Developer’s Guide

Team 3 Kuttal

Terrell Dixon
Lindsey Dunkley
Megan Planchock
Declan Smith
Sam Stone

Notable Directories and Files

Extension

The extension folder is in the root directory of the project's repository. It holds all the files that run client-side within Visual Studio Code. It is largely written in JavaScript, and runs a React application using a webpack server configured in a webpack.config.js file. The React application is displayed within a side panel of Visual Studio Code upon the extension's activation. The extension.js file in the src folder is the location of the code that handles what occurs during activation and deactivation of the extension. This is also where the webview for the panel is created in the SessionWebviewViewProvider class, passing the script for the React app inside html code for the React application. This activate function is also where event handlers track lines of code and send them through a WebSocket to the database.

Figure 1: Extension directory

VM

The VM folder is in the root directory of the project's repository. It is built and run using Docker through a compose.yml file, and consists of four subparts that are run separately from the extension on a server. In our case, this was run on a VM. The largest reason for running these on a server was to host a database for all users of the system, to allow users to connect across computers, and to lighten the CPU and memory usage on a user's computer from building using Docker.

Figure 2: VM directory

Proxy

The proxy has two main functionalities. It moves traffic from http to https, and it serves different applications behind path names for reaching and interacting with the major Docker containers.

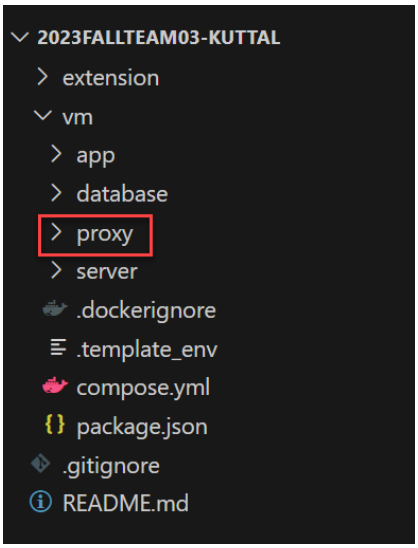

Figure 3: Proxy directory

Web App

The app is used for pairing two users, displaying the video call between them, and running facial recognition using a WebSocket connection to HUME AI. The src folder inside the app holds the pages and components that display within the React application for creating a session between two users and for displaying the video call. Notably, the App.js file contains the code for requesting access to the user's media streams and for switching between pages based on the status of the session.

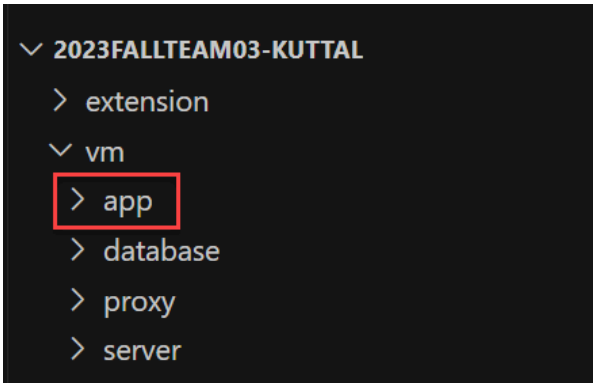

Figure 4: Web app directory

Emotions.js

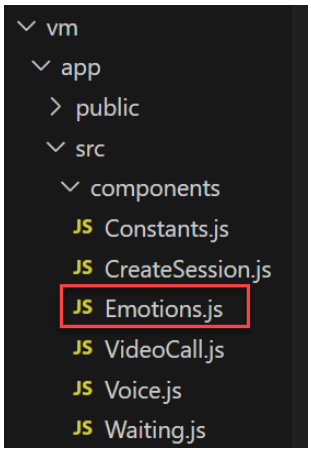

Figure 5: Emotion.js File

One notable file of the app directory running on the VM is Emotions.js. This component is passed a media stream from another component called VideoCall. It then takes images from the stream and passes them over a WebSocket to HUME AI to be processed before returning an emotion based on the user's expression.

Database

The database is used to store data for users and the session. This data includes speech-to-text utterances, facial expressions, lines of code, users, sessions, and the final report calculations for each user. The mongo-init.js file runs when the database is built for the first time and creates a database owner that can be used for creating collections and creating, editing, and deleting data from the database. Once the database is built, data is stored inside of a data folder within the database directory.

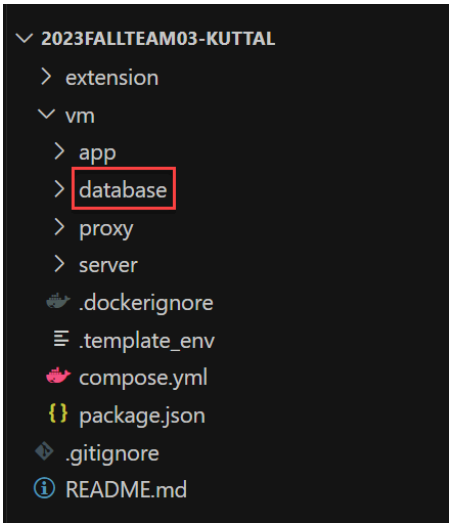

Figure 6: Database directory

Server

The server folder encompasses an express server with multiple functionalities. The server itself runs in the server.js file. Additionally, a PeerJS server runs from this file which is used for creating the call between the users behind the /myapp path (which is served by the proxy behind /webrtc/myapp).

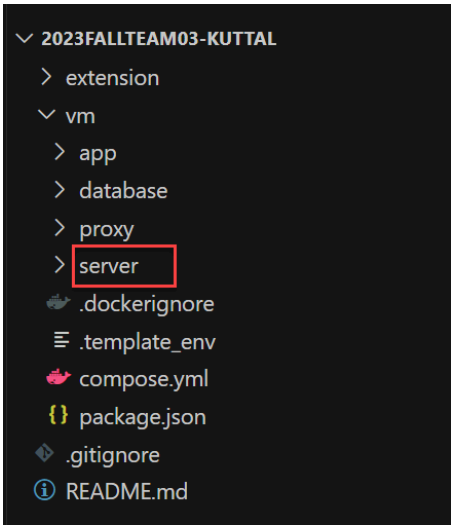

Figure 7: Server directory

APIs

The API endpoints are used to insert and retrieve data from the database. This data includes information about a session, users, utterances, and end reports. It is also used to calculate interruptions and delete utterances at the end of a session.

