references | 0 implementations
class MaxMinTest extends TestCase {
    0 references | 0 overrides
    public function testNormalCase() {
        $items = ["Item1", "Item2", "Item3"];
        $attendances = [30, 50, 40];
        $expected = ["Item2 - 50", "Item1 - 30"]; // Max, Min
        $this->assertEquals($expected, getMaxMin($items, $attendances));
    }
    0 references | 0 overrides
    public function testEmptyArrays() {
        $items = [];
        $attendances = [];
        $expected = [null, null];
        $this->assertEquals($expected, getMaxMin($items, $attendances));
    }
    0 references | 0 overrides
    public function testValidArray() {
        $items = ["Item1", "Item2", "Item3", "Item4"];
        $attendances = [10, 20, 15, 25];
        $expected = ["Item4 - 25", "Item1 - 10"]; // Max, Min
        $result = getMaxMin($items, $attendances);
        $this->assertEquals($expected, $result);
    }

    0 references | 0 overrides
    public function testInValidArray() {
        $items = ["Item1", "Item2", "Item3", "Item4"];
        $attendances = [6, 32, 4, 13];
        $expected = ["Item2 - 32", "Item3 - 4"]; // Max, Min
        $result = getMaxMin($items, $attendances);
        $this->assertEquals($expected, $result);
    }
}
?>
```

I have four main test cases which include:
testNormalCase(),
testEmptyArrays(),
testValidArray(),
testInvalidArray()

These cases show the possible situations where a user can input data types.

Shown below is also the working display print of the four functions

We see the expected results match the actual results; we can proceed onwards.

40295919
Connor Mallon

```
PHPUnit 9.6.14 by Sebastian Bergmann and contributors.

....                                                                4 / 4 (100%)

Time: 00:00.490, Memory: 4.00 MB

OK (4 tests, 4 assertions)
```

I additionally created a .gitlab-ci.yml file which looks like:

qubengage-maxmin > .gitlab-ci.yml

```yaml
stages:
  - test

test:phpunit:
  image: php:8.2
  stage: test
  script:
    - php -v
    - apt-get update -qq
    - apt-get install -yqq libxml2-dev libonig-dev
    - docker-php-ext-install xml mbstring
    - php ./vendor/bin/phpunit --configuration phpunit.xml

```

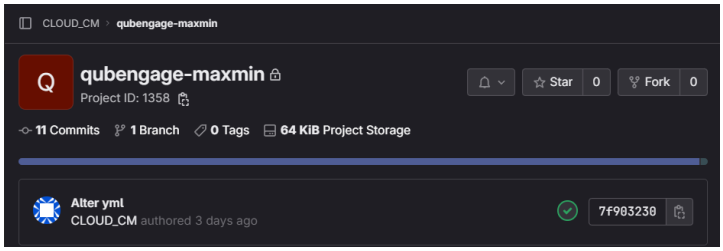
In this .gitlab-ci.yml file, where is one main test case, it will run over the stage and image shown, execute the phpunit.xml file. As shown below:

qubengage-maxmin > phpunit.xml

```xml
<phpunit bootstrap="src/functions.inc.php"
         colors="true"
         verbose="true"
         stopOnFailure="false">
    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>src/</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
            <directory suffix=".php">src/</directory>
        </whitelist>
    </filter>

    <logging>
        <log type="testdox-text" target="logs/testdox.txt"/>
        <log type="junit" target="logs/junit.xml" logIncompleteSkipped="false"/>
        <log type="coverage-html" target="logs/coverage" lowUpperBound="50" highLowerBound="90"/>
        <log type="coverage-clover" target="logs/clover.xml"/>
    </logging>

    <php>
        <ini name="error_reporting" value="E_ALL"/>
        <ini name="display_errors" value="1"/>
        <ini name="display_startup_errors" value="1"/>
        <env name="APP_ENV" value="testing"/>
    </php>
</phpunit>

```

This configuration file will set up the PHPUnit, it allows to define where test files are, and define the order of test file execution. It was essential to test and pipeline the php maxmin application.

Shown here is the final working pipeline for the qubengage-MaxMin repository.



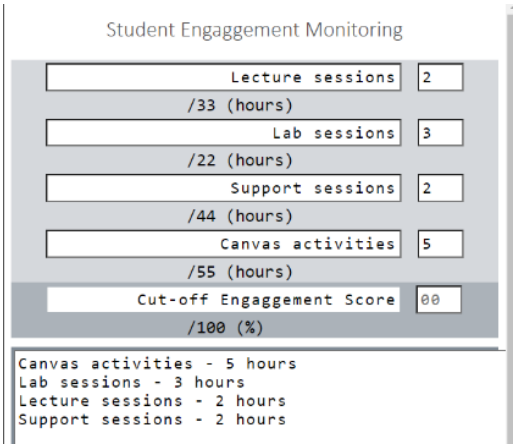
Sort Attendance

Similarly for the Sort Attendance button, if a user inputs a correct integer for each specific attendance, they receive an expected sorted attendance for each session, as shown on the screenshot on the right.

However, if a user inputs a character input at one of the following attendances, they will receive this error shown below; much alike the maximum and minimum attendance shown above.

Lecture sessions	2	/33 (hours)
Lab sessions	3	/22 (hours)
Support sessions	2	/44 (hours)
Canvas activities	g	/55 (hours)
Cut-off Engaggement Score	00	/100 (%)

```
Invalid input. Please enter only numeric values for attendance fields.
```



Likewise, this exception is handled in the same way much alike the maximum and minimum attendance code.
The final back-end code for the sorted attendance looks like the following:

The same premise will be shared with the previous code shown below; however, it will maintain different functionality with the same attendance request, exception handling and call to respected functions.
Shown below are the testing instances for the qubengage-sort, where it is called sortTest.php.

It is a lot simpler in comparison to the previous qubengage-maxmin,

```php
<?php
header("Access-Control-Allow-Origin: *");
header("Content-type: application/json");
require('functions.inc.php');

$output = array(
  "error" => false,
  "items" => "",
  "attendance" => 0,
  "sorted_attendance" => ""
);
$item_1 = $_REQUEST['item_1'];
$item_2 = $_REQUEST['item_2'];
$item_3 = $_REQUEST['item_3'];
$item_4 = $_REQUEST['item_4'];
$attendance_1 = $_REQUEST['attendance_1'];
$attendance_2 = $_REQUEST['attendance_2'];
$attendance_3 = $_REQUEST['attendance_3'];
$attendance_4 = $_REQUEST['attendance_4'];

// Add Error handling to check if it is empty and an integer has been input
if (empty($attendance_1) || empty($attendance_2) || empty($attendance_3) || empty($attendance_4) ||
    !is_numeric($attendance_1) || !is_numeric($attendance_2) || !is_numeric($attendance_3) || !is_numeric($attendance_4)) {
    $output['error'] = true;
    $output['message'] = 'Invalid input. Please enter only numeric values for attendance fields.';
    echo json_encode($output);
    exit();
}

$items = array($item_1, $item_2, $item_3, $item_4);
$attendances = array($attendance_1, $attendance_2, $attendance_3, $attendance_4);

$sorted_attendance=getSortedAttendance($items, $attendances);

$output['items']=$items;
$output['attendance']=$attendances;
$output['sorted_attendance']=$sorted_attendance;

echo json_encode($output);
exit();
```

Test Sorted Attendance

This is the test.php which I have used to test the code. I have one test case called “testGetSortedAttendance” which compares an expected case to the actual result. Where I have an array of 10, 20, 30.
Then I call the getSortedAttendance() function which sorts the given values in descending order. I then compare the result to the expected result, if the assertEquals comes out true. Then we can guarantee that the function does as expect. Therefore, we can proceed forward.
The function also tests an empty array, ensuring that the empty array, will match the expected empty array as well.

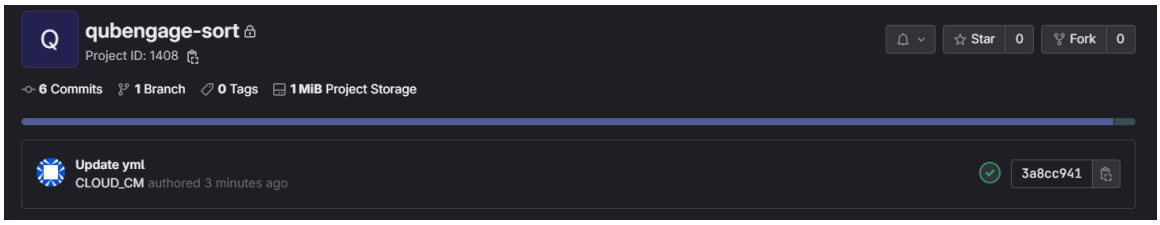
```php
<?php
require_once __DIR__ . '/functions.inc.php';
require_once __DIR__ . '/../vendor/autoload.php';

use PHPUnit\Framework\TestCase;

0 references | 0 implementations
class SortTest extends TestCase {
    0 references | 0 overrides
    public function testGetSortedAttendance() {
        $items = ['Item1', 'Item2', 'Item3'];
        $attendances = [10, 20, 30];
        $expected = [
            ['item' => 'Item3', 'attendance' => 30],
            ['item' => 'Item2', 'attendance' => 20],
            ['item' => 'Item1', 'attendance' => 10],
        ];
        $actual = getSortedAttendance($items, $attendances);
        $this->assertEquals($expected, $actual, "The actual sorted attendances do not match the expected output.");
    }

    0 references | 0 overrides
    public function testEmptyInputs() {
        $items = [];
        $attendances = [];
        $expected = [];
        $actual = getSortedAttendance($items, $attendances);
        $this->assertEquals($expected, $actual, "The function should handle empty inputs without error.");
    }
    0 references | 0 overrides
    public function testUnsortedInput() {
        $items = ['Item1', 'Item2', 'Item3', 'Item4'];
        $attendances = [45, 10, 30, 20];
        $expected = [
            ['item' => 'Item1', 'attendance' => 45],
            ['item' => 'Item3', 'attendance' => 30],
            ['item' => 'Item4', 'attendance' => 20],
            ['item' => 'Item2', 'attendance' => 10],
        ];
        $actual = getSortedAttendance($items, $attendances);
        $this->assertEquals($expected, $actual, "The function should correctly sort a completely unsorted input.");
    }

}
```

The last function is values unsorted, which randomly assigns values, and the expected array will sort by descending order. The final expected result will be 45,30,20,10.
It only has three test case which assigns three values and returns the actual sorted set.
If the expected result is like the actual result, then we can identify that the test case is working as intended.

I have one test case called “testGetSortedAttendance” which compares an expected case to the actual result. Where I have an array of 30, 50, 20, It then calls upon the getSortedAttendance() function, within the functions.inc class which sorts the given values. I then compare the result to the expected result, if the assertEquals comes out true. Then we can guarantee that the function does as we expect. Therefore, we can proceed onwards.

```
C:\Users\Connor Mallon\Documents\MENG CSC\Level 3\CSC3065 I Cloud\A2\qubengage-sort\src>..\vendor\bin\phpunit SortTest.php
PHPUnit 9.6.14 by Sebastian Bergmann and contributors.

...                                                                 3 / 3 (100%)

Time: 00:00.012, Memory: 4.00 MB

OK (3 tests, 3 assertions)
```

We can see here the test case passed for qubengage-sort, executing on php version 8.2.11.
In terms of CI-pipeline the qubengage-sort class, here we can see the working pipeline result



Which I have managed through the result of creating this .gitlab-ci.yml file:

Similarly, alike the php Maxmin, we only have one test stage, which consists of checking the php version, installing the necessary dependencies and executing.

A phpunit.xml file was additionally required in the process of a successful CI-pipeline.

qubengage-sort > .gitlab-ci.yml

```yaml
stages:
  - test

test:phpunit:
  image: php:8.2
  stage: test
  script:
    - php -v
    - apt-get update -qq
    - apt-get install -yqq libxml2-dev libonig-dev
    - docker-php-ext-install xml mbstring
    - php ./vendor/bin/phpunit --configuration phpunit.xml

```

This will call the original functions.inc.php code as shown on the top, and structures how the test case should look when calculating the test case class.

This class also allows the .gitlab-ci.yml file to be pushed and pass the pipeline test as shown on the GitLab repository, much alike the previous maxmin microservice application.

PHP requires the phpunit.xml file for pipelining purposes.

```xml
<phpunit bootstrap="src/functions.inc.php"
         colors="true"
         verbose="true"
         stopOnFailure="false">
    <testsuites>
        <testsuite name="Project Test Suite">
            <directory>src/</directory>
        </testsuite>
    </testsuites>

    <coverage includeUncoveredFiles="true" processUncoveredFiles="true">
        <include>
            <directory suffix=".php">src/</directory>
        </include>
    </coverage>

    <php>
        <ini name="error_reporting" value="E_ALL"/>
        <ini name="display_errors" value="1"/>
        <ini name="display_startup_errors" value="1"/>
        <env name="APP_ENV" value="testing"/>
    </php>
</phpunit>
```

Qub-total hours

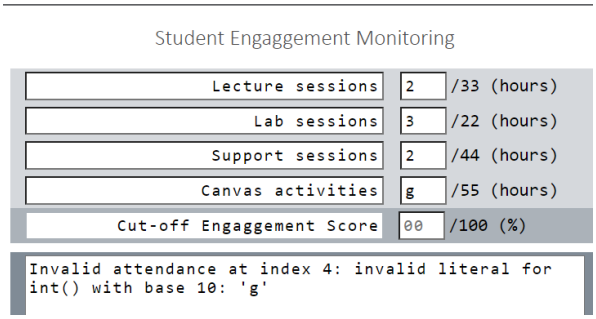
Moving on, I have added error handling instances with the python total hours back-end code, so the result looks like:

```python
from flask import Flask, request, Response, jsonify
from flask_cors import CORS
import json
import total

app = Flask(__name__)
CORS(app)

# Define the maximum attendance values for each session type
MAX_VALUES = {
    "Lecture sessions": 33,
    "Lab sessions": 22,
    "Support sessions": 44,
    "Canvas activities": 55
}
@app.route('/')
def addnumbers():
    items = []
    attendances = []
    errors = []

    for i in range(1, 5):
        item = request.args.get(f'item_{i}')
        attendance = request.args.get(f'attendance_{i}')

        if not item or not attendance:
            errors.append(f"Item or attendance for index {i} is missing.")
            continue

        try:
            attendance_int = int(attendance)
            if attendance_int < 0:
                errors.append(f"Attendance cannot be negative for {item}.")
                continue
            if attendance_int > MAX_VALUES.get(item, 100):  # Use max value or
                errors.append(f"Attendance for {item} exceeds maximum limit.")
                continue
            items.append(item)
            attendances.append(attendance_int)
        except ValueError as e:
            errors.append(f"Invalid attendance at index {i}: {str(e)}")
```

```python
    # If there are any errors, return the first one
    if errors:
        return Response(
            response=json.dumps({
                "error": True,
                "message": errors[0]
            }),
            status=400,
            mimetype='application/json'
        )

    total_attendance = total.total(*attendances)

    response_data = {
        "error": False,
        "items": items,
        "attendance": attendances,
        "total": total_attendance
    }

    return jsonify(response_data)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80)
```

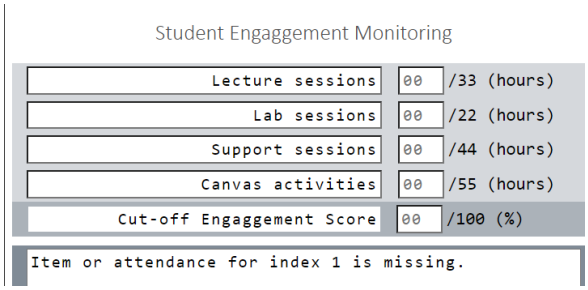
This alteration allows to ensure the input attendances cannot be left black, prevents any incongruent inputs such as character inputs from the user, as well as preventing the attendance input to be higher than total hours.
We can see a working case of this in the front-end code:

40295919
Connor Mallon



An example of incompatible data, where instead of a normal integer input, a character ‘g’ is input instead.

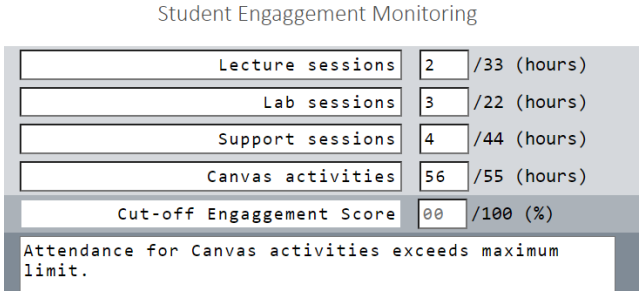
We can see the index is specified as well as the invalid input ‘g’ being acknowledged within the window box.



Similarly, if the user has left all attendances blank, they will receive this error in response.

Once again capturing an input which is not recognised by the front end and outputting a respective prompt to the user.
Specifying that index 1 is missing, in the same case if index 1 is filled, index 2 will be flagged as missing and so forth.

Similarly, they cannot exceed the total hours amount as shown here.
The user will receive a respective value in the case that an attendance exceeds the total hours amount for the given session.



It will specify the attendance which needs to be readjusted,

In this example, canvas attendance = 45 hours, when the total attendance is only 44 hours.

The user will have to look at this and input a valid value which respects the front end.

Testing

```python
import unittest
import json
from app import app

class AddNumbersTestCase(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_add_numbers(self):

        response = self.app.get(
        '/?item_1=Lecture%20sessions&attendance_1=10'
        '&item_2=Lab%20sessions&attendance_2=20'
        '&item_3=Support%20sessions&attendance_3=30'
        '&item_4=Canvas%20activities&attendance_4=10')
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data)
        self.assertEqual(data['error'], False)
        self.assertEqual(data['items'],
        ['Lecture sessions', 'Lab sessions', 'Support sessions', 'Canvas activities'])
        self.assertEqual(data['attendance'], [10, 20, 30, 10])
        self.assertEqual(data['total'], 70)

    def test_exceeds_max_values(self):
        response = self.app.get(
        '/?item_1=Lecture%20sessions&attendance_1=35'
        '&item_2=Lab%20sessions&attendance_2=20'
        '&item_3=Support%20sessions&attendance_3=30'
        '&item_4=Canvas%20activities&attendance_4=10')
        self.assertEqual(response.status_code, 400)
        data = json.loads(response.data)
        self.assertEqual(data['error'], True)
        self.assertIn("exceeds maximum limit", data['message'])

    def test_missing_parameters(self):
        # Missing parameters in the query string
        response = self.app.get('/?item_1=Lecture%20sessions&attendance_1=10')
        self.assertEqual(response.status_code, 400)
        data = json.loads(response.data)
        self.assertEqual(data['error'], True)
        self.assertIn("is missing", data['message'])
```

I have adjusted the test case total hours which now falls in line better with the new total hours app.py code.

I have four test cases, which include:
test_add_numbers(),
test_exceed_max_values(),
test_missing_paramters(),
test_negative_attendance()

```python
    def test_negative_attendance(self):
        response = self.app.get(
        '/?item_1=Lecture%20sessions&attendance_1=-1'
        '&item_2=Lab%20sessions&attendance_2=20'
        '&item_3=Support%20sessions&attendance_3=30'
        '&item_4=Canvas%20activities&attendance_4=10')
        self.assertEqual(response.status_code, 400)
        data = json.loads(response.data)
        self.assertEqual(data['error'], True)
        self.assertIn("cannot be negative", data['message'])

if __name__ == '__main__':
    unittest.main()
```

In the test cases, I have validated that the missing parameters will return a respective erorr message, similarly, with the negative values cannot be entered.
Finally if a value input exceeds the total amount of hours available for that specific microservice. It will also return an error message to the user. It will additionally specifiy which attendance needs to be looked at, as shown previously in the last page.

We can see that the test cases have passed, and we can proceed onwards.

```
OK
(.venv) PS C:\Users\Connor Mallon\Documents\MENG CSC\Level 3\CSC3065 I
ocuments/MENG CSC/Level 3/CSC3065 I Cloud/A2/.venv/Scripts/python.exe"
G CSC/Level 3/CSC3065 I Cloud/A2/qubtotal_hours/src/test.py"
....
----------------------------------------------------------------------
Ran 4 tests in 0.019s

OK
```

I had to readjust the Dockerfile as such:
Which runs and executes on local container port 80.
We can now proceed onto the next microservice.

```
FROM python:3.8-slim

WORKDIR /app

COPY requirements.txt ./

RUN pip install --no-cache-dir -r requirements.txt

COPY src/ .

EXPOSE 80

ENV NAME World

CMD ["python", "app.py"]
```

Qub engagement score

I have altered the existing C code which checks the existing attendances values and ensures that the user has input all the required fields to display a correct engagement score result.

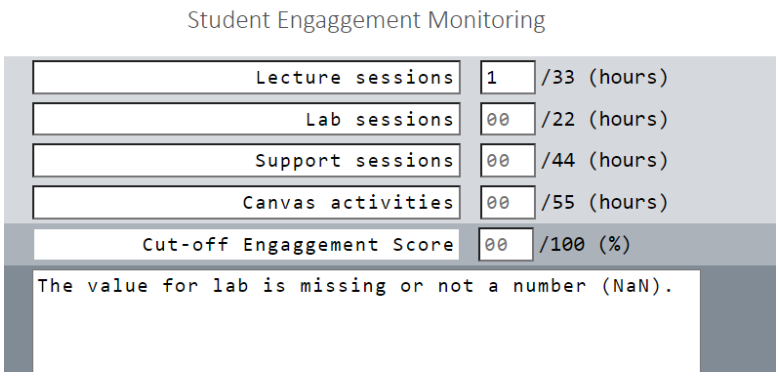
```
1  [HttpGet]
2  0 references
3  public ActionResult<double> GetEngagementScore(
4      [FromQuery] double? lec, [FromQuery] double? lecTotal,
5      [FromQuery] double? lab, [FromQuery] double? labTotal,
6      [FromQuery] double? supp, [FromQuery] double? suppTotal,
7      [FromQuery] double? can, [FromQuery] double? canTotal)
8  {
9      var parameters = new Dictionary<string, double?>
10     {
11         {"lec", lec},
12         {"lecTotal", lecTotal},
13         {"lab", lab},
14         {"labTotal", labTotal},
15         {"supp", supp},
16         {"suppTotal", suppTotal},
17         {"can", can},
18         {"canTotal", canTotal}
19     };
20
21     foreach (var param in parameters)
22     {
23         if (!param.Value.HasValue || double.IsNaN(param.Value.Value))
24         {
25             return BadRequest($"The value for {param.Key} is missing or not a number (NaN).");
26         }
27         if (param.Value <= 0)
28         {
29             return BadRequest($"The value for {param.Key} must be greater than zero.");
30         }
31     }
32
33     // Additional checks to ensure attendance does not exceed total
34     if (lec > lecTotal || lab > labTotal || supp > suppTotal || can > canTotal)
35     {
36         return BadRequest("Attendance values must not exceed total values.");
37     }
```

Shown here are the added if statements ensuring the parameter has user input value, as shown on on line 35, within the given loop.

The loop will determine which attendance has to be altered. For index 1 is lectures, index 2 labs, and so forth. This is how it ensures that, for each of the fields, a specific value has been input from the end user.
We will see the result appear later.

On the other case if a user inputs a value which is not a number, such as an arithmetic value or character input, the same prompt should appear. If a user were to input a negative value, the respected output will also prompt where the attendance – "must be greater than zero."

Test: Engagement score



Shown here, the user has not input any attendances values, an error prompt will be shown to the user, asking to input the required attendances.
Showing that the value is missing or not a number has been input.

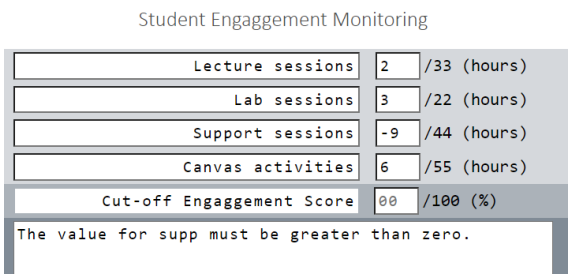
In this example, we can see that the user has left the first index attendance blank, therefore the first prompt being

recognised is the lecture attendance missing.

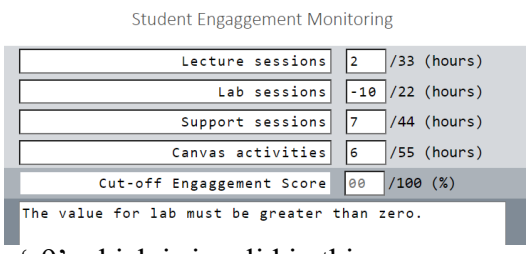
The following response will be handled by this function,
I have slightly altered this function, as I have now implemented the multiple URL instances upon completing this section. However, the same premise remains, for successful responses, it will execute through the try section, and if it is any other response from a 200. It will iterate the else block, printing the error message to the user, or it will cause the catch block to execute. Once, again it will print the error message to the user, explaining what has happened, and they will need the correct it to get a specific engagement score result.

```
let xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState === 4) {
        if (xhttp.status === 200) {
            try {
                let response = JSON.parse(xhttp.responseText);
                if (!response.error) {
                    successCallback(response);
                } else {
                    errorCallback(response.message);
                }
            } catch (e) {
                // If JSON parsing fails, pass the raw respons
                errorCallback(xhttp.responseText);
            }
        } else {
            // If the status code is not 200, attempt to parse
            try {
                let errorResponse = JSON.parse(xhttp.responseT
                errorCallback(errorResponse.message);
            } catch (e) {
                // If parsing the error response fails, pass t
                errorCallback(xhttp.responseText);
            }
        }
```

We can also see another example of this, where a user was to input a negative value.



The support attendance is specified in this example. We can see that the user has input a value ‘-9’ which is invalid in this case.



Similarly, the same premise for the Lab shown in the bottom screenshot.
We can verify the required error handling is managed as expected.

Risk of Student Failure.

```
git > qubengagement-failure > JS totalRiskApp.js > app.get('/') callback > [@] i
const path = require('path');
const express = require('express');
const cors = require('cors');
const app = express();
const port = 89;  // run test on port 89
app.use(cors());
module.exports = app;

app.get('/', (req, res) => {
  const indexPath = path.join('/mnt/c', 'Users',
'Connor Mallon', 'Documents', 'MENG CSC',
 'Level 3', 'CSC3065 I Cloud', 'A2',
 'qubengage-frontend', 'src', 'index.html');

  res.sendFile(indexPath, (err) => {
    if (err) {
      console.error('Error sending index.html:', err);
      res.status(500).send('Internal Server Error');
    }
  });
});

app.get('/health', (req, res) => {
  res.send('Server is running!');
});

const LectureTotal = 33;
const LabTotal = 22;
const SupportTotal = 44;
const CanvasTotal = 55;
const LectureWeight = 0.3;
const LabWeight = 0.4;
const SupportWeight = 0.15;
const CanvasWeight = 0.15;
```

For the risk example, I have altered the backend code as such:

```
app.get('/EngagementScore', (req, res) => {
  // Check for non-empty and non-zero values before parsing
  const lec = req.query.lec && req.query.lec !== '0' ? parseFloat(req.query.lec) : '';
  const lab = req.query.lab && req.query.lab !== '0' ? parseFloat(req.query.lab) : '';
  const supp = req.query.supp && req.query.supp !== '0' ? parseFloat(req.query.supp) : '';
  const can = req.query.can && req.query.can !== '0' ? parseFloat(req.query.can) : '';

  // Check for empty strings after the conditions above
  if (lec === '' || lab === '' || supp === '' || can === '') {
    return res.status(400).send("Please provide all attendance values. They should not be zero or left blank." +
    "\nTry again.");
  }

  // Check for values exceeding total allowed
  if (lec > LectureTotal || lab > LabTotal || supp > SupportTotal || can > CanvasTotal) {
    return res.status(400).send("Attendance values must not exceed total values.\nTry again.");
  }

  // All checks pass, calculate the engagement score
  let engagementScore = calculateEngagementScore(lec, lab, supp, can);
  engagementScore = Math.min(engagementScore, 1.0);

  // Round the score to two decimal places
  const roundedScore = Number(engagementScore.toFixed(2));
  res.send(roundedScore.toString());
});

// Function to calculate engagement score
function calculateEngagementScore(lec, lab, supp, can) {
  const lecScore = (lec / LectureTotal) * LectureWeight;
  const labScore = (lab / LabTotal) * LabWeight;
  const suppScore = (supp / SupportTotal) * SupportWeight;
  const canScore = (can / CanvasTotal) * CanvasWeight;
  return lecScore + labScore + suppScore + canScore;
}
```

40295919
Connor Mallon

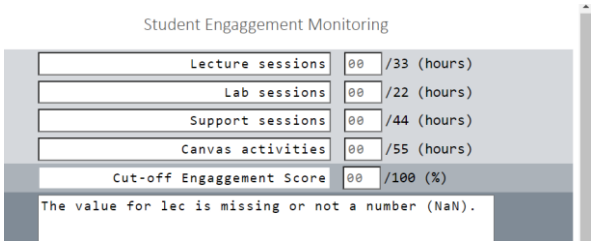
I have added error handling for both the engagement score feature and risk assessment, to ensure all cases of incongruent input has been verified.

```
// Check url for /RiskAssessment
app.get('/RiskAssessment', (req, res) => {
  const engagementScore = req.query.engagementScore && req.query.engagementScore !== '0' ?
  parseFloat(req.query.engagementScore) : '';
  const cutoff = req.query.cutoff && req.query.cutoff !== '0' ?
   parseFloat(req.query.cutoff) : '';

  // Check for empty strings or invalid values
  if(engagementScore === '' || cutoff === '' || engagementScore < 0 ||
   engagementScore > 1 || cutoff < 0 || cutoff > 100) {
    return res.status(400).send("Invalid engagement score or cutoff values.");
  }

  const risk = calculateRisk(engagementScore, cutoff);
  res.send(risk);
});

function calculateRisk(engagementScore, cutoff) {
  const normalizedCutoff = cutoff / 100;
  if (engagementScore >= normalizedCutoff) {
    return "Low";
  } else if (engagementScore >= normalizedCutoff * 0.5) {
    return "Moderate";
  } else {
    return "High";
  }
}

if (require.main === module) {
  app.listen(port, () => {
    console.log(`Server running on port ${port}`);
  });
}
```

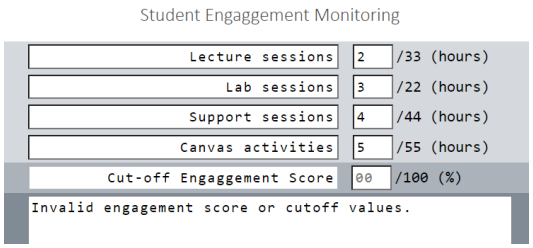
The user in this case will be promoted with a proper result which addresses the issue which they have input.

These added code changes involve a validation check to ensure the current attendance does not exceed the total amount of attendance time possible. Such as the lecture hours cannot exceed 33 hours which stated on the front-end page.
The user will get a response returned saying "Attendance must not exceed total hours."

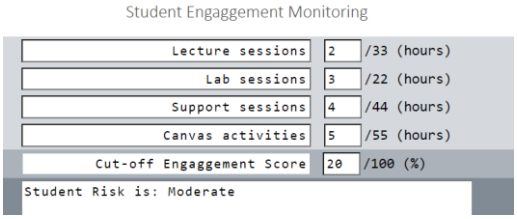
Additionally adding cases where attendances cannot be left blank, asking the respective user to input a correct value, as well as ensuring the risk assessment output cannot allow for a cut-off above 100, or below 0. The end user will get a respective response back.



As shown here, the back-end code will iterate through the engagement score initially, therefore my engagement score container must also be running to compute the risk of student failure.



If a user were to input all the attendances, however, disregard the cut off they would receive this error:
Ensuring that all attendance, including the cut-off must be validated.



Where I have now ensured a correct value for each attendance and cut-off which allows to return a correct result.
In this case the URL example would like the following:

http://localhost:89/RiskAssessment?engagementScore=0.10&cutoff=20.
Which prints the moderate result, when running directing. If any instances of the URL is left blank, an appropriate error appears.

40295919
Connor Mallon

Average Engagement

Similarly, I have adjusted the following code as such for the Go.
Average application

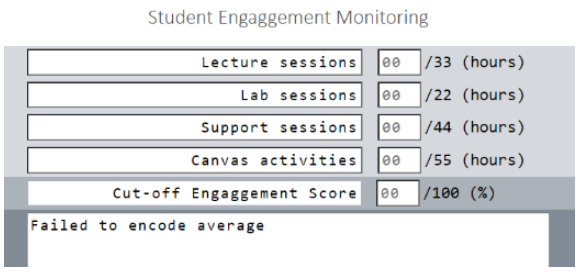
```go
package main
import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "strconv"
)
type Attendance struct {
    Lecture float64 `json:"lecture"`
    Lab     float64 `json:"lab"`
    Support float64 `json:"support"`
    Canvas  float64 `json:"canvas"`
}
const (
    // Define constants according to the screenshot
    LectureTotal  = 33.0
    LabTotal      = 22.0
    SupportTotal  = 44.0
    CanvasTotal   = 55.0
)

func calculateAverage(attendance Attendance) float64 {
    total := attendance.Lecture + attendance.Lab + attendance.Support
    + attendance.Canvas
    average := total / 4
    return average
}

func averageHandler(w http.ResponseWriter, r *http.Request) {
    // Set CORS headers for any incoming request
    w.Header().Set("Access-Control-Allow-Origin", "*")
    w.Header().Set("Access-Control-Allow-Methods", "GET, OPTIONS")
    w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

    // Handle the preflight request for CORS
    if r.Method == http.MethodOptions {
        w.WriteHeader(http.StatusOK)
        return
    }

    // Only proceed with GET requests for the actual data request
    if r.Method != http.MethodGet {
        http.Error(w, "Method not supported", http.StatusMethodNotAllowed)
        return
    }
```

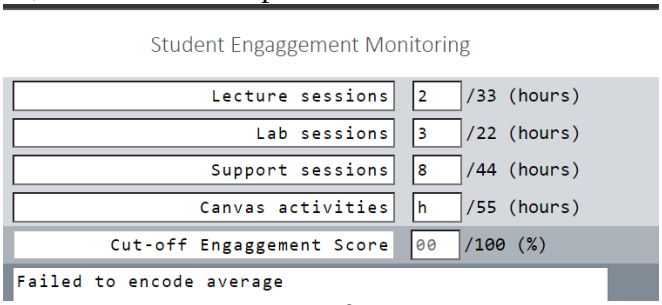
```go
    // Only proceed with GET requests for the actual data request
    if r.Method != http.MethodGet {
        http.Error(w, "Method not supported", http.StatusMethodNotAllowed)
        return
    }