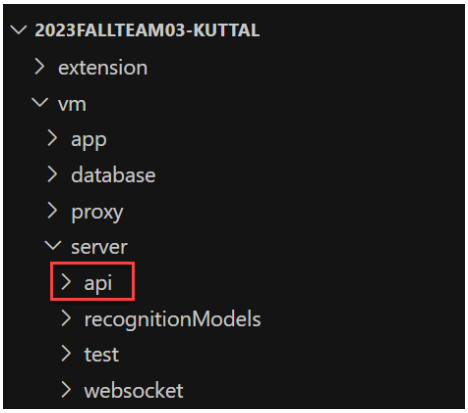

Figure 8: API directory

Websocket

The websocket folder contains a WebSocketRoutes.js file with two WebSocket routers on the paths ‘/ws’ and ‘/extension/ws.’ One is used for connecting the extension to handle various stages of the session from start to closing the end report. The other is used for connecting the

React app that runs on the VM to pair users in a session and to end the session between them simultaneously. Both handle sending a close message when one user ends a session to stop sending data and to display the end report to the user.

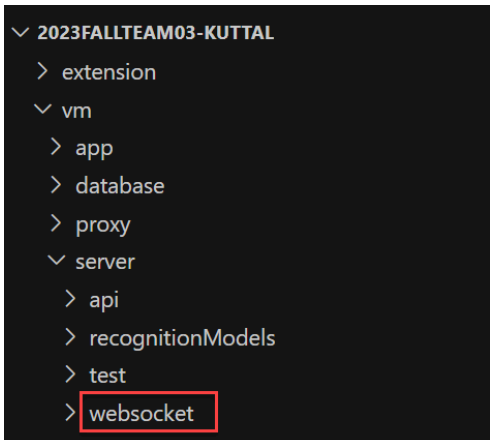

Figure 9: WebSocket directory

Recognition Models

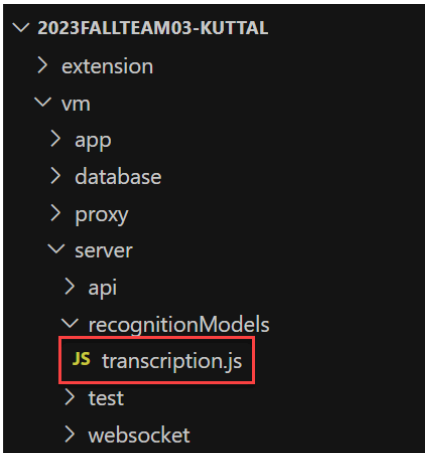

Figure 10: Transcription File

The transcription.js file is the only recognition model that runs within the express server. It runs a WebSocket connection to the app to receive audio data that can then be sent from the file running on node to handle the speech-to-text utterances.

Requirements

Due to the deadlines, evolving requirements, and technical setbacks, it became necessary to reduce the working scope of the project in order to still deliver a polished product. As such, there are certain features in the initial requirements that are still unfulfilled.

Implemented

- Video Call - A WebRTC peer-to-peer video call connected using PeerJS and an express server.
- Facial Expressions - The web application (that also hosts the video call) accesses the camera stream to record facial expressions. Frames from the video are processed using the Hume AI Facial Expression API to return a list of emotions and a confidence level. The most confident results are aggregated and recorded. After an interval, the score returns an emotion. That emotion is then categorized into score categories depending on if the emotion is positive, neutral, or negative and recorded. Self-efficacy is calculated based on how that score changes over time.
- Utterances - The web application accesses the microphone stream and uses Deepgram's Speech-to-Text API to create transcriptions of the user's utterances. The API can return a neutral timestamp of the user's transcriptions, which are used to determine interruptions. The text transcription may also be used when implementing rapport building.
- Lines of Code - The VSCode extension API provides listeners that will activate code for specific events, also returning information of the document. Lines of code provided for a user are calculated by subtracting the number of lines in the document before and after a change and adding that to the user's total.
- Pairing Sessions - Each of the web applications connect a WebSocket to an express server. They "register" their id with the server and whenever a user enters another user's id, the server ensures that the mentioned server has been registered. Assuming both users are stored, it pairs them and sends a message to start the session.

Not Implemented

- Role Switching - Due to misunderstandings with our sponsor, the specifics of this requirement changed quite a bit. The definition of the roles themselves changed from explicit to implicit and back to explicit roles. The "correct" length for an utterance or amount code to contribute in a pair programming session was also changing often. However, our lack of simultaneous editing and time meant this requirement was unable to be implemented.
- Simultaneous editing - In order to address the issues of the previous project, the focus for the first iterations of the project were mainly on integrating the recognition models and the video call. The method for implementing shared code editors was also undecided. Eventually LiveShare, an extension made by Microsoft that allows for simultaneous editing in groups, was found. However, LiveShare requires users to sign

into a Microsoft or VSCode account before use, which we were informed is not ideal for a research environment. By the time permission was given to implement, there was not enough time to add and debug LiveShare integrated within the custom extension.

- Rapport Building - Rapport building was originally out-of-scope for the project and was not included in the requirements. However, in the future this could be implemented using Deepgram. Deepgram includes functionality for topic detection and could be used to track if someone is talking about programming or not.

Installation

Mac

1. Install Visual Studio Code.
    a. Open the following link in a browser: https://code.visualstudio.com/download.

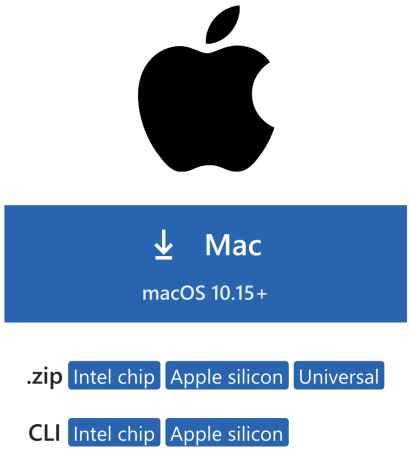

Figure 11: Mac installation window for VS Code

    b. Click the box labeled ‘Mac’ to download the zip file. Once the download is completed, open your Downloads folder and right-click on the file and select Open.

c. The application should then be extracted within the Downloads folder. To open it, right-click on the 'Visual Studio Code' application and select Open.

Figure 12: Mac VS Code Installation pop-up window

d. If the above message appears, select the 'Open' button. Visual Studio Code should now be running.

2. Install Node.
   a. Open the following link in your browser: https://nodejs.org/en/download
   b. Select the box labeled 'macOS Installer.'