    // Parse query parameters from URL
    query := r.URL.Query()
    lecStr := query.Get("lecture")
    labStr := query.Get("lab")
    suppStr := query.Get("support")
    canStr := query.Get("canvas")

    // Check if any of the parameters are empty
    if lecStr == "" || labStr == "" || suppStr == "" || canStr == "" {
        http.Error(w, "All attendance values must be provided", http.StatusBadRequest)
        return
    }

    // Parse query parameters into floats
    lec, errLec := strconv.ParseFloat(lecStr, 64)
    lab, errLab := strconv.ParseFloat(labStr, 64)
    supp, errSupp := strconv.ParseFloat(suppStr, 64)
    can, errCan := strconv.ParseFloat(canStr, 64)

    // Check for parsing errors
    if errLec != nil || errLab != nil || errSupp != nil || errCan != nil {
        http.Error(w, "Invalid attendance values", http.StatusBadRequest)
        return
    }

    // Check if any attendance values are negative
    if lec < 0 || lab < 0 || supp < 0 || can < 0 {
        http.Error(w, "Attendance values cannot be negative", http.StatusBadRequest)
        return
    }

    // Check if any attendance values exceed their respective totals
    if lec > LectureTotal || lab > LabTotal || supp > SupportTotal || can > CanvasTotal {
        http.Error(w, "Attendance values must not exceed total values",
        http.StatusBadRequest)
        return
    }
```

```go
    // Create an Attendance instance and calculate the average
    att := Attendance{
        Lecture: lec,
        Lab:     lab,
        Support: supp,
        Canvas:  can,
    }
    average := calculateAverage(att)

    // Respond with the calculated average as JSON
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK) // Explicitly set the status code to 200 OK
    err := json.NewEncoder(w).Encode(struct {
        Average float64 `json:"average"`
    }{
        Average: average,
    })

    // Check for errors in JSON encoding
    if err != nil {
        http.Error(w, "Failed to encode average", http.StatusInternalServerError)
    }
}

func main() {
    http.HandleFunc("/average", averageHandler)
    fmt.Println("Server starting on port 86...")
    log.Fatal(http.ListenAndServe(":86", nil))
}
```

I have added an error validation, checking if the input values are left blank, and prompt the user with an appropriate response. Additionally validate that any inputs which are less than 0, values cannot be larger than respective total, as well as check for parsing errors, shown on line 68 will also be flagged. I have defined the totals within the backend code making it easier to ensure the validation holds true when converting to front end code.

Additionally, I have ensured that the returned JSON encoding will also return a corresponding value, else response with an error, that it cannot be parsed.

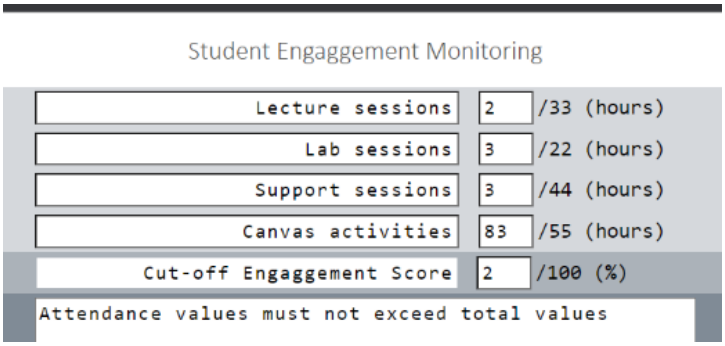




Shown above and below are the respective error responses shown in the front end.

In the case no attendance is entered, or an invalid input is recognised will present this display.

Attendance exceeds the total hours available.



40295919
Connor Mallon

- *Task C: Proxy*

- Function (What Operation): reverseproxy.py
- Repository URL: https://repository.hal.davecutting.uk/ConnorM70/reverseproxy
- Live Service URL: http:localhost:5000/?port=(portnum)

I created my reverse proxy in python, I used ChatGPT for the initial base of proxy() function, additionally configure_service() function with ports the proxy onto port 5000, then forwards the request onto the backend code such as the total hours microservice or the qub-engagementscore micro-service. As well as the assistance of generative AI to help with threading.lock() implementation.

Proxy:

qubproxy > reverseproxy.py > proxy

```
1   from flask import Flask, request, Response, jsonify
2   from flask_cors import CORS
3   import requests
4   import threading
5   app = Flask(__name__)
6   CORS(app)
7
8   proxy_config = {
9       '81': 'http://localhost:81',
10      '82': 'http://localhost:82',
11      '83': 'http://localhost:83',
12      '84': 'http://localhost:84/EngagementScore', # check engagement URL
13      '89': 'http://localhost:89/RiskAssessment', # check risk assessment URL
14      '86': 'http://localhost:86/average' # FAAS Implementation
15  }
16  config_lock = threading.Lock()
17  @app.route('/', methods=['GET', 'POST', 'OPTIONS'])
18  def proxy():
19      # Extract port from the query string
20      port_str = request.args.get('port')
21      if not port_str or port_str not in proxy_config:
22          return jsonify({'error': 'Service not found or no port specified'}), 404
23
24      base_url = proxy_config[port_str]
25
26      query_params = request.args.to_dict()
27      query_params.pop('port', None)
28      query_string = "&".join(f"{key}={value}" for key, value in query_params.items())
29      full_url = f"{base_url}?{query_string}"
30
31      if request.method == 'OPTIONS':
32          # OPTIONS requests are usually sent as CORS preflight,
33          # to check if the CORS protocol is understood.
34          response = Response()
35          response.headers['Access-Control-Allow-Origin'] = '*'
36          response.headers['Access-Control-Allow-Methods'] = 'GET, POST, OPTIONS'
37          response.headers['Access-Control-Allow-Headers'] = 'Content-Type, Authorization'
38          return response
```

On line 8, we can see the list of available URL address that the proxy contains, it contains the existing URL address of the MaxMin, Sort, total hours, and so forth. We can contain all the microservices in the "proxy_config" array.
Thus, allowing to access the back-end code.
The proxy URL would be such as this:
localhost:5000/port=81?

For example, where port=81 would allow to host the first microservice (MaxMin) for example.
Similarly, allowing port=82 would allow the qub-sort to start running.

On line 24, we can see the array of proxy config accessing each specific port, then containing that within an array called "base_url" where this url is the localhost:500/port=81? For example, and an extension "query string" will be added onto this url in order to get the lecture, lab, support, canvas totals in order to link to the correct URL address.

40295919
Connor Mallon

```
39
40      resp = requests.request(
41          method=request.method,
42          url=full_url,
43          headers={key: value for (key, value) in request.headers if key != 'Host'},
44          data=request.get_data(),
45          allow_redirects=False
46      )
47
48      excluded_headers = ['content-encoding', 'content-length', 'transfer-encoding', 'connection']
49      headers = {name: value for (name, value) in resp.headers.items() if name.lower() not in excluded_headers}
50
51      # Send back the response to the client
52      response = Response(resp.content, resp.status_code, headers=headers)
53      response.headers['Access-Control-Allow-Origin'] = '*'
54      return response
55
56  @app.route('/configure/<int:port>', methods=['POST'])
57  def configure_service(port):
58      port_str = str(port)
59      new_url = request.data.decode('utf-8')
60      with config_lock:
61          proxy_config[port_str] = new_url
62      return jsonify({'message': f"Configured service on port {port_str} with URL {new_url}"}), 200
63
64  if __name__ == '__main__':
65      app.run(host='0.0.0.0', port=5000, debug=True, threaded=True)
66
```

Additionally, if the proxy address is not on the given proxy_config array. It will return a message where “service not found or no port specified” and return a 404 error.

In the second screenshot here, we’re accessing json requests, where it automatically allows for localhost: URL addresses to be forwarded back to the backend code.

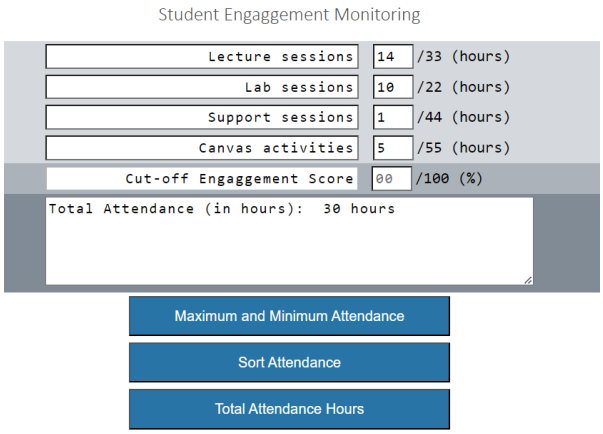
If the request made is authentic, there is no need to redirect the request to another web service. We can instead receive the client request and forward onto the requested microservice function. We can dynamically change the service, by simply changing the port access from 81, to 82 for example.
This response is then handled at line 58, where the actual returned content data, json 200 response and headers will be considered and stored to a variable called “response” Then brought to the client backend code.

Similarly, with the qub-total hours microservice. I have implemented this, with the thanks of python flask, where I can execute and run the program as shown in the terminal:

```
WARNING: This is a development server. Do not use it in a production deployment. Use
a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://10.14.73.199:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 128-589-463
```

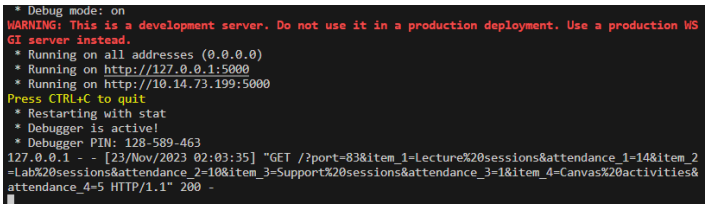
This will receive any incoming requests from the front end and determine what to do there on. The reverse proxy is simply from front-end to client side. Compared to the traditional proxy of client-side to front end service.

40295919
Connor Mallon



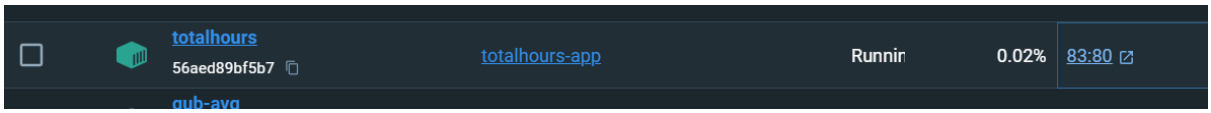
We can see here; I have input values to the front-end service of values.

Whenever I hit the blue button, it sends a request to the python proxy terminal which looks as follows:



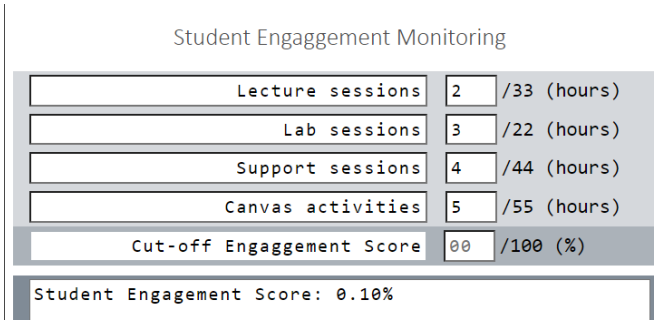
This takes the port=83, as well as the query string as shown above in proxy python code description. Which as shown in the terminal now: takes attendance_1=14, att_2=10, att_3=1, att_4=5. We can see a 200 successful response as well.

This is then transferring to the back end total hours python code which is running in a docker container.

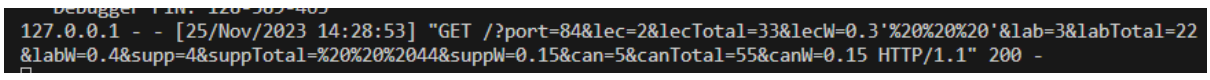


We can see with the aid of a proxy; we can redirect network traffic to specified locations to where it is needed.

Similarly, if the engagement scores where to print such as this:



Then the response will be handedled by the proxy and return this response shown below:



We can see that the lecture, lab, support and canvas engagement has been specified in the response and returns a successful 200 towards to end of the URL.

The url to access the proxy address would look like the following:
http://5000/?port=84&lec=2&lecTotal=33&lecW=0.3'%20%20%20'&lab=3&labTotal=22&
&labW=0.4&supp=4&suppTotal=%20%20%2044&suppW=0.15&can=5&canTotal=55&canW=0.15

It will filter through the proxy URL, and redirect the traffic to the specified data, to where it is needed. If I were to run it directly, the URL simply looks like http://localhost:84.
With this level of protection, we can ensure that malicious intend can also be avoided in the case of an attack.

For testing purpose, I have created this test case class called: testProxy.py.
In this case, I have four test cases, ensuring that the proxy will operate as expected.

```python
import unittest
import json
from reverseproxy import app

class ProxyTest(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_home_page(self):
        response = self.app.get('/')
        self.assertEqual(response.status_code, 404)

    def test_valid_proxy_request(self):
        # Test a valid proxy request to a configured port
        response = self.app.get('/?port=81')
        self.assertEqual(response.status_code, 200)
        # Additional checks can be added here

    def test_invalid_proxy_request(self):
        # Test request to an unconfigured port
        response = self.app.get('/?port=9999')
        self.assertEqual(response.status_code, 500)

    def test_configure_service(self):
        # Test the configuration of a new service
        new_url = "http://localhost:9999"
        response = self.app.post('/configure/9999', data=new_url)
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data.decode('utf-8'))
        self.assertEqual(data['message'], f"Configured service on port 9999 with URL {new_url}")

if __name__ == '__main__':
    unittest.main()
```

```
..
----------------------------------------------------------------------
Ran 4 tests in 0.021s

OK
```

Shown in the testing instance, we have passed all the test.
In the cases, I have configured a temporary new port 9999, we have then assumed a 200 successful response to ensure it is working as expected, this is how we can ensure that the proxy is redirecting the necessary url responses.

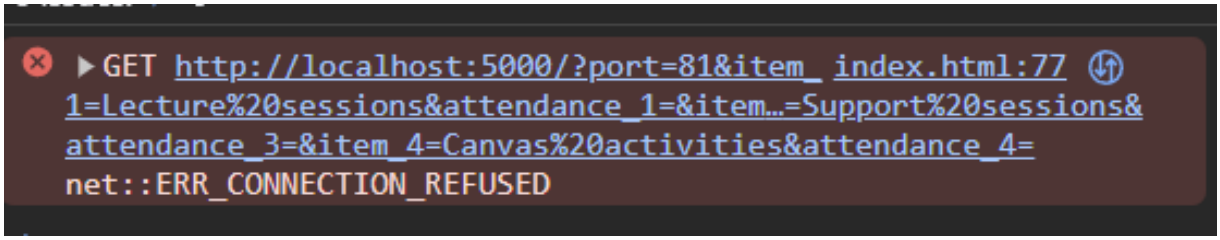
I have tested a valid URL running on port 81, which is the existing qubengage-max min shown here in the docker container:

We can also identify that the proxy is also getting the response from maxmin.

In conclusion, we can see the proxy is working as expected, it is able to access each specific function and create a “middleman” instance where the function is not possible, unless the proxy is also up and running through the python code “reverseproxy.py”
We can see the front-end responses being forwarded to the proxy, then the proxy will then send the request to the back-end code.

In the case the proxy is not running, I will get a returned message as such:

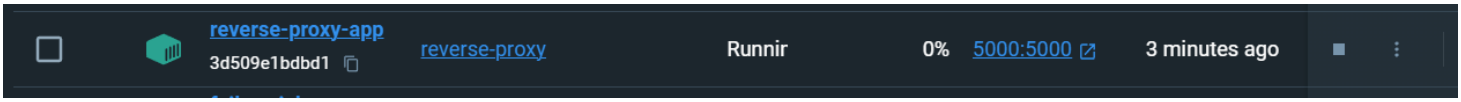
40295919
Connor Mallon



Where a connection is not found through the url: http://localhost:5000/?port=81
Where this is the MaxMin function, and in result no output will come.
The end user will not see or visualise any output, despite inputting all the correct attendances.

Unless the proxy is up and running again, only then can we see an output response.

I have now converted the reverse proxy to a container as shown:



Using the following Dockerfile to do so:

This allows for us to run the docker container alongside other applications, without having to manually run and execute the reverse proxy code from visual studio code.

The service now allows for proxy porting through http: locahost:5000/?port=(portnum) Where we can access each microservice.

```
1  FROM python:3.9-slim
2
3  WORKDIR /usr/src/app
4
5
6  COPY requirements.txt .
7
8  RUN pip install --no-cache-dir -r requirements.txt
9
10 COPY reverseproxy.py .
11 EXPOSE 5000
12
13 CMD ["python", "./reverseproxy.py"]
```

When running the containerised version of the code, It was required to additionally alter the services array, in order for the docker container to recoginise the specific URL's for each given micro-service.
In this case, I have altered the services as such within the proxy_config array:

```
# run for the container
proxy_config = {
    '81': 'http://host.docker.internal:81',
    '82': 'http://host.docker.internal:82',
    '83': 'http://host.docker.internal:83',
    '84': 'http://host.docker.internal:84/EngagementScore', # check engagement URL
    '89': 'http://host.docker.internal:89/RiskAssessment', # check risk assessment URL
    '86': 'http://host.docker.internal:86/average' # FAAS Implementation
}
```

Implementing this design instead of the localhost:81 will allow for the containerised proxy application, to also recognise the containerised microservices I have completed.

Because I had to adjust the proxy application when containerised,

I have added the entire code for the reverseproxy.py code down below:

This is the final result of the proxy application when containerised.

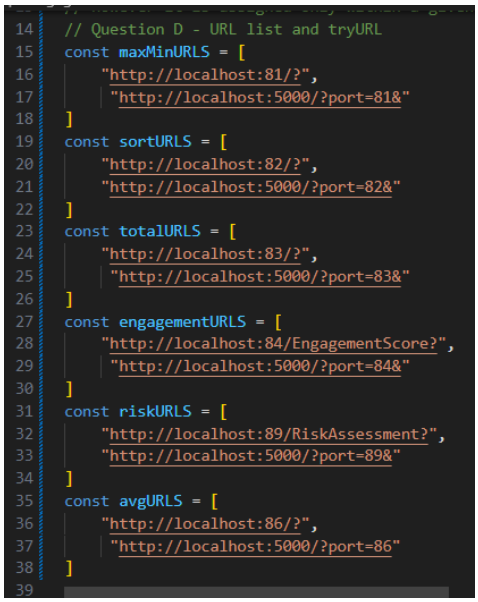
40295919 > reverseproxy > reverseproxy.py

```python
from flask import Flask, request, Response, jsonify
from flask_cors import CORS
import requests
import threading
app = Flask(__name__)
CORS(app)

# run for the container
proxy_config = {
    '81': 'http://host.docker.internal:81',
    '82': 'http://host.docker.internal:82',
    '83': 'http://host.docker.internal:83',
    '84': 'http://host.docker.internal:84/EngagementScore', # check engagement URL
    '89': 'http://host.docker.internal:89/RiskAssessment', # check risk assessment URL
    '86': 'http://host.docker.internal:86/average' # FAAS Implementation
}
config_lock = threading.Lock()
@app.route('/', methods=['GET', 'POST', 'OPTIONS'])
def proxy():
    # Extract port from the query string
    port_str = request.args.get('port')
    if not port_str or port_str not in proxy_config:
        return jsonify({'error': 'Service not found or no port specified'}), 404

    base_url = proxy_config[port_str]

    query_params = request.args.to_dict()
    query_params.pop('port', None)
    query_string = "&".join(f"{key}={value}" for key, value in query_params.items())
    full_url = f"{base_url}?{query_string}"

    if request.method == 'OPTIONS':
        response = Response()
        response.headers['Access-Control-Allow-Origin'] = '*'
        response.headers['Access-Control-Allow-Methods'] = 'GET, POST, OPTIONS'
        response.headers['Access-Control-Allow-Headers'] = 'Content-Type, Authorization'
        return response
    resp = requests.request(
        method=request.method,
        url=full_url,
        headers={key: value for (key, value) in request.headers if key != 'Host'},
        data=request.get_data(),
        allow_redirects=False
    )
    excluded_headers = ['content-encoding', 'content-length', 'transfer-encoding', 'connection']
    headers = {name: value for (name, value) in resp.headers.items() if name.lower() not in excluded_headers}

    # Send back the response to the client
    response = Response(resp.content, resp.status_code, headers=headers)
    response.headers['Access-Control-Allow-Origin'] = '*'
    return response
```

```python
@app.route('/configure/<int:port>', methods=['POST'])
def configure_service(port):
    port_str = str(port)
    new_url = request.data.decode('utf-8')
    with config_lock:
        proxy_config[port_str] = new_url
    return jsonify({'message': f"Configured service on port {port_str} with URL {new_url}"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True, threaded=True)
```

- *Task D: Frontend Failure Handler*

To implement the multiple URL's, I have separated each specific microservice into an array, and allowed the microservice to access either the direct URL, or the proxy related URL. I have done so as this:



As shown, the maxmin can access port 81, or proxy 5000/?port=81, similarly, the sort microservice is able to access port 82, or proxy 5000/?port=82.
I have done so for all five functions within the index.html page.

I have additionally created a new function, called "tryURLs" which handles all the xttp JSON responses for all of the functions I have completed.

The functions is then used by maxmin, sort, total and so forth.

In the case the response is either 200 or otherwise, it is handled appropriately shown in the if statements.

Additionally if it were anything other than the specific 400 or 200 response, it is handled appropriately in the catch blocks, and calls to the displayError()

The only exception is where the desired output is for the risk of failure. Where this will be explained upon approaching that front end of code.

The functions will iterate through their given arrays, and if the length of the array has been reached, without a successful 200 response from the service. It will return the error "All services are currently unavailable." However in the case a URL does operate as intended, it will proceed as normal through the access of either the direct URL, or the proxy connected URL.

40295919
Connor Mallon

Front end code for each function

I have added below each edited function which calls upon the tryURLS function.

getMaxmin()

```
function getMaxMin() {
    // Retrieve the input values and validate them
    let item_1 = document.getElementById('item_1').value;
    let item_2 = document.getElementById('item_2').value;
    let item_3 = document.getElementById('item_3').value;
    let item_4 = document.getElementById('item_4').value;
    let attendance_1 = document.getElementById('attendance_1').value;
    let attendance_2 = document.getElementById('attendance_2').value;
    let attendance_3 = document.getElementById('attendance_3').value;
    let attendance_4 = document.getElementById('attendance_4').value;

    // Validation for attendance not to exceed total hours
    if (attendance_1 > lecTotal || attendance_2 > labTotal || attendance_3 > suppTotal
                                                    || attendance_4 > canTotal) {
        alert("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
    // Construct the query string with the input values
    let queryString = "item_1=" + encodeURIComponent(item_1) + "&attendance_1="
        + encodeURIComponent(attendance_1) + "&item_2=" + encodeURIComponent(item_2) + "&attendance_2="
        + encodeURIComponent(attendance_2)+ "&item_3=" + encodeURIComponent(item_3) + "&attendance_3="
        + encodeURIComponent(attendance_3) + "&item_4=" + encodeURIComponent(item_4) + "&attendance_4="
        + encodeURIComponent(attendance_4);

    tryURLs(maxMinURLS, queryString,
        response => {displayMaxMin(response.max_item, response.min_item); },
        errorMessage => { displayError(errorMessage); }
    );
}
```

getTotal()

```
// working error handling + multiple URLS
function getTotal() {
    let item_1 = document.getElementById('item_1').value;
    let item_2 = document.getElementById('item_2').value;
    let item_3 = document.getElementById('item_3').value;
    let item_4 = document.getElementById('item_4').value;

    let attendance_1 = document.getElementById('attendance_1').value;
    let attendance_2 = document.getElementById('attendance_2').value;
    let attendance_3 = document.getElementById('attendance_3').value;
    let attendance_4 = document.getElementById('attendance_4').value;

    document.getElementById('output-text').value = ''; // Clear previous messages

    if (attendance_1 > lecTotal || attendance_2 > labTotal
                        || attendance_3 > suppTotal || attendance_4 > canTotal) {
        displayError("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
    let queryString = "item_1=" + encodeURIComponent(item_1) + "&attendance_1="
                                                    + encodeURIComponent(attendance_1)
                    + "&item_2=" + encodeURIComponent(item_2) + "&attendance_2="
                                                    + encodeURIComponent(attendance_2)
                    + "&item_3=" + encodeURIComponent(item_3) + "&attendance_3="
                                                    + encodeURIComponent(attendance_3)
                    + "&item_4=" + encodeURIComponent(item_4) + "&attendance_4="
                                                    + encodeURIComponent(attendance_4);

    tryURLs(totalURLS, queryString, displayTotal, displayError);
}
```

getSortedAttendance()

```
function getSortedAttendance() {
    let item_1 = document.getElementById('item_1').value
    let item_2 = document.getElementById('item_2').value
    let item_3 = document.getElementById('item_3').value
    let item_4 = document.getElementById('item_4').value
    let attendance_1 = document.getElementById('attendance_1').value
    let attendance_2 = document.getElementById('attendance_2').value
    let attendance_3 = document.getElementById('attendance_3').value
    let attendance_4 = document.getElementById('attendance_4').value

    if (attendance_1 > lecTotal || attendance_2 > labTotal || attendance_3 > suppTotal
                                                    || attendance_4 > canTotal) {
        alert("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
    let queryString = "item_1=" + encodeURIComponent(item_1)
                                                    + "&attendance_1=" + encodeURIComponent(attendance_1)
        + "&item_2=" + encodeURIComponent(item_2) + "&attendance_2=" + encodeURIComponent(attendance_2)
        + "&item_3=" + encodeURIComponent(item_3) + "&attendance_3=" + encodeURIComponent(attendance_3)
        + "&item_4=" + encodeURIComponent(item_4) + "&attendance_4=" + encodeURIComponent(attendance_4);

        tryURLs(sortURLS, queryString,
          response => {
              // Process the sorted attendance data
              let sorted_attendance_returned = response.sorted_attendance;
              let sorted_attendance = '';
              for (let i = 0; i < sorted_attendance_returned.length; i++) {
                  sorted_attendance += sorted_attendance_returned[i]['item']
                          + ' - ' + sorted_attendance_returned[i]['attendance'] + ' hours\n';
              }
              document.getElementById('output-text').value = sorted_attendance.trim();
              // Display sorted attendance
          },
          errorMessage => { displayError(errorMessage); }
      );
    }
```

getEngagementScore()

```
function getEngagementScore() {
    // Actual attendance values from the input fields
    let lecInput = document.getElementById('attendance_1').value.trim();
     let labInput = document.getElementById('attendance_2').value.trim();
     let suppInput = document.getElementById('attendance_3').value.trim();
     let canInput = document.getElementById('attendance_4').value.trim();
    // Check if any of the parsed numbers are NaN and set them to 0 if true
    let lec, lab, supp, can;
    let lecW = 0.3;
    let labW = 0.4;
    let suppW = 0.15;
    let canW = 0.15;

    // Error handling for input values greater than the totals
    if (lec > lecTotal || lab > labTotal || supp > suppTotal || can > canTotal) {
        alert("Input values cannot be higher than the total hours.");
        return; // Stop execution if any value is too high
    }
     // Attempt to parse the input values to floats
     lec = parseFloat(lecInput);
     lab = parseFloat(labInput);
     supp = parseFloat(suppInput);
     can = parseFloat(canInput);

     // Set the values to an empty string if they are NaN
     lec = isNaN(lec) ? '' : lec;
     lab = isNaN(lab) ? '' : lab;
     supp = isNaN(supp) ? '' : supp;
     can = isNaN(can) ? '' : can;
     // Use XMLHttpRequest to send the GET request
     let queryString = `lec=${encodeURIComponent(lec)}&lecTotal=${encodeURIComponent(lecTotal)}`
         + `&lab=${encodeURIComponent(lab)}&labTotal=${encodeURIComponent(labTotal)}`
         + `&supp=${encodeURIComponent(supp)}&suppTotal=${encodeURIComponent(suppTotal)}`
         + `&can=${encodeURIComponent(can)}&canTotal=${encodeURIComponent(canTotal)}`;
     tryURLs(engagementURLS, queryString, displayEngagementScore, displayError);
}
```

From these given four functions, all follow a similar format, where calling the tryURLS's function occurs at the end of the specific function. It will firstly parse all the attendances and create an approate URL queary string which is then passed into the tryURLS to complete the specific operations in order to print a result to the end user.
Similarly shown below for the getRisk() and getAvgerage() functions.

40295919
Connor Mallon

getRisk()

```
function getRisk() {
    let lecInput = document.getElementById('attendance_1').value.trim();
    let labInput = document.getElementById('attendance_2').value.trim();
    let suppInput = document.getElementById('attendance_3').value.trim();
    let canInput = document.getElementById('attendance_4').value.trim();

    let lec = lecInput ? parseFloat(lecInput) : '';
    let lab = labInput ? parseFloat(labInput) : '';
    let supp = suppInput ? parseFloat(suppInput) : '';
    let can = canInput ? parseFloat(canInput) : '';
    // Check if any of the values are NaN and replace with blank string
    lec = isNaN(lec) ? '' : lec;
    lab = isNaN(lab) ? '' : lab;
    supp = isNaN(supp) ? '' : supp;
    can = isNaN(can) ? '' : can;
    // Check if the input values exceed the total hours
    let cutoff = parseFloat(document.getElementById('cut-off').value);
    cutoff = isNaN(cutoff) ? '' : cutoff;
    let engagementQueryString = `lec=${lec}&lecTotal=${lecTotal}&lab=${lab}&labTotal=
    ${labTotal}&supp=${supp}&suppTotal=${suppTotal}&can=${can}&canTotal=${canTotal}`;
    // Handle the engagement score response
    function handleEngagementScore(engagementScore) {
        // Directly use the engagementScore value assuming it's already a number
        console.log("Engagement score here: " + engagementScore);
        if (isNaN(engagementScore)) {
            displayError("Invalid engagement score received.");
            return;
        }
        let riskQueryString = `engagementScore=${engagementScore}&cutoff=${cutoff}`;
        tryURLs(riskURLS, riskQueryString, displayRisk, displayError);
    }
    // Start the process by trying to get the engagement score first
    tryURLs(engagementURLS, engagementQueryString, handleEngagementScore, displayError);
}
```

getAverage ()

```
function getAverage() {
    let lec = parseFloat(document.getElementById('attendance_1').value);
    let lab = parseFloat(document.getElementById('attendance_2').value);
    let supp = parseFloat(document.getElementById('attendance_3').value);
    let can = parseFloat(document.getElementById('attendance_4').value);
    lec = isNaN(lec) ? '' : lec;
    lab = isNaN(lab) ? '' : lab;
    supp = isNaN(supp) ? '' : supp;
    can = isNaN(can) ? '' : can;
    // Error handling for input values
    let queryString = `lecture=${lec}&lab=${lab}&support=${supp}&canvas=${can}`;
    tryURLs(avgURLS, queryString, displayAverage, displayError);
}
```

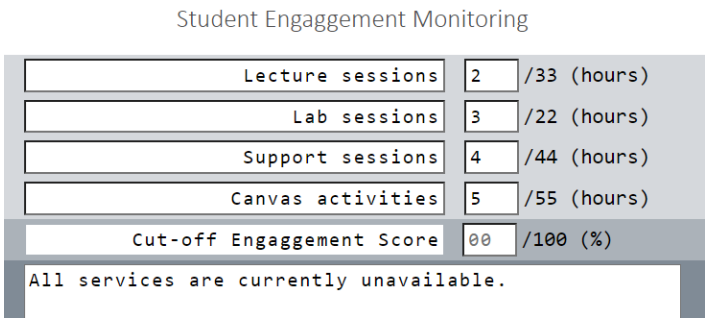
The two functions getRisk() and getAvgerage() have been significantly shortened now in comparison to the previous iterations.

Likewise, each function initially parses all the input attendances to be used with an accompanying query string. Then parsed into a tryURL's function.

Each function specified will be able to access each corresponding URL, for example getTota()
It will be able to access the proxy port of localhost:5000/?port=83.
However, in the instance that is not operation, it will instead use the direct port of localhost:83.

In this case, if a possible port is unavailable to access, it will move to the next corresponding URL address.

However, in the instances that no URL address is available, the following prompt will appear to the end user.



Signifying to the user, that is unable to access any possible URL address.

This prompt will additionally appear to the user, for all specific functions if not the URL is not found.

In summary, we can conclude that the code works as intended, and provides an overhead for failing URL responses. The tryURL's function will iterate through each URL within the specific URL instance, and handle all the responses. Additionally, printing the corresponding result of a successful URL, or print the error as shown above, that "all services are currently unavailable" despite the appropriate attendance input.

40295919
Connor Mallon

• *Task E: Monitoring*

- Function (What Operation): monitor.py
- Repository URL: https://repository.hal.davecutting.uk/ConnorM70/qub-monitor
- Live Service URL: http://localhost:5500/monitor

I have created the following monitor.py code which calls upon docker container files and has predefined values which tests against an actual and expected value, then calls upon an assert_equals function which then compares both values.

If they are the same, return a true statement for being correct, else return false.

Shown on the right, are each specific function for the monitoring.
Shown here are specific values and assuming an expected response within the expected block.
We have each service returning its docker container instance.

```
qub-monitor > monitor.py
1  from flask import Flask, jsonify, request
2  from flask_cors import CORS
3  from apscheduler.schedulers.background import BackgroundScheduler
4  import requests
5  import time
6  import json
7  import math
8  from datetime import datetime
9  import uuid
10
11 app = Flask(__name__)
12 CORS(app)
13
14 SERVICES = [
15     "http://maxmin:80/",
16     "http://sort:80/",
17     "http://totalhours:80/",
18     "http://engagement-score:80/EngagementScore?",
19     "http://failurerisk:80/RiskAssessment?",
20     "http://qub-avg:80/average?"
21 ]
22
```

Shown on the below, are the maximin and sort functions, using a predetermined input and an expected response.

I was unable to dynamically assign a actual and expected result test case, which caused more issues than necessary.

However with the predetermined set. It will assign a consistent actuala and consistent expected which are both maintained to be correct.

In the case that the actual does not meet the expected.

Then we will receive an error and have to fix this issue.