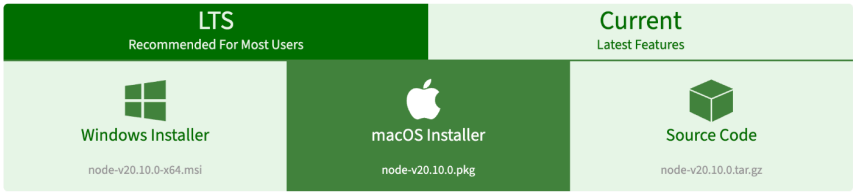

Figure 13: NodeJS installation page for Mac

   c. Once downloaded, right-click the node pkg file in your Downloads folder and select 'Open.'
   d. Select 'Continue' and 'Agree' throughout the installer when appropriate until the package is installed. Then, click 'Close' to end the installation process.
3. Install Docker Desktop.
   a. Open the following link in browser: https://www.docker.com/products/docker-desktop/
   b. Select the 'Download for Mac' option. This should match the Apple or Intel chip of your device.
      i. If unsure of your chip, select the Apple logo at the top-left corner of your screen and click 'About This Mac.'
   c. Once downloaded, right-click the Docker dmg file and select open. Once opened, drag the Docker icon into your Applications folder.
4. Install Git.
   a. First, open the "Terminal" application.
   b. Run the command 'git –version.' If a version is not returned, the terminal will prompt you to install git. Follow the instructions provided.

c. Install homebrew if you don't already have it by typing /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)" into the terminal
d. Next, type brew install git into the terminal to install git

Windows

1. Install Visual Studio Code.
   a. Open the following link in a browser: https://code.visualstudio.com/download.

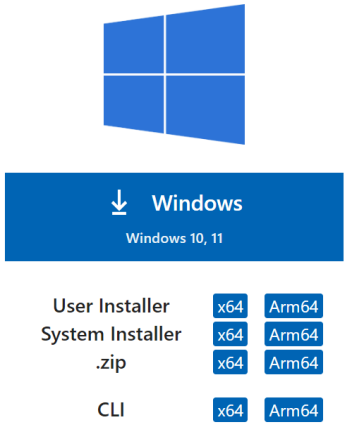

Figure 14: Windows installation window for VS Code

   b. Click the box labeled 'Windows' to download the zip file. Once the download is completed, open your Downloads folder and right-click on the file and select Open.
   c. The application should then be extracted within the Downloads folder. To open it, double click on the VSCodeUserSetup icon

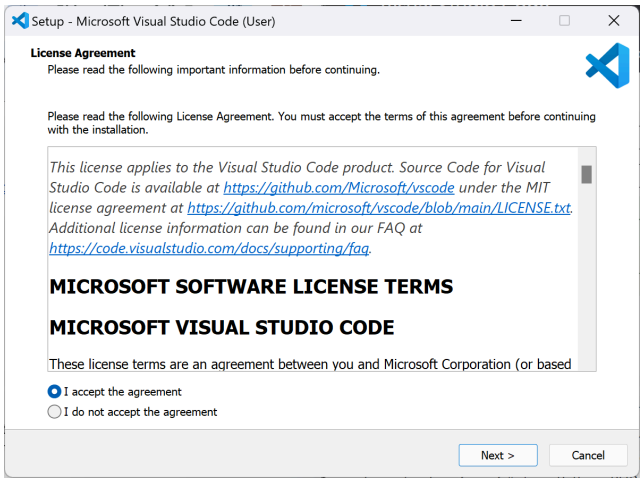

Figure 15: Windows pop up installation windows for VS Code

   d. Once the setup window opens, click "I accept the agreement" and click the Next button. Click the Next button again to accept the default VS Code configurations, and then click Install.

e. Once it has finished installing, search “VS Code” in your file explorer and double click it to run it.

2. Install Node.
   a. Open the following link in your browser: https://nodejs.org/en/download

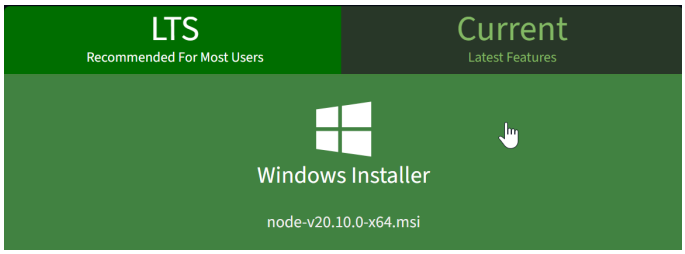

Figure 16: NodeJS Installation page for Windows

   b. Select the box labeled ‘Windows Installer’
   c. Once downloaded, right-click the node pkg file in your Downloads folder and select ‘Open.’
   d. Select ‘Continue’ and ‘Agree’ throughout the installer when appropriate until the package is installed. Then, click ‘Close’ to end the installation process.
3. Install Docker Desktop.
   a. Open the following link in browser: https://www.docker.com/products/docker-desktop/
   b. Click the “Download for Windows” button
   c. Once downloaded, right-click the Docker dmg file and select open. Once opened, drag the Docker icon into your Applications folder.
   d. On the Configuration page keep the default configuration and click “OK”. Docker Desktop will start installing.
   e. When the installation is successful, select Close to complete the installation process.

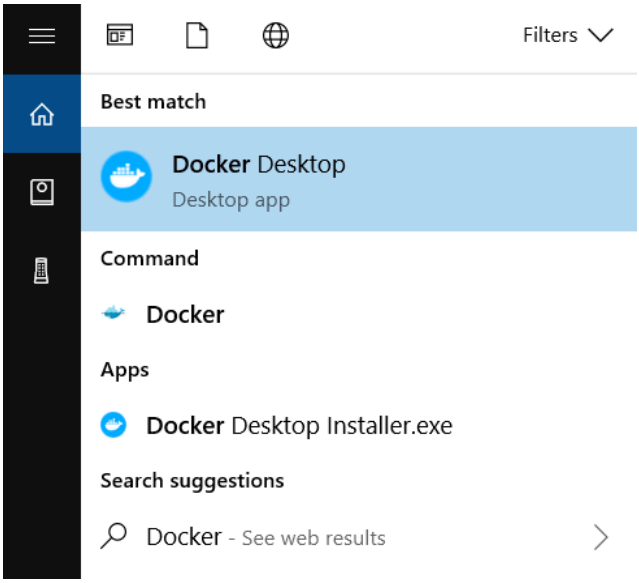

Figure 17: Docker Desktop in the File explorer

    f. Once it has finished installing, search “Docker Desktop” in your file explorer and double click it to run it.
    g. Select “Accept” on the Docker Subscription Service Agreement pop-up
4. Install Git.
    a. Go to the website: https://git-scm.com/download/win and select either “32-bit Git for Windows Setup” or “64-bit Git for Windows Portable” under the Standalone Installer section
    b. Install Git by double clicking on the executable file that was downloaded