```
23 TEST_CASES = {
24     "http://maxmin:80/": [
25         {
26             "input": {
27                 'item_1': 'Lecture sessions', 'attendance_1': 50,
28                 'item_2': 'Lab sessions', 'attendance_2': 40,
29                 'item_3': 'Support sessions', 'attendance_3': 30,
30                 'item_4': 'Canvas activities', 'attendance_4': 20 },
31             "expected": {
32                 "error": False,
33                 "items": ["Lecture sessions",
34                 "Lab sessions", "Support sessions", "Canvas activities"],
35                 "attendance": [50, 40, 30, 20],
36                 "max_item": "Lecture sessions - 50",
37                 "min_item": "Canvas activities - 20" }}],
38
39     "http://sort:80/": [
40         {
41             "input": {
42                 "item_1": "Lecture sessions", "attendance_1": 2,
43                 "item_2": "Lab sessions", "attendance_2": 3,
44                 "item_3": "Support sessions", "attendance_3": 4,
45                 "item_4": "Canvas activities", "attendance_4": 5
46             },
47             "expected": {
48                 "sorted_attendance": [
49                     {"item": "Canvas activities", "attendance": 5},
50                     {"item": "Support sessions", "attendance": 4},
51                     {"item": "Lab sessions", "attendance": 3},
52                     {"item": "Lecture sessions", "attendance": 2}
53                 ]
54             }
55         }
56     ],
57     "http://totalhours:80/":[
```

40295919
Connor Mallon

Below are the other test case, totalhours and engagement-score examples within the monitoring system.

```
"http://totalhours:80/":[
        {
            "input": {
            'item_1': 'Lecture sessions', 'attendance_1': 10,
            'item_2': 'Lab sessions', 'attendance_2': 20,
            'item_3': 'Support sessions', 'attendance_3': 30,
            'item_4': 'Canvas activities', 'attendance_4': 10 },

        "expected": {
            "error": False,
            "items": ["Lecture sessions", "Lab sessions",
            "Support sessions", "Canvas activities"],
            "attendance": [10, 20, 30, 10],
            "total": 70 }}],

"http://engagement-score:80/EngagementScore?":  [
{
    "input": {
        'lec': 2, 'lecTotal': 33, 'lecW': 0.3,
        'lab': 14, 'labTotal': 22, 'labW': 0.4,
        'supp': 4, 'suppTotal': 44, 'suppW': 0.15,
        'can': 5, 'canTotal': 55, 'canW': 0.15
    },
    "expected": 0.3
}],
```

Where an input is taken, and an expecgted output is received. For total hours, the attendance hours are given, with an expected item associated with it, aswell as a "total" result which will sum all the given attendance hours.

Additionally the engagement score takes the input paramters of the attendances, weights and the given totals for each attendance.

In this case, it only gives one expected result which is the engagement score, if they do not match, it will give a "correct: false" result. In the case, they do match, then the outcome will be the score itself.

Similarily below, if the failure risk and the average actual results do not match the expected results. Then the result will be false, failure risk will take two input values, shown the engagement score and the cut off, It will prints a string such as "High", "Moderate", "Low" for the expected result.
In this case, it is "Low".

```
82      "http://failurerisk:80/RiskAssessment?":  [
83          {
84          "input": {
85              'engagementScore': 0.44,
86              'cutoff': 0.75
87              },
88              "expected": "Low" } ],
89
90        "http://qub-avg:80/average?": [
91          {
92            "input": {
93                'lecture': 20,
94                'lab': 10,
95                'support': 30,
96                'canvas': 20
97              },
98              "expected" : {
99                  "average": 20 }}],
100 }
```

Additionally the qub-avg service will print the single expected result of the average values taken.

For each attendance it will divide the total by 4, the number of attendances accounted for.

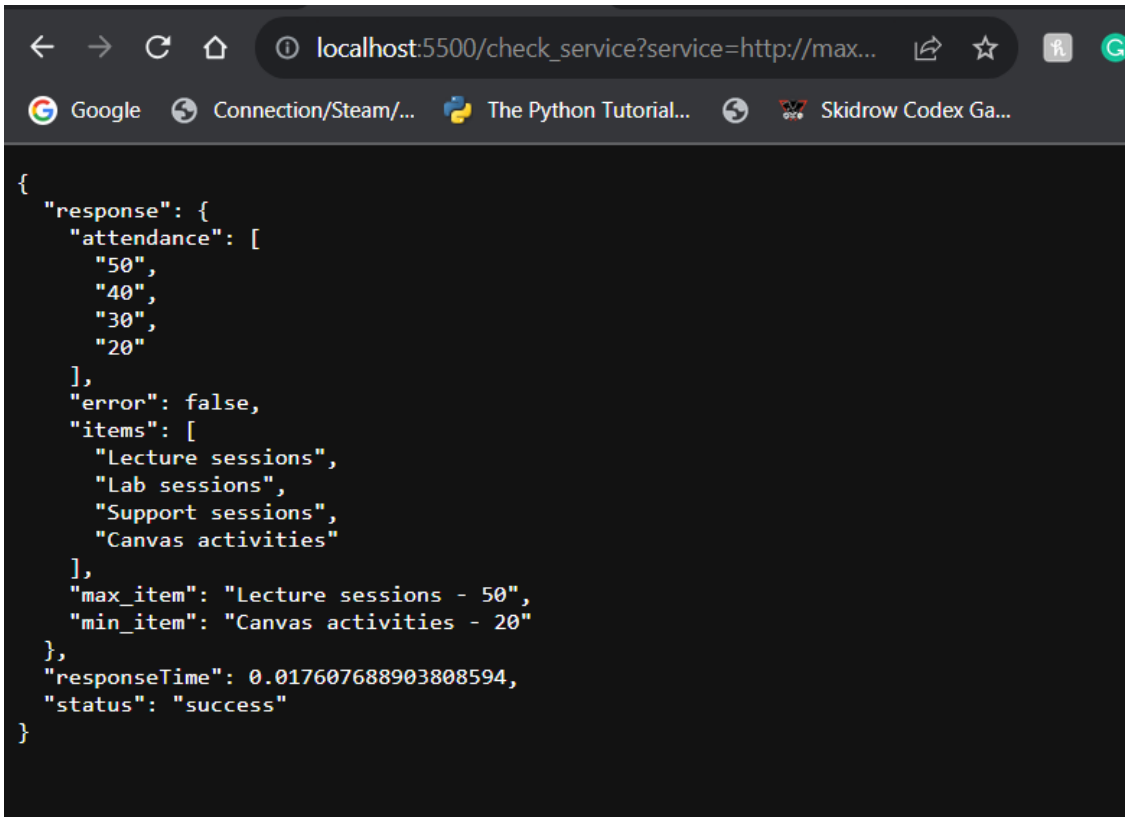
For each attendance hour chanhed, the expected result will also changed, depenedant on the total number of hours the given student has attendanded.

Continuing on with the code, we have a fubction called check_service

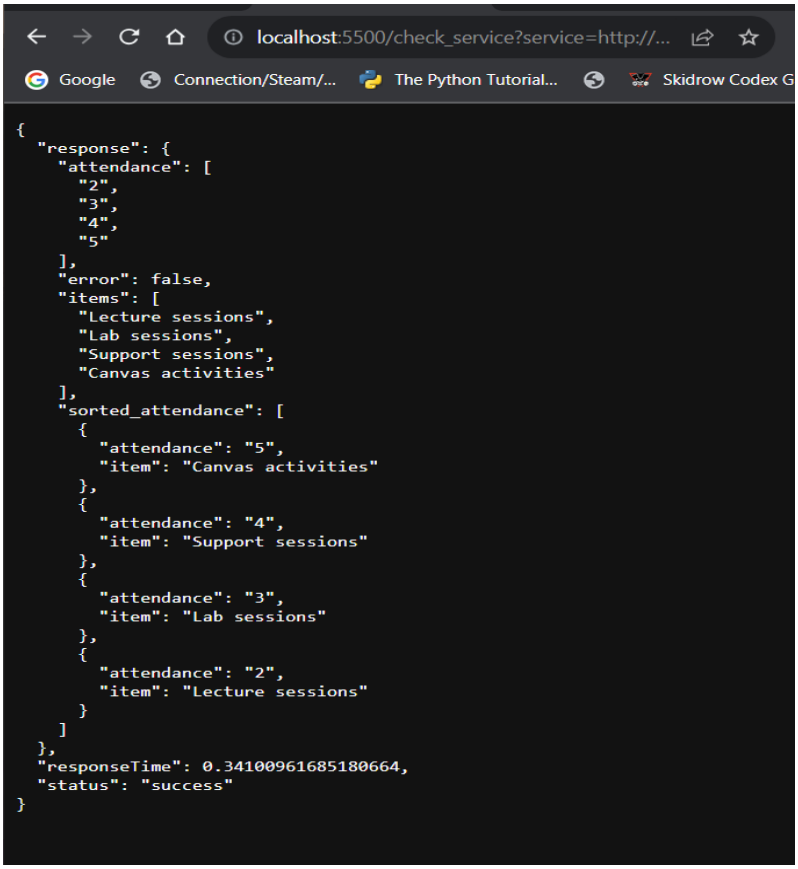
```python
@app.route('/check_service', methods=['GET'])
def check_service():
    service_identifier = request.args.get('service')
    if service_identifier is None:
        return jsonify({"status": "error", "message": "Service identifier is required"}), 400

    service_url = next((url for url in SERVICES if service_identifier in url), None)
    if service_url:
        test_case = TEST_CASES.get(service_url, [{}])[0]
        try:
            response_text, response_time = monitor_service(service_url, test_case["input"])
            return jsonify({"status": "success", "response": response_text, "responseTime": response_time})
        except Exception as e:
            return jsonify({"status": "error", "message": str(e)}), 500
    else:
        return jsonify({"status": "error", "message": "Service not found"}), 404

```

Which It will specificially check the health and functionality of each service.
As shown below, each service running:

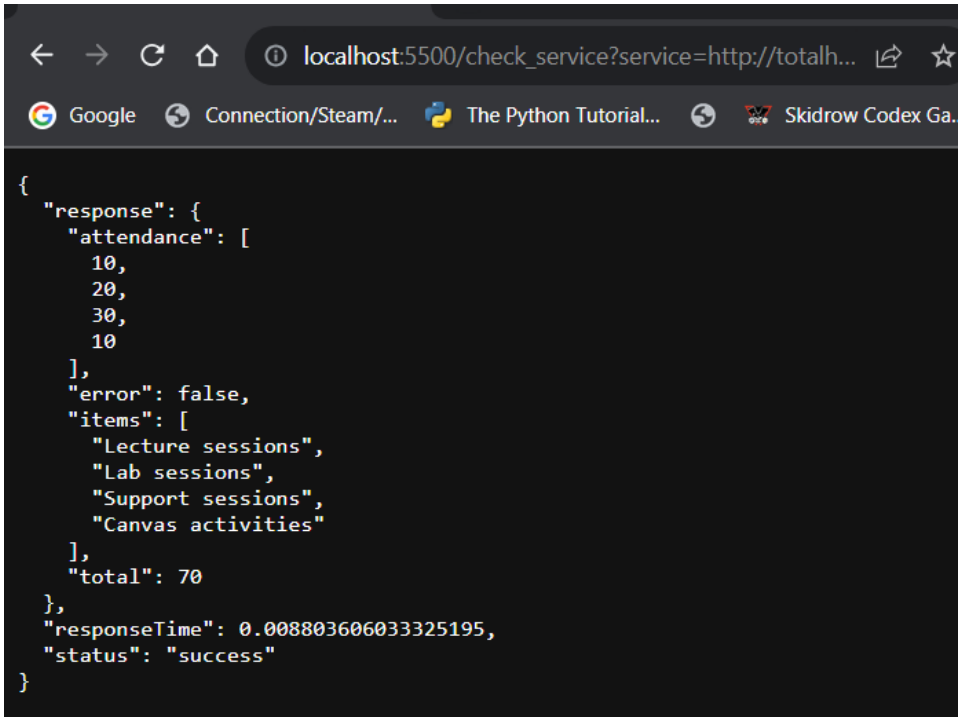
A parameter example for the first service looks as follows:
/check_service?service=http://maxmin:80/
We can see the screenshot given below prints the actual results, and their given result, With a response time and a successful status. Showing that the service works as expected.



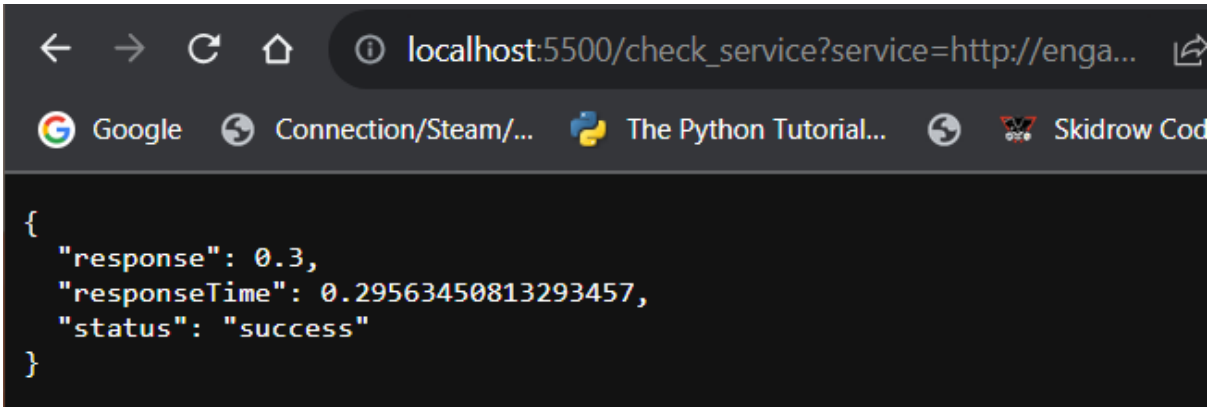
Where using Similarly with the other services.
In this case, it is the sort service.



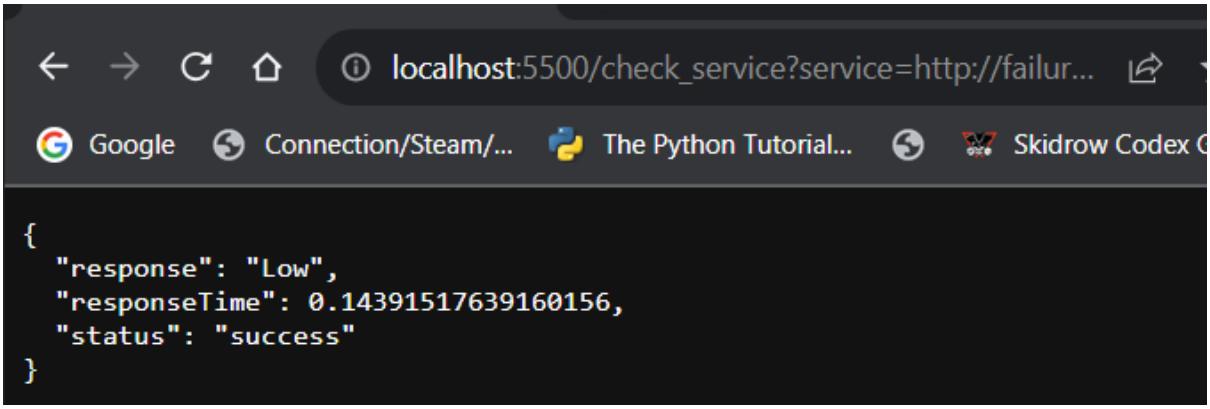
Shown below is the check_service/totalhours



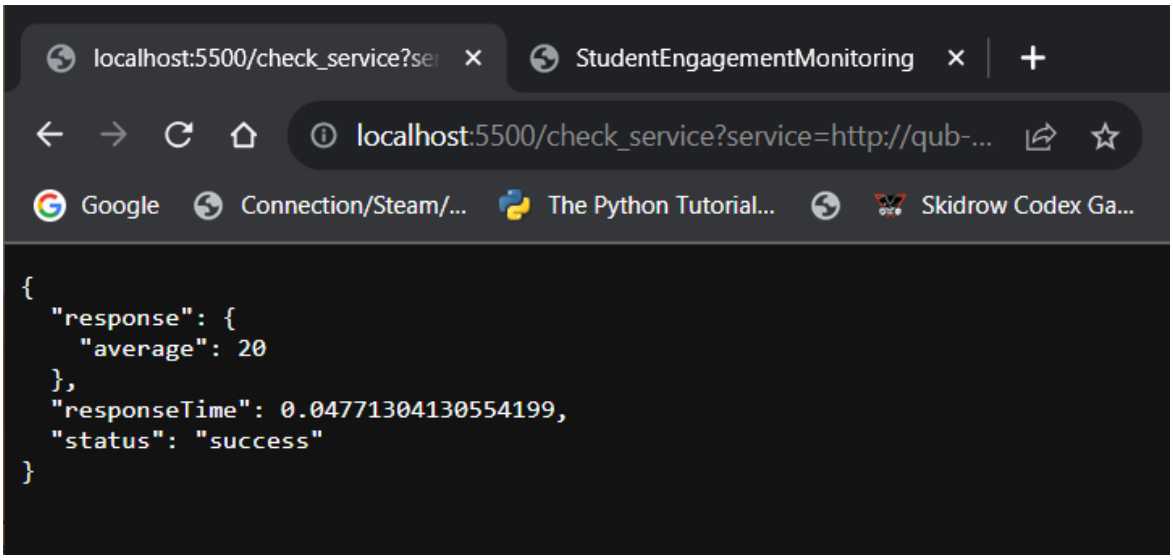
Below is the check_service/engagement-score



Below is the check_service/failurerisk



Below is the check_service/qub-avg



Each service as shown as a successful status so we can verify that each service is up and operating as expected.
Each service will show an expected, a response time a error status or a successful status
Keep in mind that, because I am running everything within a container, I have created a docker-compose.yml file which will explained later.

Moving onto the next function within monitor.py as shown below:
I have a function called monitor ()

```python
@app.route('/monitor', methods=['GET'])
def monitor():
    results = []
    for service_url, test_cases in TEST_CASES.items():
        for test_case in test_cases:

            try:
                unique_id = uuid.uuid4()
                response_text, response_time = monitor_service(service_url, test_case["input"])
                is_correct, error_message = assert_equals(response_text, test_case["expected"], service_url)
                result = {
                    "actual": response_text,
                    "expected": test_case["expected"],
                    "responseTime": response_time,
                    "correct": is_correct,
                    "error": error_message,
                    "url": service_url
                }
                # Record and print the result
                record_result(service_url, result)
                print_result(result)
                result["id"] = str(unique_id)

                # Append the result to the results list
                results.append(result)

            except Exception as e:
                # Append error information if an exception occurs
                results.append({"url": service_url, "error": str(e), "responseTime": time.time() - start_time})

    return jsonify(results)
```

Which will specific all the services at the same time. It will print the actual and expected results within one JSON response shown using the URL: localhost:5500/monitor
As shown in the next page>


```
[
  {
    "actual": {
      "attendance": [
        "50",
        "40",
        "30",
        "20"
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "max_item": "Lecture sessions - 50",
      "min_item": "Canvas activities - 20"
    },
    "correct": true,
    "error": "",
    "expected": {
      "attendance": [
        50,
        40,
        30,
        20
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "max_item": "Lecture sessions - 50",
      "min_item": "Canvas activities - 20"
    },
    "id": "a2c87bc4-c059-4966-a8e5-3f37af0f330a",
    "responseTime": 0.12154150009155273,
    "url": "http://maxmin:80/"
  },
  {
    "actual": {
      "attendance": [
        "2",
        "3",
        "4",
        "5"
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "sorted_attendance": [
        {
          "attendance": "5",
          "item": "Canvas activities"
        },
        {
          "attendance": "4",
          "item": "Support sessions"
        },
        {
```

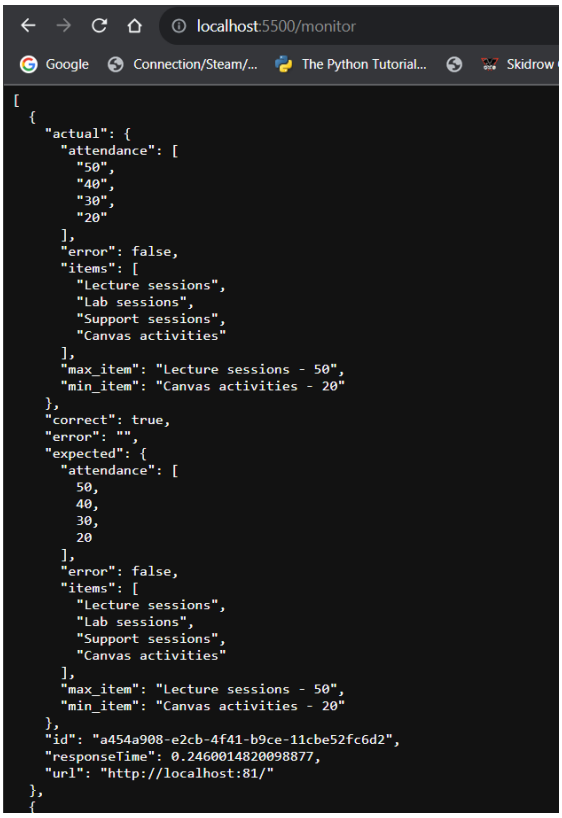

This is under the check_service/monitor parameter – where it is able to access a test case for each specific service.


```
  },
  {
    "actual": {
      "attendance": [
        10,
        20,
        30,
        10
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "total": 70
    },
    "correct": true,
    "error": "",
    "expected": {
      "attendance": [
        10,
        20,
        30,
        10
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "total": 70
    },
    "id": "c14da9ad-5cd7-4ffc-889a-7910f1e79fab",
    "responseTime": 0.007104396820068359,
    "url": "http://totalhours:80/"
  },
```



```
  {
    "actual": 0.3,
    "correct": true,
    "error": "",
    "expected": 0.3,
    "id": "96fa08e2-ccd2-44b9-bde8-05c0ef643a02",
    "responseTime": 0.005766868591308594,
    "url": "http://engagement-score:80/EngagementScore?"
  },
  {
    "actual": "Low",
    "correct": true,
    "error": "",
    "expected": "Low",
    "id": "e938b9f4-3ef4-435b-82fd-05eace500d6f",
    "responseTime": 0.00574183464050293,
    "url": "http://failurerisk:80/RiskAssessment?"
  },
  {
    "actual": {
      "average": 20
    },
    "correct": true,
    "error": "",
    "expected": {
      "average": 20
    },
    "id": "3c6f6e75-9767-4386-a3bc-65c104b237ca",
    "responseTime": 0.002405881881713867,
    "url": "http://qub-avg:80/average?"
  }
]
```

Where using the /monitor function
This following result will occur:



Where in this example, it gets the result of the localhost:81

As shown it gets an actual max min value of 50, 40, 30, 20 which responds to the “Lecture”, “Lab”, “Support”, “Canvas”

This then compares against the expected result within the expected block of attendances.

We can also see the “correct” is set to true, so the actual and expected results are similar.

Shown towards to bottom of the response.
It also defines the “max_item” and “min_item” which are defined, “Lecture” and “Canvas” Which is true as defined in the values: 50, 20.

Therefore the max = Lecture and min = Canvas.

It additionally prints the response time, aswell as a specific URL id for each specific function.

The next function within the monitor is a print statement which looks as follows:

```
150  def print_result(result):
151      actual_text = result['actual']
152      expected_text = result['expected']
153      result_text = "True" if result['correct'] else "False"
154      print(f"Actual: {actual_text}, Expected: {expected_text}, Result: {result_text}")
```

These print both, the actual and expected result with the result.
I have an assert equals function, which is too long to show in the report unfortunately.

However its main functionality uses a try catch statement to check each specific service, in the case if it is a maximin, engagement score or such. It will assign the actual and expected results and if they do not match firstly, it will print an error, with a found result.

Since each service requires different parameters, and prints a different result, each outcome of an integer, array, string or anything else needs to be accounted for, which makes the function longer than initially wanted and complicated.

In the elif blacks, it will go through total or average, which will also specific the integer error special cases.

In the final if errors block, it will return the given service, with false response, and the given error associated.

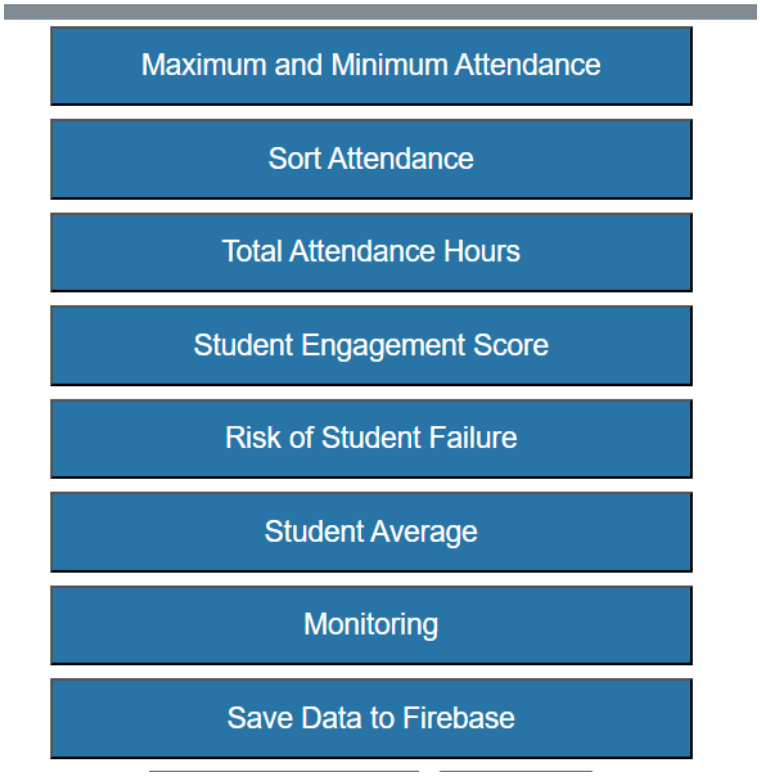
Next, I have this function.

```python
def record_result(service_url, test_result):
    with open('service_monitoring_log.txt', 'a') as file:
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        # Serialize actual response to JSON string if it's a dict
        response_text = json.dumps(test_result['actual']) if isinstance(test_result['actual'], dict) else str(test_result['actual'])
        status = "Passed" if test_result['correct'] else "Failed"
        file.write(f"{timestamp} - Test on {service_url} {status}. Response time: {test_result['responseTime']:.2f} seconds with response: {response_text}\n")
```

It will record the result to a text file, with the given result of a successful pass or a error.

It will record a .txt file called “service_monitoring_log.txt” which can be accessible and viewed separately.

Additionally, the contents of the monitoring can be viewed by clicking the monitor button on the front end code as shown within this screenshot, By clicking the monitor button.



It will prompt the user to this screen, shown below:

Here, we can see the /monitor section as shown previously within the window.

This time, it is accessible right from the front end index.html application.

Similarly again, it is all predetermined test cases, with a correct: true response.

Monitoring and Metrics

Monitoring Details


```
[
  {
    "actual": {
      "attendance": [
        "50",
        "40",
        "30",
        "20"
      ],
      "error": false,
      "items": [
        "Lecture sessions",
        "Lab sessions",
        "Support sessions",
        "Canvas activities"
      ],
      "max_item": "Lecture sessions - 50",
      "min_item": "Canvas activities - 20"
    },
    "correct": true,
    "error": "",
    "expected": {
      "attendance": [
        50,
        40,
```

We can also click the button at the bottom of the page as shown:


```
    },
    "id": "4f0c7319-15ac-4e4b-a77b-0514e2519c5f"
    "responseTime": 0.053116083145141 6,
    "url": "http://qub-avg:80/average?"
  }
]
```


Back to Main Page

As seen on the button “Back to Main Page” by clicking this button, it will bring us back to the front-end code of Student Engagement Monitoring.

```
273
274   def scheduled_monitoring():
275       print("Scheduled monitoring...")
276
277   scheduler = BackgroundScheduler()
278   scheduler.add_job(func=scheduled_monitoring, trigger="interval", minutes=1)
279   scheduler.start()
280
281   if __name__ == "__main__":
282       try:
283           app.run(host='0.0.0.0', port=5500, debug=True)
284       except (KeyboardInterrupt, SystemExit):
285           scheduler.shutdown()
286
```

Upon running the monitor.py code, it will run on port 5500 as shown within the __name__ try catch block, additionally it will try an interval limit of 1 minute scheduled monitoring segments each time.

So, for every minute passing, it will schedule a new monitoring session to ensure the monitoring continues to run throughout the application program execution.

Finally, we have the monitorTest.py

Here below is the monitorTest.py code result of a final passing result

```
C:\Users\Connor Mallon\Documents\MENG CSC\Level 3\CSC3065 I Cloud\A2\qub-monitor>python3 monitorTest.py
..Actual: {'some': 'data'}, Expected: {'error': False, 'items': ['Lecture sessions', 'Lab sessions', 'Support sessions', 'Canvas activities'], 'atten
dance': [50, 40, 30, 20], 'max_item': 'Lecture sessions - 50', 'min_item': 'Canvas activities - 20'}, Result: False
Actual: {'some': 'data'}, Expected: {'sorted_attendance': [{'item': 'Canvas activities', 'attendance': 5}, {'item': 'Support sessions', 'attendance':
 4}, {'item': 'Lab sessions', 'attendance': 3}, {'item': 'Lecture sessions', 'attendance': 2}]}, Result: False
Actual: {'some': 'data'}, Expected: {'error': False, 'items': ['Lecture sessions', 'Lab sessions', 'Support sessions', 'Canvas activities'], 'attenda
nce': [10, 20, 30, 10], 'total': 70}, Result: False
Actual: {'some': 'data'}, Expected: 0.3, Result: False
Actual: {'some': 'data'}, Expected: Low, Result: False
Actual: {'some': 'data'}, Expected: {'average': 20}, Result: False
.
----------------------------------------------------------------------
Ran 3 tests in 0.019s

OK
```

We can see the test had passed with a Result error: false, and a final OK result with the 3 test cases.

In the code, we have the following test cases:

40295919 > qub-monitor > monitorTest.py

```python
import unittest
from monitor import app
from unittest.mock import patch
import json

class TestMonitorService(unittest.TestCase):
    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True

    def test_check_service_valid(self):
        with patch('monitor.requests.get') as mock_get:
            mock_get.return_value.status_code = 200
            mock_get.return_value.json.return_value = {'some': 'data'}

            response = self.app.get('/check_service?service=http://maxmin:80/')
            self.assertEqual(response.status_code, 200)

    def test_check_service_invalid(self):
        # Test for the check_service route with an invalid service identifier
        response = self.app.get('/check_service?service=invalid_service')
        self.assertEqual(response.status_code, 404)

    def test_monitor(self):
        # Test for the monitor route
        with patch('monitor.monitor_service') as mock_monitor:
            mock_monitor.return_value = ({'some': 'data'}, 0.1)

            response = self.app.get('/monitor')
            self.assertEqual(response.status_code, 200)