Linux

1. Install Visual Studio Code.
    a. Go to the website: https://code.visualstudio.com/docs/setup/linux and follow the steps based on the Linux distribution you have.
2. Install Node.
    a. Go to the website: https://www.geeksforgeeks.org/installation-of-node-js-on-linux/# and follow the steps based on the Linux distribution you have.
3. Install Docker Desktop.
    a. Go to the website: https://docs.docker.com/desktop/install/linux-install/ and follow the steps based on the Linux distribution you have.
4. Install Git.
    a. In the terminal, if you’re not on a Debian-based distribution, run the command `sudo dnf install git-all`. Otherwise, run the command `sudo apt install git-all`

Running the Project Locally

1. Open Visual Studio Code.
2. Checkout the git repository.
    a. At the top of the screen, hover over ‘Terminal’ and select ‘New Terminal.’ If you have a preferred terminal, you can alternatively use that.
    b. From the terminal, use the cd command to reach the desired location for the project folder.
        i. For example, run the command ‘cd Project’ to reach the Project directory within your Desktop folder.
            c. Open the repository to be cloned in a browser.

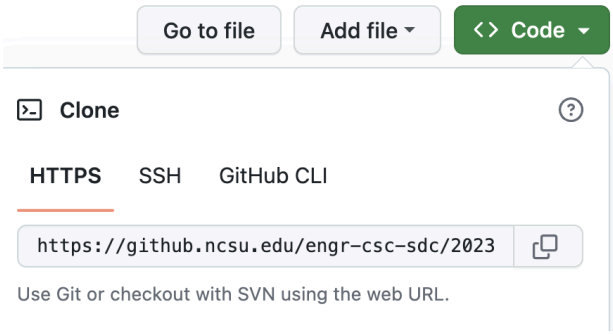

Figure 18: The window in Github to clone the repository

    d. Click the ‘<> Code’ button. Copy the link using the button below HTTPS next to the link provided.
    e. Go back to the terminal in Visual Studio Code, and run the following command:
        i. git clone link
            1. where link is replaced with the copied link from the repository
    f. Enter any GitHub information if prompted.
3. Build the VM folder.
    a. Open the ‘Docker’ application.
    b. Open the VM folder inside Visual Studio Code, or any preferred IDE that supports JavaScript.
    c. Set up the .env files.
        i. Create an .env in the vm folder using the vm folder’s .template_env file as a reference.
            1. Do not use the same username for the MONGO_INITDB_ROOT_USERNAME and the MONGO_USER. The MONGO_USER is the owner of the database created using Docker, which will be used for accessing the database to add, edit, and delete data.

2. vm > .env

```
MONGO_INITDB_ROOT_USERNAME=theMajorAdmin
MONGO_INITDB_ROOT_PASSWORD=password
MONGO_INITDB_DATABASE=pairProgrammingTool
MONGO_USER=admin
MONGO_PASSWORD=123
```

Figure 19: Sample VM ENV File

ii. Create an .env in the /vm/app folder using the vm folder's .template_env file as a reference.
   1. REACT_APP_HUME_API_KEY
      a. Go to the Hume AI website: https://hume.ai/ and click the Sign Up button
      b. Create an account with Hume AI
      c. Once you have an account and log in, go to https://beta.hume.ai/
      d. Go to the API Keys tab https://beta.hume.ai/settings/keys
      e. Copy the API key and paste it into the .env folder, setting it equal to the REACT_APP_HUME_API_KEY.
   2. The REACT_APP_HUME_ENDPOINT should equal 'wss://api.hume.ai/v0/stream/models.'
3. vm > app > .env

```
REACT_APP_HUME_API_KEY='1234567890abcdefghijklmnop'
REACT_APP_HUME_ENDPOINT='wss://api.hume.ai/v0/stream/models'
REACT_APP_API_URL='http://127.0.0.1/server/api'
REACT_APP_WEBSOCKET_URL='ws://localhost/server/ws'
REACT_APP_VOICE_WEBSOCKET_URL='ws://localhost/server/voice'
REACT_APP_PEER_HOST='localhost'
REACT_APP_PEER_PATH="/webrtc/myapp"
```

Figure 20: Sample VM App ENV file

   4. When running locally using Docker, Figure 19 is an example of a local VM .env file for the App folder. The only significant change should be the HUME API key, which will be equal to the one created for your specific account.

iii. Create an .env in the server folder using the server folder's .template_env file as a reference.
   1. The first three lines of this should match the last three lines of the .env file for the vm folder.
   2. API_URL should be API_URL='http://127.0.0.1/server/api'
   3. The MONGODB_URI should equal 'mongodb://db:27017/'
   4. To get an api key to set for DEEPGRAM_API_KEY, you will need to make a Deepgram account and generate a key.
      a. Go to deepgram.com and sign up for an account.

b. Once you make an account and login, go to the API Keys panel.
c. Click the Create New API Key button.
d. Set your desired name, permissions, and expiration settings, then click create key.
e. Copy the generated secret key and save it somewhere secure. This is your API key, and it is not saved by Deepgram so they cannot give it to you if you lose it. You will just need to generate a new one.

5.

Figure 21: Sample Server .ENV File

d. In your compose.yml inside the VM folder, alter line 8 from the following ‘./proxy/default.conf.template:/etc/nginx/templates/default.conf.template’ to ‘./proxy/local.conf.template:/etc/nginx/templates/default.conf.template.’
e. In the terminal in the project’s VM directory, run the following command: docker compose up --build.

4. Run the extension.
   a. Open the extension folder inside Visual Studio Code.
   b. Set up the .env file using the .template_env.
      i. The REACT_APP_WEBPAGE_URL will be set equal to ‘http://localhost.’
      ii. The REACT_APP_WEBSOCKET_URL will be set equal to ‘ws://localhost.’
   c. Edit the extension.js file link.
      i. In the extension.js file, navigate to the line where a websocket is connected, in the connect(ws) function (currently line 24). Change the line to connect to the localhost, in the form of ‘ws://localhost/server/extension/ws.’
   d. Run the build task.
      i. Hit Control+Shift+P (Command+Shift+P on Mac) to open the VS Code command palette.
      ii. Select “Run Build Task”
      iii. Select “npm: watch - extension”
      iv. Ensure extension.js is open in the active editor window, and press f5 (fn+f5 on Mac) to open the extension development host in a new window.