if __name__ == '__main__':
    unittest.main()

```

It is a simple test case class which tests one existing service, it returns an expected 200 response to ensure it is up and running. The next test case is an invalid service, which is expected to return a 404 response.

Similarly, if the /monitor service is opertational, it wil assume a valid 200 response.

Since as shown previously, the three test cases have passed with an OK response so we can progress forward.

40295919
Connor Mallon

To connect all the services, I first created each specific service with image as shown here:
docker build -t maxmin-app .
docker build -t qubsort-app .
docker build -t totalhours-app .
docker build -t engagement-score-app .
docker build -t avg-app .

I then created a docker-compose.yml file which creates a multiple container instance within a compose instance. Which looks like the following:

Instead of creating each specific container instance, this file will create a compose instance which builds a compose project containing all the containers together, which makes life a lot easier.

This can be done easily by running the command “docker compose up -d”

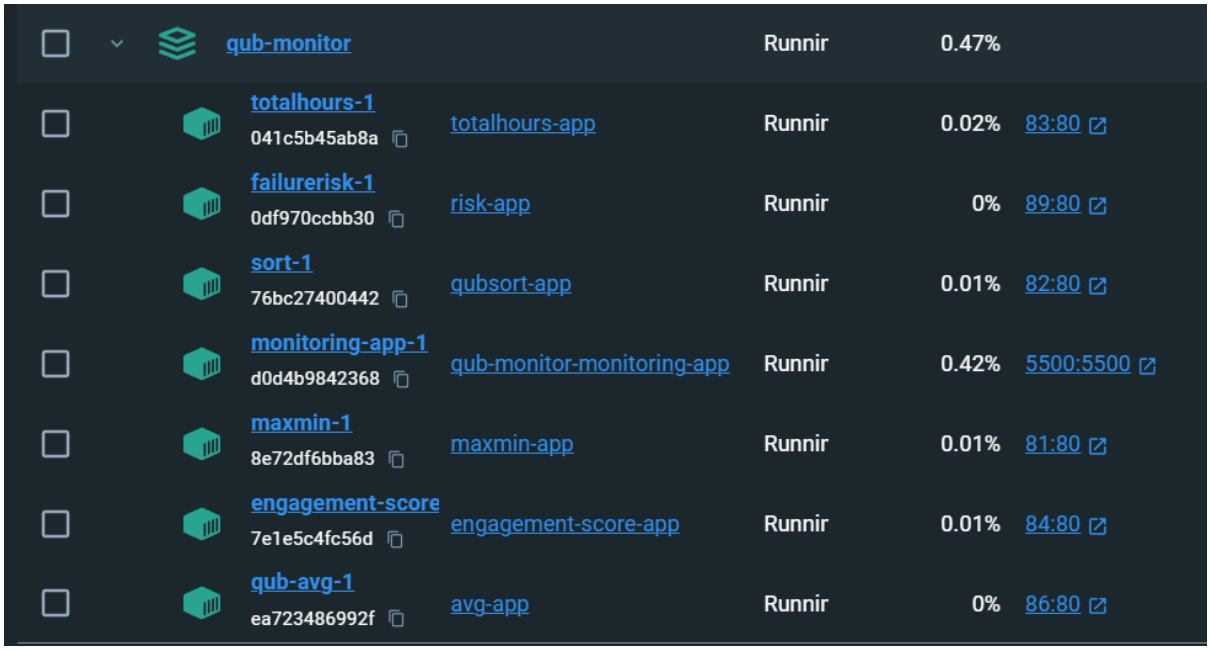
Where it will run in detached mode.

It will aditionaly connect each specific port for the given service, with the associated needed name.

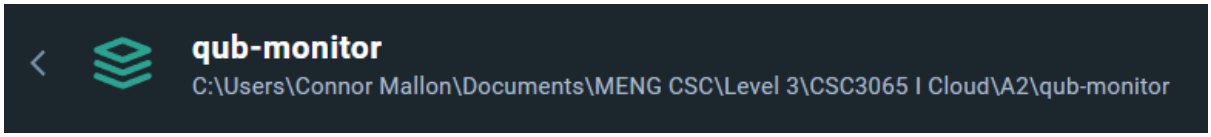
using this, we no lonher need to create a docker run command for each individual container.

We can just run by the docker compse and boots up the container using this file.

We can see the docker compose project up and running below:



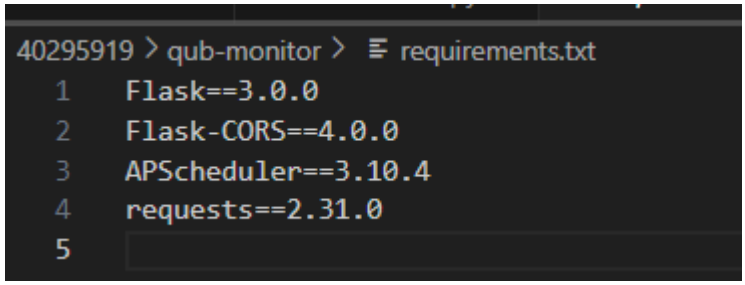
This makes the monitoring easier to done as we can ensure all the containers will boot up at the same time regardless. We can additionally access the individual service of maxmin, sort, total hours and so forth by directly accessing the port as normal.



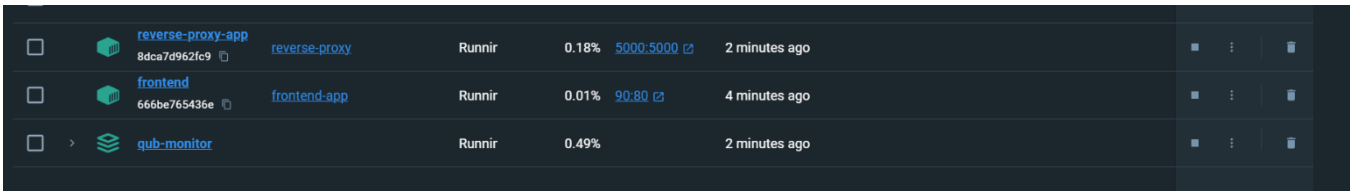
We can see the directory of where the qub-monitor compose project is being build shown above here.

I additionally have the following requirements.txt file for the given docker-compose.yml file:

Which imports the necessary dependencies for flask, CORS, APSchedular and Requests.



Upon finalising the code, we can see below, are the running containers in order for the program to be used and executed:



Where the qub-monitor holds all the separate services into one compose file, as shown previously, in the screenshot above.

The reverse-proxy-app is separate in the case that it is isolated away from all the other services.
The proxy can be accessible separately and allows for the services to be accessed through the proxy as well.

Additionally, the front-end application, in which the index.html is also running separately on port 90. Which allows to access the services in a front end like neat process.

This is how I am running the services throughout the application process.

40295919
Connor Mallon

- ## *Task F: Stateful Savings*

For the stateful savings, I have added a Firebase database which is a NoSQL database which handles everything to allow me to enter the specific details on the front end index.html, then the data will be stored directly into the Firebase database.

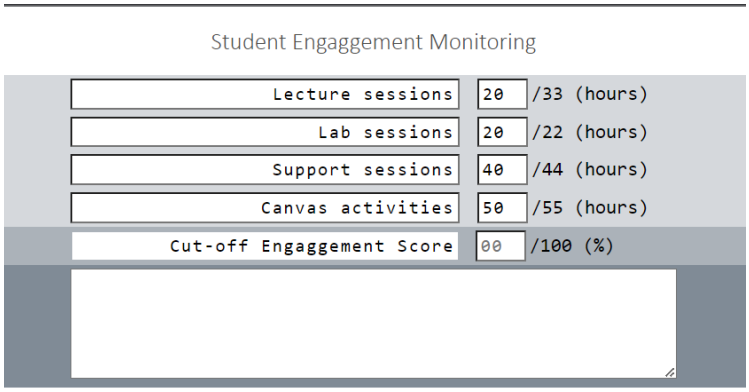
```
// Part F - config data for the firebase
const firebaseConfig = {
  apiKey: "AIzaSyAhfKbpI9-tfAwcPaz0YQIGm4SO6Jwtvsc",
  authDomain: "csc3065-a2.firebaseapp.com",
  projectId: "csc3065-a2",
  storageBucket: "csc3065-a2.appspot.com",
  messagingSenderId: "959619578762",
  appId: "1:959619578762:web:da777424c29c572c3e2673",
  measurementId: "G-SXR71YD5D6"
};
```

I have firstly taken the firebase_config shown here.

I have created a project on firebase called CSC3065 A2,

Firebase consists of a collection which contains a document specific to each user instance. In this case, I have a data-storage collection which holds all the user ID's, where each user ID can store the lecture, labs, support and Canvas activies.

If I were to input the following values into the index.html file as follows:



I have created an additional button shown here.

I have created this within the within the CSS section of the index.html code.

Where this links to the saveDataToFirebase() function and has the title "Save Data to Firebase."

```
<div>
    <button class="sembutton-active" onclick="saveDataToFirebase();">Save Data to Firebase</button>
</div>
<div>
    <input type="text" id="student-id" placeholder="Enter Student ID">
    <button onclick="getDataFromFirebase();">Retrieve Data</button>
</div>
```

Firstly I have entered my student number to: 40295919,
I would then click the saveData to Firebase button.

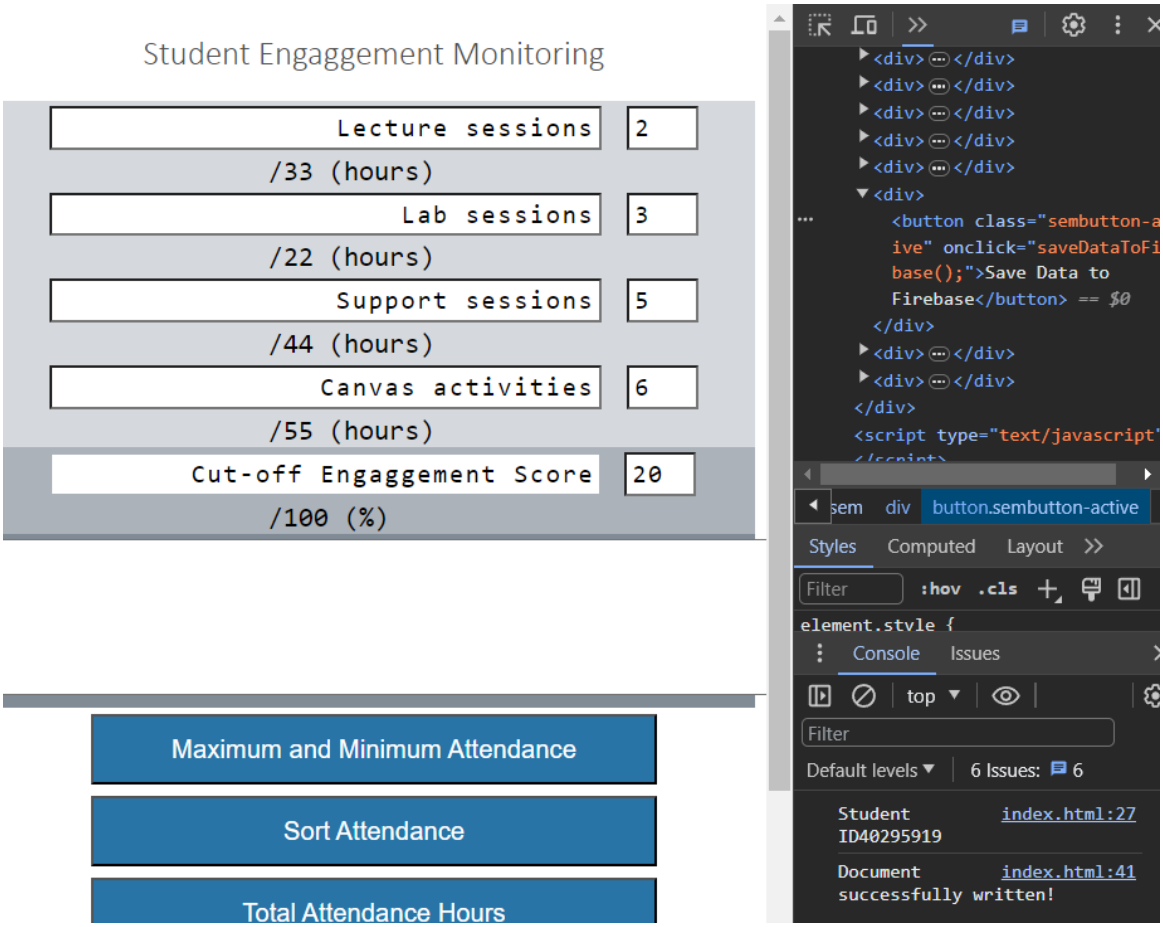
Which calls to the following function, shown within the code section above.
Within the saveDataToFirebase() function is the following:

```
function saveDataToFirebase() {
    const studentId = document.getElementById('student-id').value;
    console.log('Student ID'+studentId);
    const data = {
        attendance_1: document.getElementById('attendance_1').value,
        attendance_2: document.getElementById('attendance_2').value,
        attendance_3: document.getElementById('attendance_3').value,
        attendance_4: document.getElementById('attendance_4').value,
        index_1: document.getElementById('item_1').value,
        index_2: document.getElementById('item_2').value,
        index_3: document.getElementById('item_3').value,
        index_4: document.getElementById('item_4').value,
        engagementScore:  document.getElementById('cut-off').value,
    };
    db.collection("data-storage").doc(studentId).set(data)
    .then(() => {
        console.log("Document successfully written!");
    })
    .catch((error) => {
        console.error("Error writing document: ", error);
    });
}
```

This function will look quite similar to the getTotal(), getMaxmin(), getSort() and so forth functions.

It will additionally print a “Document successfully written” if the values are valid and the response is correct.

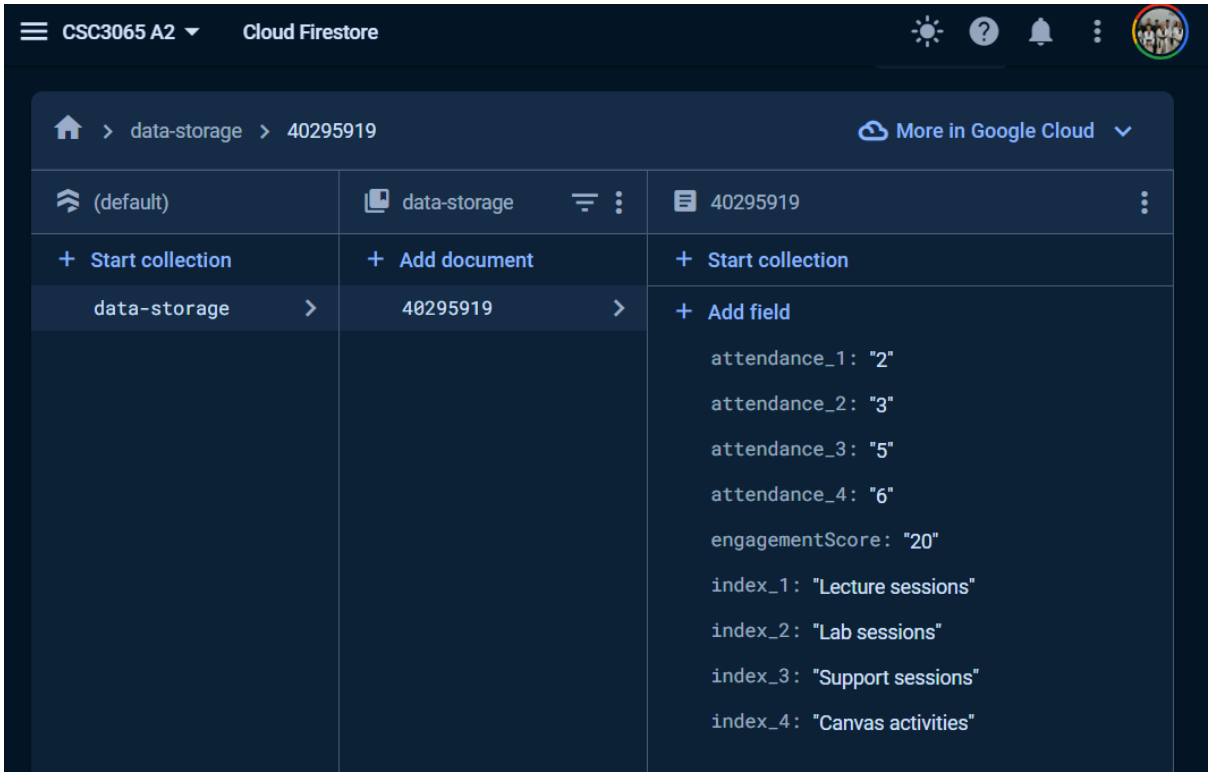
We can see an example of this below within the inspect element section:



We can see within the console of the last screenshot that the student ID: 40295919.

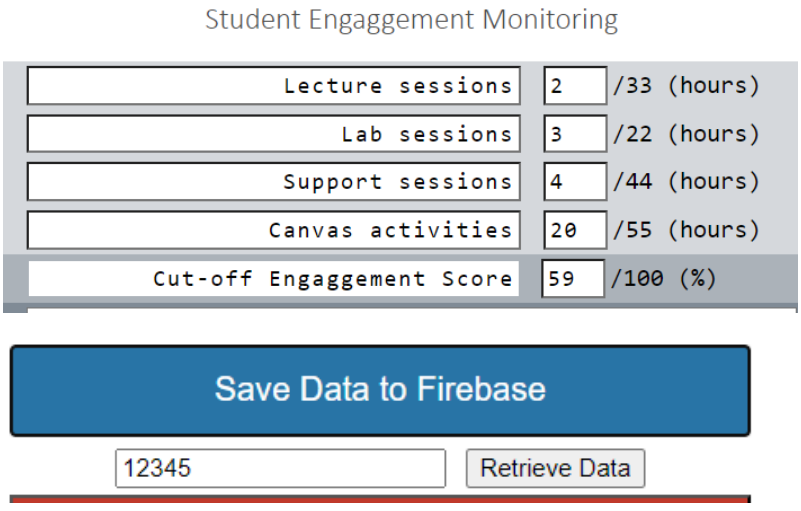
Additionally, we can see the document is successfully written with the attendances: 2,3,5,6, Aswell as the cut off engagement score is 20. We can also retrieve the cut-off engagement score.

It will display this format within the firebase database, as we can see below:



This is now working as shown, for each specific instance of each attendance.

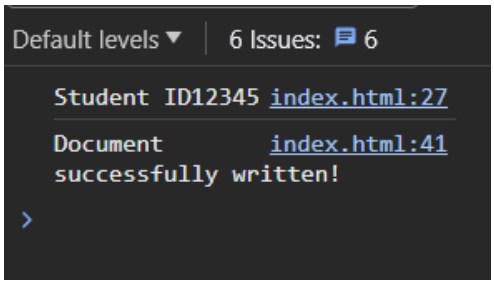
We can additionally test another case where ID is 12345 with the following values



This is my following attendances,

Shown below is the Student ID associated with the specific user.

When clicking the Save Data to Firebase we can see the following prompt.

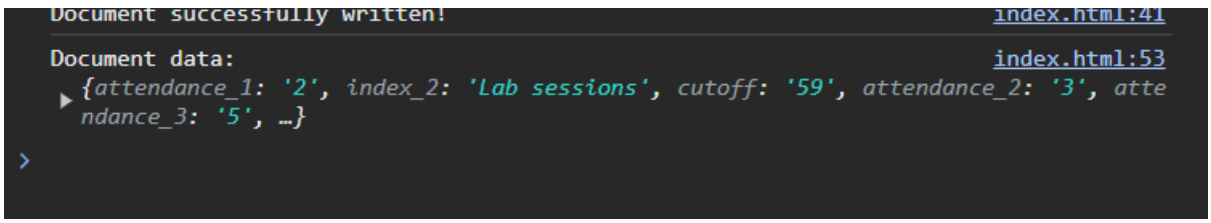


Then moving to the Firebase software.

It will add the new user ID 12345, to the existing data-storage collection, with the associated data.

Now we can see multiple instances of Student ID's can be added to the system.
In the case we want to retrieve a specific student attendance we can input the student ID and click the "Retrieve data" button shown here. In this case, I want to retrieve the data for Student ID: 40295919.

In turn it prints this result in the inspect element:



In order to get this result in the inspect element.