Running All Tests

1. Setup dependencies.
    a. Install development tools and dependencies on a given operating system.
        i. Node, npm
        ii. Docker
        iii. Visual Studio Code
        iv. npm install (extension)
    b. Configure the VM folder.
        i. Build the web application, server, proxy, and database using Docker.
    c. Install dependencies and build the extension.
2. Run the extension tests.
    a. Navigate to extension folder
        i. Run npm install
        ii. Run npm test
            1. To view the full coverage report run npm test - -coverage
3. Run the app tests.
    a. Navigate to the directory hosting the web application
    b. Access the vm/app folder
        i. Run npm install
        ii. Run npm test
            1. To view the full coverage report run npm test - -coverage
4. Run the API endpoint tests.
    c. Navigate to the directory hosting the web application
    d. Access the vm/server/test folder
        i. Run npm install
        ii. Run npm test
            1. To view the full coverage report run npm test –coverage

Common Development Use Cases

API Endpoints

Calling the API Endpoints

Our implementation uses axios to call the API endpoints. Axios is a promise-based HTTP client for NodeJS and the browser.

Documentation: https://axios-http.com/docs/intro

```
try {
  axios.put(`${process.env.REACT_APP_API_URL}/users/${id}/expressionScore/${score}`)
    .then(() => {
      console.log("updates emotion score");
    })
    .catch((error) => {
      console.error("Error in axios.put:", error);
    });
} catch (error) {
  console.error("Error in axios.put:", error);
}
```

Figure 22: Example of Axios PUT Request

```
React.useEffect(() => {
    const fetchData = async () => {
        try {
            await axios.post(`${process.env.REACT_APP_API_URL}/sessions/${userId}/${partnerId}`);
            setIsInDatabase(true);
        } catch (error) {
            console.log(error);
        }
    };
    if (isPaired){
        fetchData();
    }
}, [isPaired]);
```

Figure 23: Example of Axios POST Request

The current API endpoints can be found here: **https://github.ncsu.edu/engr-csc-sdc/2023FallTeam03-Kuttal/blob/main/vm/server/api/APIRoutes.js**. You can use this documentation as a reference for what to put in the header or body when calling an endpoint.

Creating New Collections

We are using MongoDB for our database, which is a NoSQL database. This means that the database does not require relational tables, which makes it more flexible to insert different data. In NoSQL databases, collections are what are the most akin to tables.
We are using Mongoose to handle the collections. Mongoose is a Object Document Mapping framework which makes it easier to interact with the collections as objects instead of raw data. This also makes it easier to define schemas.

The Mongoose models represent a collection schema. The location of the current model classes can be found in Figure 24 below.

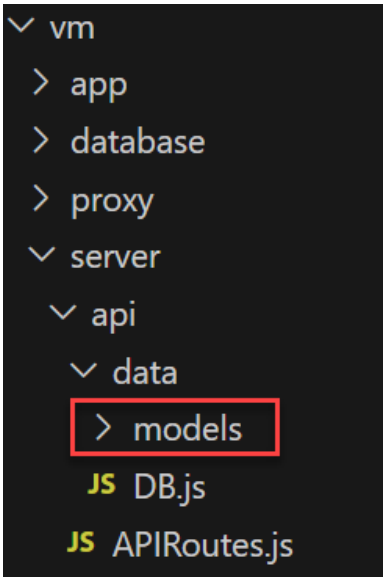

Figure 24: Mongoose models directory

Figure **25** is an example of what a schema looks like. You can build your own collection schema in a similar way. For more information on creating schemas, you can refer to the Mongoose documentation (https://mongoosejs.com/docs/guide.html). If you want to associate a collection with a specific user, make sure to include a field for the user id.

Report.js

vm > server > api > data > models > Report.js > ...

```
1   const mongoose = require('mongoose');
2   const { Schema } = mongoose;
3
4   const reportSchema = new Schema({
5       user_id: String,
6       primary_communication: String,
7       leadership_style: String,
8       communication_style: String,
9       self_efficacy_level: String
10  });
11
12  module.exports = mongoose.model("Report", reportSchema);
```

Figure 25: Example of a Mongoose Schema

To use the collections, navigate to the APIRoutes.js file as shown in Figure **26**.

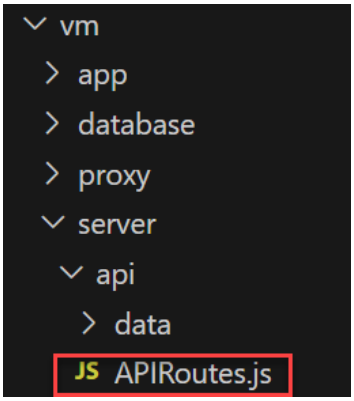

Figure 26: API routes file location

```
// Insert a report
apiRouter.post("/reports", async (req, res) => {
    const report = new Report({
        user_id: req.body.user_id,
        primary_communication: req.body.primary_communication,
        leadership_style: req.body.leadership_style,
        communication_style: req.body.communication_style,
        self_efficacy_level: req.body.self_efficacy_level
    });
    try {
        const sessionUser = await Session.findOne({ $or: [{ user1_id: req.body.user_id }, { user2_id: req.body.user_id }] });

        if (!sessionUser) {
            return res.status(409).send("A session does not exist with these users");
        }

        await report.save();
        return res.send(report);
    } catch(err) {
        return res.status(500).send("Failed to insert Report");
    }
});
```

Figure 27: Example of using a Mongoose schema in an API endpoint

To use the endpoint in an API endpoint you just have to create a new instance of your schema class. Make sure that the names of the fields are the same as the ones you defined in the schema class you created. Then, you can directly insert it into the database with res.send(schemaObject).

Extending Fields In Existing Collections

If you'd like to add more fields to the existing collections (Report, Session, User, and Utterance) you can simply add a new field to its schema in the models directory (as shown in Figure: 21).

Make sure that you also change the API that POSTs that model into the database. For this, just add the new field where the schema object is being instantiated.

You also need to ensure that the places calling this endpoint are also changed. The current locations where POST endpoints are called are shown in the following figures.

POST Report

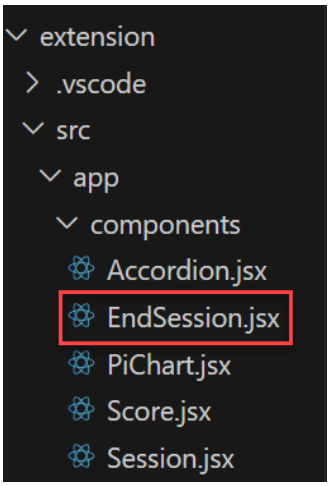

Figure 28: Directory where Reports are posted

```
19      try {
20        const report = {
21          user_id: userId,
22          primary_communication: "",
23          leadership_style: leadershipStyle,
24          communication_style: communicationStyle,
25          self_efficacy_level: selfEfficacy,
26        };
27
28        await axios.post(`${process.env.REACT_APP_WEBPAGE_URL}/server/api/reports`, report);
29        console.log('Report updated successfully');
```

Figure 29: Location in EndSession.jsx where the POST Report is called

POST Session

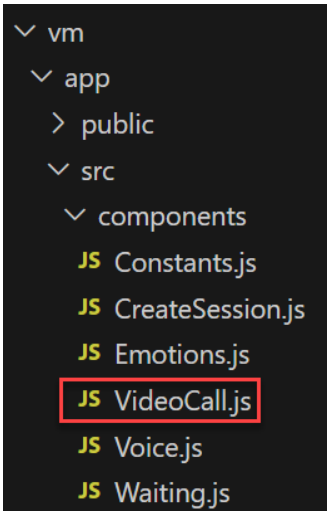

Figure 30: Directory where Sessions are posted

```
66              try {
67                  await axios.post(`${process.env.REACT_APP_API_URL}/sessions/${userId}/${partnerId}`);
```

Figure 26: Location in VideoCall.js where the POST Session endpoint is called

POST Utterances

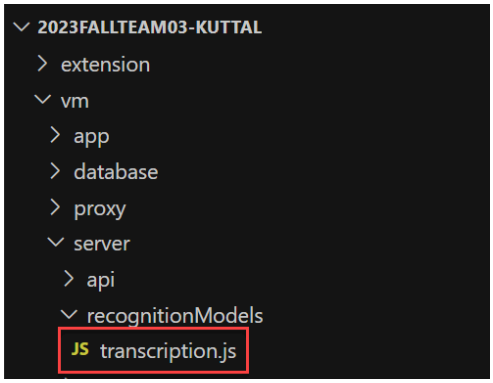

Figure 31: Directory where Utterances are posted

```
91              //if end of utterance, and currentUtterance has any data in it, post to database and reset
92              if(data.speech_final && currentUtterance.transcript.length > 0) {
93                  //post to database
94                  axios.post(process.env.API_URL + "/utterances", currentUtterance)
```

Figure 32: Location in transcription.js where the POST Utterance endpoint is called

POST User

User's are automatically posted to the database when the POST /sessions endpoint is called. If you'd like to change the User Schema navigate to the APIRoutes.js file. You can change the schema object's instantiation here.

```
14  // Insert a Session
15  apiRouter.post("/sessions/:user1_id/:user2_id", async (req, res) => {
16      const user1Id = req.params.user1_id;
17      const user2Id = req.params.user2_id;
18
19      const session = new Session({
20          user1_id: user1Id,
21          user2_id: user2Id
22      });
23
24      const user1 = new User({
25          user_id: user1Id,
26          lines_of_code: 0,
27          num_role_changes: 0,
28          expression_scores: [0],
29          num_interruptions: 0 ,
30          num_utterances: 0
31      });
32
33      const user2 = new User({
34          user_id: user2Id,
35          lines_of_code: 0,
36          num_role_changes: 0,
37          expression_scores: [0],
38          num_interruptions: 0,
39          num_utterances: 0
40      });
```

Figure 33: Location in APIRoutes.js where the User object is instantiated

Creating New API Endpoints

To create new API endpoints navigate to the APIRoutes.js file. The routes are created with an Express Router. The documentation for creating routes with Express can be found here: https://expressjs.com/en/guide/routing.html.

To create an endpoint with route parameters, add the name of the parameter to the route, preceded by a colon. To access the route parameters, use “req.params”. An example of using route parameters is shown in Figure 34.

```
// Insert a Session
apiRouter.post("/sessions/:user1_id/:user2_id", async (req, res) => {
    const user1Id = req.params.user1_id;
    const user2Id = req.params.user2_id;
```

Figure 34: Example of Using Route Parameters in an API Endpoint

To create an endpoint with a request body, you can access it by using “req.body”. An example of using a request body in an endpoint is shown in Figure 35.

```
// Insert a new utterance
apiRouter.post("/utterances", async (req, res) => {
    const utterance = new Utterance({
        user_id: req.body.user_id,
        start_time: req.body.start_time,
        end_time: req.body.end_time,
        transcript: req.body.transcript
    });
```

Figure 35: Example of Using a Request Body in an API Endpoint

The documentation for retrieving, inserting, deleting, and updating documents with MongoDB can be found here: https://www.mongodb.com/docs/mongodb-shell/crud/.

To send a response from an endpoint you can use “res.status(Int).send(“Message”)”. An example of this is shown in Figure 36.

```
if (!sessionUser) {
    return res.status(409).send("A session does not exist with these users");
}
```

Figure 36: Example of Sending a Response from an Endpoint

Testing API Endpoints

The location of the API tests are found in Figure 37. To run the tests, change directories into the test folder and run “npm run test” from the command line.

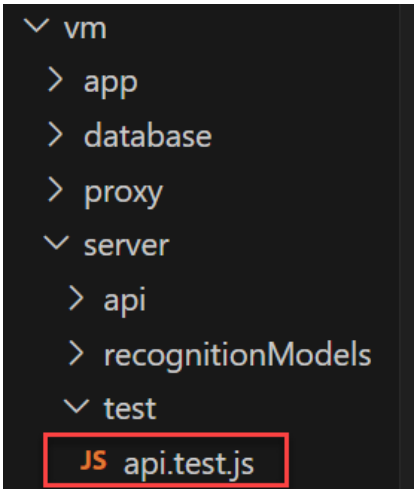

Figure 37: Location of the API Tests

All of the tests use a MongoDB in-memory server to simulate actual entries being retrieved and inserted into the database. The testing is done using Supertest to test the endpoints and routes. The documentation for Supertest is here: https://github.com/ladjs/supertest.