```
function getDataFromFirebase() {
    const studentId = document.getElementById('student-id').value;

    db.collection("data-storage").doc(studentId).get().then((doc) => {
        if (doc.exists) {
            const data = doc.data();
            console.log("Document data:", data);
            document.getElementById('attendance_1').value = data.attendance_1;
            document.getElementById('attendance_2').value = data.attendance_2;
            document.getElementById('attendance_3').value =  data.attendance_3;
            document.getElementById('attendance_4').value = data.attendance_4;
            document.getElementById('item_1').value = data.item_1;
            document.getElementById('item_2').value = data.item_2;
            document.getElementById('item_3').value = data.item_3;
            document.getElementById('item_4').value = data.item_4;
            document.getElementById('cut-off').value = data.cutoff;
        } else {
            // doc.data() will be undefined in this case
            console.log("No such document!");
        }
    }).catch((error) => {
        console.log("Error getting document:", error);
    });
}
```

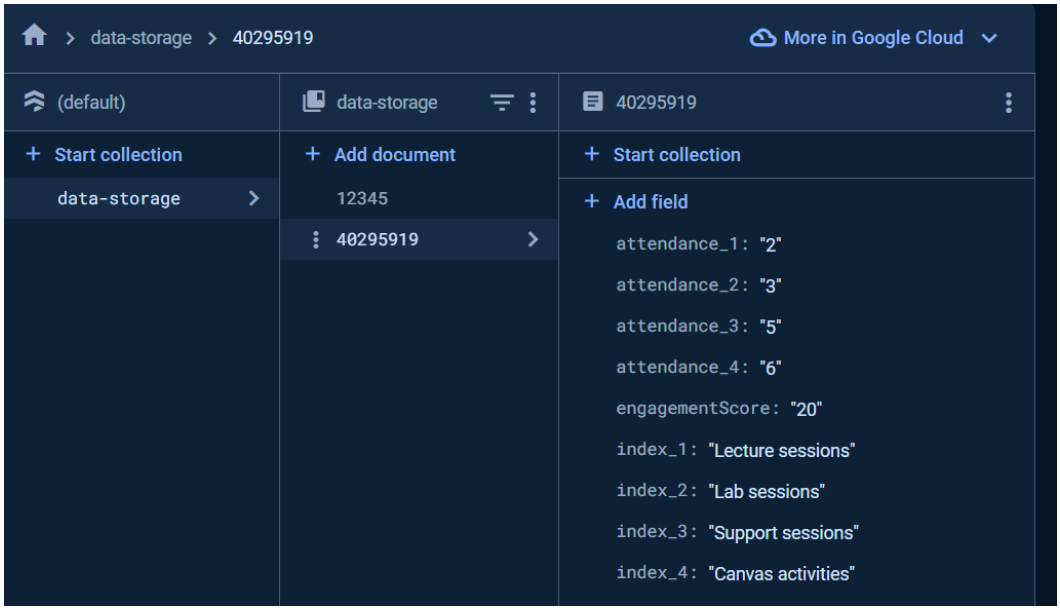
I created the following function called getDataFromFirebase()

This will retrieve any existing data attendances from the “data-storage” collection on Firebase.

If no such collection data is found in the firebase system. It will execute the else block,
Or print the error “Error getting document”.

In a successful retrieval of the firebase data attendances. It will parse the attendance, as well as the cut off and print the associated attendance within the inspect element section. We use this function to print to the console when inspecting the element section.

Next,
We can verify the element values by going to the Firebase Database of the student ID:



We can see the retrieved values are the same as the firebase values.

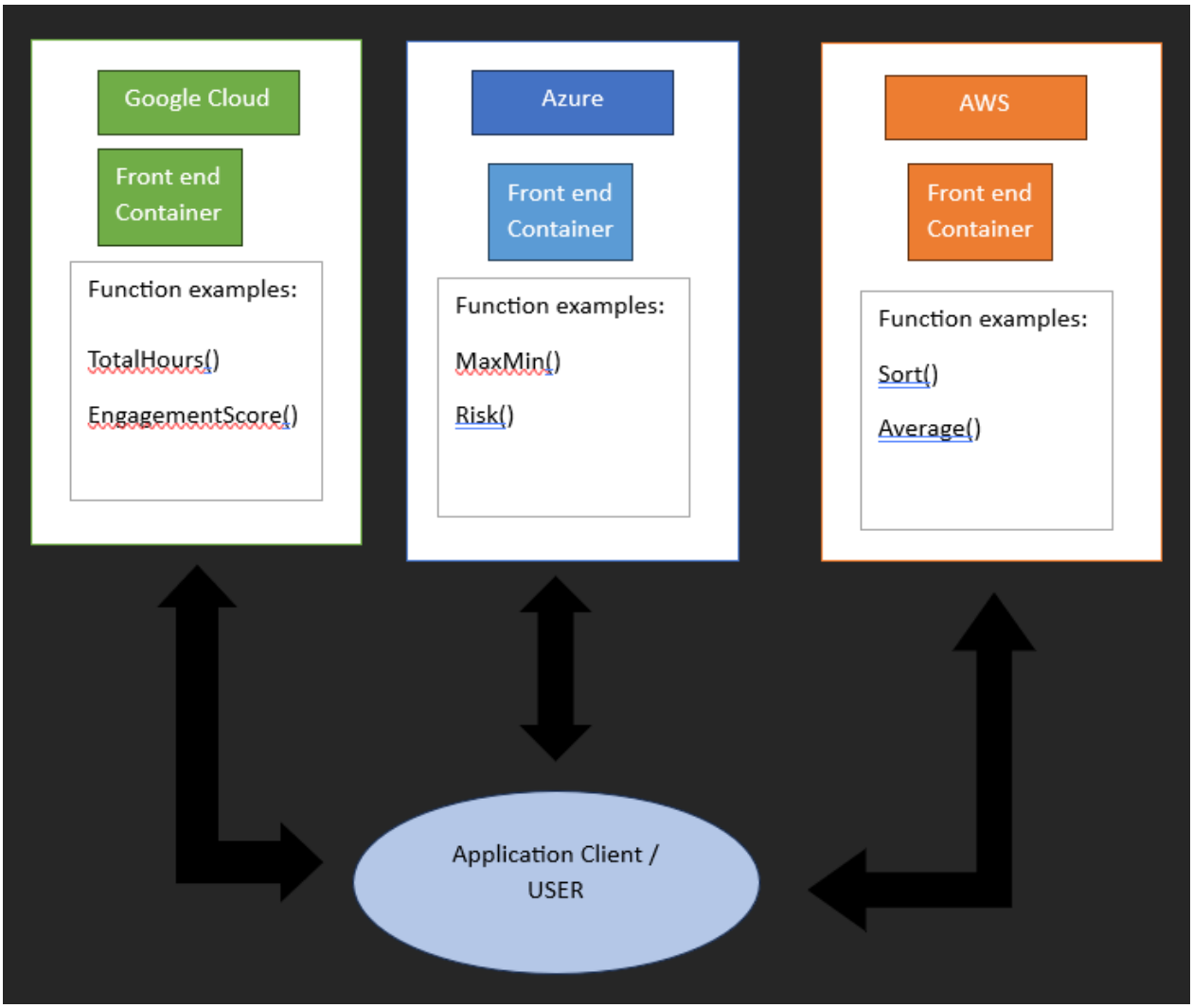
So can verify that the data input can be stored to the firebase database.

Additionally, for each specific Student ID, we can retrieve each attendance for each Student ID.

40295919
Connor Mallon

Task G: Design

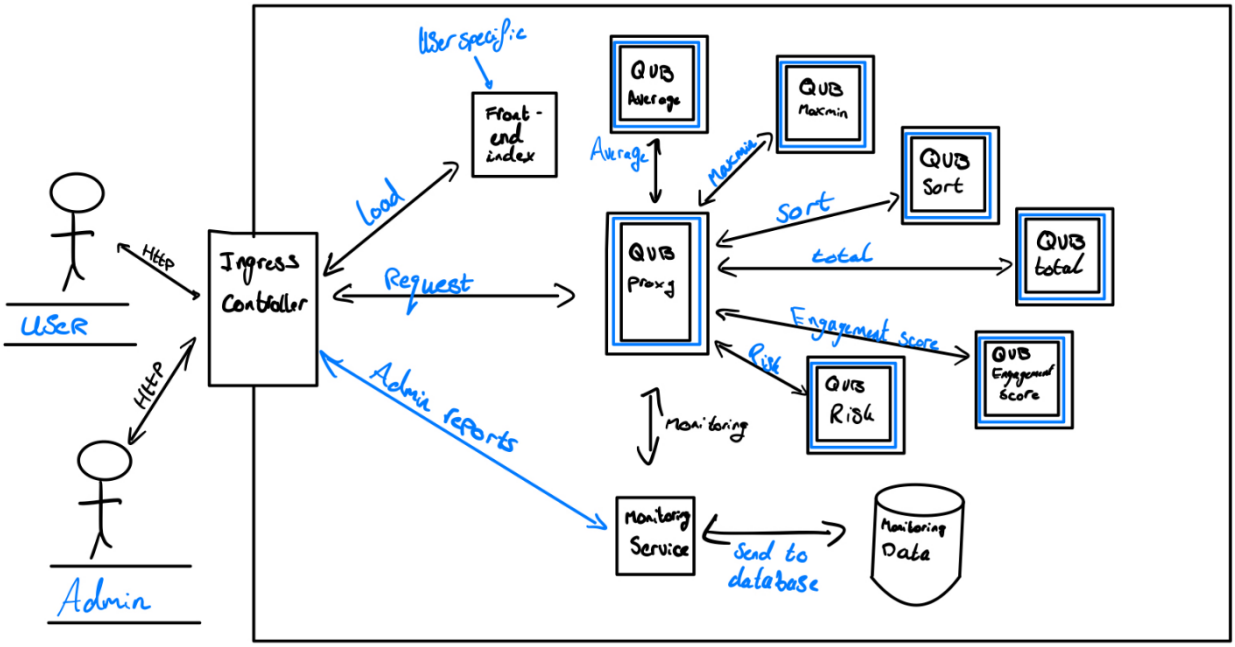
Here is an example where a user can access multie containers from different services. For example, The user is able to access the total hours and engagement score from one vendor



service, while also being able to access the Maxmin and Risk from the Azure vendor service.

In this example, say we have 3 mutli-vendor architecture, such as Google Cloud, Microsoft Azure and AWS.
The application Client is able to use the services from all the given platforms simultaneously.

40295919
Connor Mallon



Here is another example where a user is able to access the microservices through a reverse proxy example, Where the User entity is able to access the front end index.html interface.

However only the admin is able to access the monitoring and then feed any histories into the database to be stored and retrieved as needed by the admin users.
The user entity is only able to access the QUB average, QUB maxmin, QUB sort, QUB total, QUB engagement-score, QUB Risk.

These microservices are highlighted with a blue outline as shown in the image above.

In this case, the proxy is seen as a “middle man” where each service is only accessible through the proxy address.

The proxy in this instance will handle the responses, then forward the traffic to the specific request to where it is needed.

If a malious user where to additionally come in, the requests would then be forwarded to an invalid requests which prevents the malious user from interfering with important or vulnerable data. The proxy is an essential part of the program in this case which provides another level of security.

The monitoring service additionally allows to send data to the firebase system as defined in Section: F of the program. Where it can be sent and retrieved as necessary. Where it is only accessible by the Admin. The monitor will monitor the health of the system, as well as schedule intervals of each services current progress. The Data can be retrieved by the admin when needed.

40295919
Connor Mallon

Acknowledgements:

In conclusion, the assistance of Generative AI tools, as well as Stack Overflow as aided in the result of the working, finalised report documentation.

For section A, Gratitude to Python, C#, Node.js and Go, In order to create the separate features. Regarding the section: A, I
I have used ChatGPT 4 in order to create the .gitlab-ci.yml files.
I have copied the necessary files of the python C# code, and so forth into ChatGPT in order to create the .gitlab file.

In the initial stages of developing the new functionalities for the container services, I consulted ChatGPT for the design of a Python function capable of summing arguments. This led to the creation of a total function in Python, which was a pivotal element in my Flask application. ChatGPT's guidance was instrumental in implementing efficient error handling within the Flask app, ensuring robust response mechanisms for both correct and incorrect user inputs.

In example: I have copied the app.py code from the total microservice. I have additionally copied the test case class called "test.py" which is the given test cases for the original source code.

The provided .gitlab-ci.yml file included a single test case and the necessary code to pass for a CI pipeline test on GitLab.
It not only suggested relevant test cases for the Flask application but also guided me through the creation of a .gitlab-ci.yml file for GitLab's CI pipeline. This dual approach ensured a thorough testing process, contributing significantly to the project's stability.

For some of the initial pipelines had failed, I have used ChatGPT to break down the error issue shown on the GitLab repository. Then I would ask for a solution to speed up the process of completing the assignment. In this case ChatGPT 4 has sped up the error handling process. Which allows for more time to move onto the next part of the assignment.

For section: B, enhancing the frontend error handling, ChatGPT provided guidance on implementing 'readonly' attributes in CSS to prevent user alteration of front-end titles. This advice was crucial in maintaining the integrity of user inputs and improving the user experience.

In developing the MaxMinTest.php, ChatGPT's insights were invaluable. It helped in formulating test cases like testNormalCase(), testEmptyArrays(), and others, enhancing the thoroughness of testing. Additionally, its guidance on setting up a .gitlab-ci.yml file facilitated the establishment of an effective CI pipeline.

For section C. I have used ChatGPT 4 to create a Proxy code base. It provided the necessary tools which allows for the CORS features to operate and allow for a successful connection to the direct Service such as engagement-score and total hours.

For handling concurrent requests, ChatGPT's recommendation to use threading.lock() was crucial. This ensured that the proxy could manage multiple simultaneous requests without performance degradation or data inconsistency.

40295919
Connor Mallon

In testing the reverse proxy, ChatGPT's role was pivotal in designing the test cases in testProxy.py. It helped formulate scenarios that tested the proxy's ability to redirect traffic to the correct services and handle various types of requests, ensuring the robustness of the proxy implementation.

For Section D. I asked ChatGPT how to change the front-end index.html code in order to access multiple URLs such as the direct :81 URL for the maxmin, as well as the proxy URL. Where it returned the tryURL function, where through trial and error, I had asked multiple queries in order to get the tryURL function to it's current state, where it is now able to process string integer, array and any other form of parsed parameter type.

For section E, In the development of the monitoring and metrics service for our project, specifically the monitor.py service, the insights and guidance from ChatGPT were integral. ChatGPT provided foundational advice on how to effectively call upon Docker container files, set up predefined values for testing, and utilize assert_equals functions to accurately compare actual and expected values. This was critical in establishing a robust and reliable monitoring system.

ChatGPT also played a pivotal role in designing specific monitoring functions for each service, including maxmin, sort, total hours, and engagement score. The AI's expertise was particularly beneficial in handling predetermined inputs and expected responses, which ensured consistent and reliable monitoring outcomes. Although we encountered challenges in dynamically assigning actual and expected results for test cases, ChatGPT's recommendations aided in formulating a strategy that relied on a consistent set of actual and expected values, thus enhancing the reliability of our monitoring service.

The development of the check_service function, responsible for checking the health and functionality of each service, also benefitted greatly from ChatGPT's assistance. The AI provided valuable advice on implementing service-specific checks, particularly in handling response times and status, to ensure all services were functioning as expected.

For the monitor() function, which simultaneously tests all services, ChatGPT's input on structuring JSON responses and integrating the function with the frontend was crucial. This enabled direct access to the monitoring service from the frontend application, significantly enhancing user interaction and experience.

ChatGPT's suggestions were key in implementing effective error handling and logging within the monitoring service. Its guidance on creating a service_monitoring_log.txt file to record results and errors was instrumental in maintaining a comprehensive log of service performance.

The testing of the monitoring service, conducted through monitorTest.py, was another area where ChatGPT's role was significant. The AI helped formulate test cases that ensured the operational efficiency of the service, particularly in designing tests for existing services, invalid service responses, and the operational checks of the /monitor service.

Moreover, ChatGPT's assistance in containerizing the monitoring service and creating the docker-compose.yml file was foundational. Its advice on building and deploying multiple container instances within a compose project streamlined the entire process, making it more efficient and manageable.

Lastly, for the continuous monitoring and scheduled checks of services, ChatGPT's recommendations on using an interval limit and scheduling monitoring segments were invaluable. This ensured that the monitoring service would run effectively throughout the application's execution, providing constant vigilance and performance checks.

For Section F of the project, I implemented a stateful saving feature using Firebase, a NoSQL database. This allowed for the storage and retrieval of session details, attendance hours, and results directly from the QUBengage App's frontend. The Firebase setup involved creating a data-storage collection for user IDs, each storing specific details like lecture, lab, support, and Canvas activity data.

The frontend integration in the index.html file included a "Save Data to Firebase" button, linked to a saveDataToFirebase() function. This function handled the input data, saving it to Firebase and confirming successful storage with a console message. Additionally, a getDataFromFirebase() function was implemented to enable users to retrieve their saved data by entering their student ID.

Through testing with various student IDs, the functionality was confirmed to be successful, with the Firebase console verifying the consistency of saved and retrieved data. This implementation not only ensured data persistence across sessions but also enhanced the app's user experience by providing efficient data management capabilities.