The current tests are broken into “describe()” blocks. These are used to help navigate between and combine similar tests. These tests are separated into testing the endpoints for each collection. There are describe blocks embedded within each and an it() block for the actual testing. The format for this is as follows:

describe(“What is being tested”) -> describe(“What the endpoint is doing) -> describe(“Given a condition”) -> it(“should have these results”).

An example of this is in Figure 38.

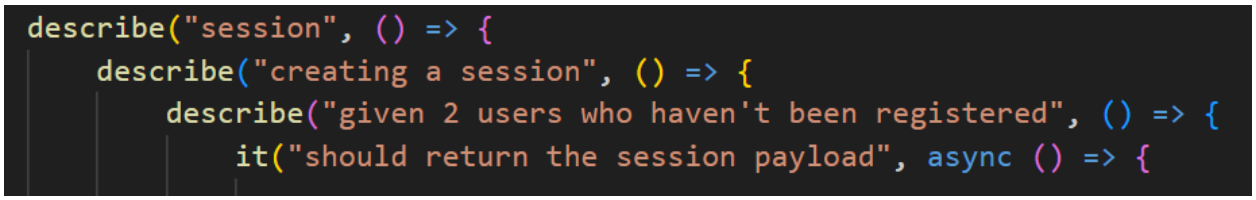

Figure 38: The Organization Structure of API Tests

To call an endpoint you use “supertest(app).requestType(‘/api/address/to/endpoint’);” To get the response, you can use res.body or res.text (depending on the endpoint). To test the response, you use expect(). In the supertest documentation there are a lot of examples on what assertions you can make. An example of a simple test case is in Figure 39.

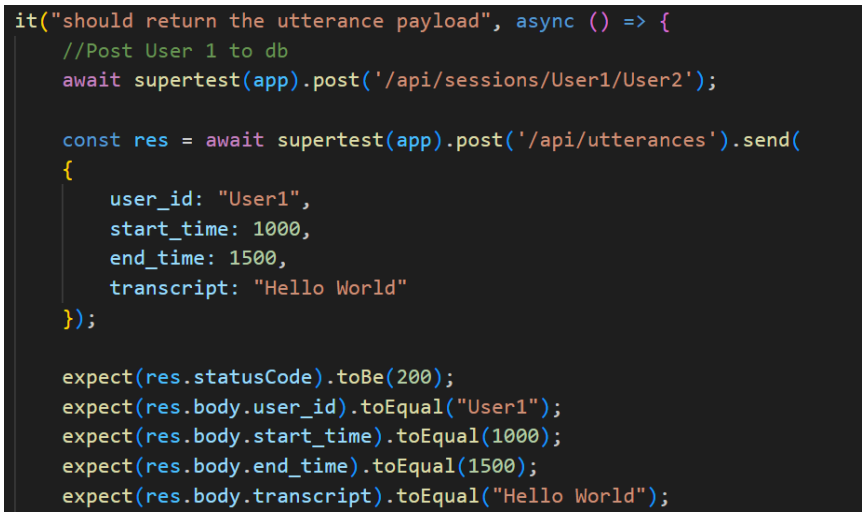

Figure 39: An Example Test for an API Endpoint

Notable Design and Implementation

VS Code Extension Limitations

VS Code extensions have a number of significant limitations, most notably they are strictly sandboxed and unable to access user media devices (which are crucial for this project). We wasted a lot of time designing and trying to implement an architecture with most of the features packaged directly in a React webview extension. We had to redesign the architecture to move the recognition models and video call to run in a web application that can access the microphone and camera through the browser.

Deepgram

Our implementation uses a MediaRecorder to send audio buffers over a websocket from the React web app to the Express server. On the express server, these buffers are forwarded to a connection to the Deepgram API. The response from deepgram is then formatted according to the Utterance schema in order to be posted to the database via an axios request. Our utterance schema a start time, end time, transcript and user id. Figure 40 shows an example transcript from Deepgram response.

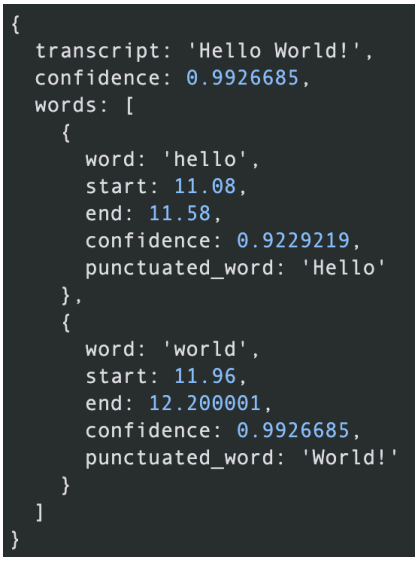

Figure 40: Example Deepgram Transcript.

In order to take these response and convert it into an utterances we need to synchronize the timestamps between clients by utilizing the timestamps in conjunction with an epoch based timestamp from the OS, utilize Deepgram's endpointing feature provide approximate utterance segmentation, and then its straightforward to extract the transcript and start/end times in order to construct an utterance.

```
{
  transcript: 'Hello World!',
  startTime: 1701701696.08,
  endTime: 170170168597.20,
  userId: User1234,
}
```

Figure 41: Example Utterance

Deepgram has documentation for its streaming api (https://developers.deepgram.com/docs/getting-started-with-live-streaming-audio), but the most useful thing we found for this project was actually this GitHub repo with an example of live streaming audio in Node.js (https://github.com/deepgram-devs/node-live-example).

Hume AI

Hume AI Streaming API documentation: https://dev.hume.ai/docs/streaming-api-tutorial

Hume AI is used for emotion detection to calculate self efficacy. Hume AI tracks 48 different emotions. Each of these emotions are given a score of either 1, -1, or 0 for either a positive, negative, or neutral emotion. The scoring of these emotions are in Constants.js, which can be found in Figure 42, and can be easily changed if you wish.

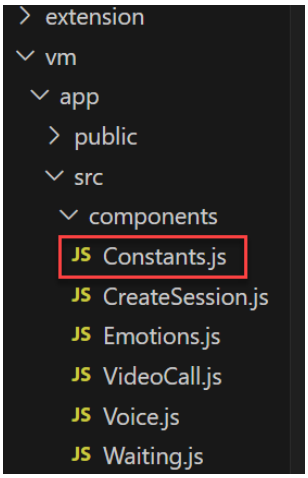

Figure 42: Location of File for Emotion Scoring

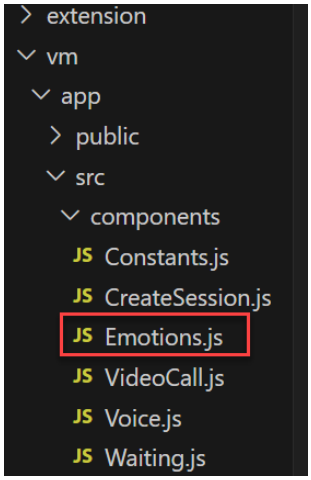

Figure 43: Location of File Where Hume is Run

Detecting the user's emotions are done in several steps. All of this functionality is in the Emotion.js file as shown below in Figure 43. First, a Web Socket is opened to the Hume streaming API. Every half a second a frame from the camera's Video Stream is sent to the API. The response is an array of emotions with an associated confidence score. Then, the emotion with the highest score is calculated, and given a positive/negative/neutral score based on Constants.js. These scores are tracked, and the average of them is accumulated on a 5 second basis. After 5 seconds, the average emotion score is sent to the user's User document in the database. Over the course of the session, the user will have an array of different emotion scores. The trend of emotions from positive to negative or negative to positive are calculated at the end session report.

API Endpoints

The API endpoints are used to insert, retrieve, remove, and update documents in the database as well as calculate interruptions. The documentation for all of the endpoints can be found here:https://github.ncsu.edu/engr-csc-sdc/2023FallTeam03-Kuttal/wiki/API-Endpoints. All of the endpoints are called using axios, which is described in detail in the API Endpoint section of the Common Use Cases section.

Final Report Calculations

Leadership Style [Authoritative or Democratic]

1. Retrieve the users lines of code contribution
2. Retrieve the partners lines of code contribution
3. If the total of the partners and the users lines of code is 0, or user lines of code is 50% ∓ 15% of the total between the partners than the leadership style is democratic
4. Otherwise it is an authoritative leadership style

Communication Style [Verbal or Non-Verbal]

1. Obtain User's Data
    - Retrieve the user's expression scores array.
    - Retrieve the user's utterances.
    - Retrieve the partner's utterances.

2. Calculate Non-Neutral Expression Percentage:
    - Determine the percentage of entries with non-zero values in the expression scores array out of the total number of entries.
    - This represents the proportion of the session dominated by non-neutral expressions.

3. Calculate User's Utterance Percentage:
    - Calculate the percentage of utterances spoken by the user out of the total utterances (sum of the user's utterances and the partner's utterances).

4. Determine Communication Style:
    - If the expression ratio (non-neutral expression percentage) is greater than 50% but the utterance ratio (user's spoken percentage) is also greater than 50%, categorize it accordingly.
    - If both the expression and utterance ratios are less than 50%, label it as nonverbal communication.
    - In all other cases, categorize it as verbal communication.

Self Efficacy [High or Low]:

1. Retrieve the expression scores array from the user.
2. Calculate the average for the first third of the array.
3. Compute the average for the remaining two-thirds of the array.
4. Determine if the average of the first third is greater than the average of the rest.
5. Classify the result as low self-efficacy if the condition is met; otherwise, categorize it as high self-efficacy.

Collaboration Score [0-10]

1. Calculate Utterances Ratio:
    - Find the ratio of the user's utterances to the total between the sum of the user's and partner's utterances.

2. Calculate Interruption Ratio:
    - Determine the ratio of the number of times the user interrupted their partner to the total interrupts between both contributors.

3. Calculate Code Contribution Ratio:
    - Compute the ratio of the lines of code contributed by the user to the total lines of code between both contributors.

4. Scale Ratios to 0-10
    - Apply a scaling formula to each ratio, where a ratio of 50% results in a score of 10, and ratios of 0% or 100% yield a score of 0.
    - Score = 10 - abs(ratio - .5) * .20

5. Compute Collaboration Score
    - Average the scores obtained from the utterances ratio, code contribution ratio, and interruption ratio to derive the overall collaboration score.

Extending the Application

New Features

Simultaneous Editing and Role Switching

The two requirements we did not complete are simultaneous editing and role switching. VS Code Live Share has potential for simultaneous editing. However, we were not able to get it to work, so be sure to plan for time to pursue other options. You may want to seriously consider looking into IDEs other than VS Code that provide more options then Live Share for simultaneous editing. There is a very real chance that Live Share is simply not an option as the API for accessing it from other extensions (https://www.npmjs.com/package/vsls) has not been updated in 2 years and we were fully unable to get it to import in our extension. Moving to an IDE other than VS Code would certainly be a big undertaking, but the majority of the code is actually not dependent on VS Code. The feedback in the extension is a React app which could be moved to the web application in the VM folder, so the only thing which would have to be fully redone is the tracking of lines of code, which would also need some modification if Live Share is implemented.

Role switching could be done through a button press to explicitly switch which partner is driving and which is navigating. However, the requirement was changed mid-semester to be automatically implicitly changed based on who is currently typing. If Live Share is used, its API could be used to determine who is actively changing the document.

Rapport Building

Rapport building is a metric we determined out-of-scope for our project very early on so we did not formally define requirements for it. Some relevant data to track for rapport building might include laughter, mimicking, sharing of ideas, asking for help, or having friendly conversations unrelated to the programming task. Both Deepgram and Hume AI have features which could potentially be utilized for rapport building. Deepgram's more robust natural language processing

features require pre-recorded audio instead of live streaming. Topic detection (https://developers.deepgram.com/docs/topic-detection) could be used to detect positive side conversations. Hume AI also has features which may be useful for rapport building, such as detecting laughter and sentiment analysis for audio clips.

Packaging for Installation

The project currently needs to be cloned from GitHub and built manually. VS Code extensions can be bundled and published to be installed on the VS Code marketplace or packaged into .vsix files which can then be installed directly into VS Code. Our efforts to package the extension have not been successful, likely due to the way the React webview is implemented. If packaging proves to be necessary and is not working, a potential solution is to move the React app which displays feedback into a new route on the VM's application. Then, a simpler webview could be implemented in the extension, which simply opens that route to display the webpage.

Security

Currently, the security for the API routes is not robust. You may wish to add some kind of token or certificate authentication system. However, many middleware authentication approaches would require a login, which our sponsor generally discouraged as it is not ideal for research purposes.

Common Troubleshooting Procedures

Turn on Developer Tools

The VSCode Extension debugger that the extension runs in has a built-in console. It can be accessed by navigating to the ‘Help’ tab on the taskbar and clicking ‘Toggle Developer Tools.’ This is useful when debugging extension elements using console.log and other forms of output.

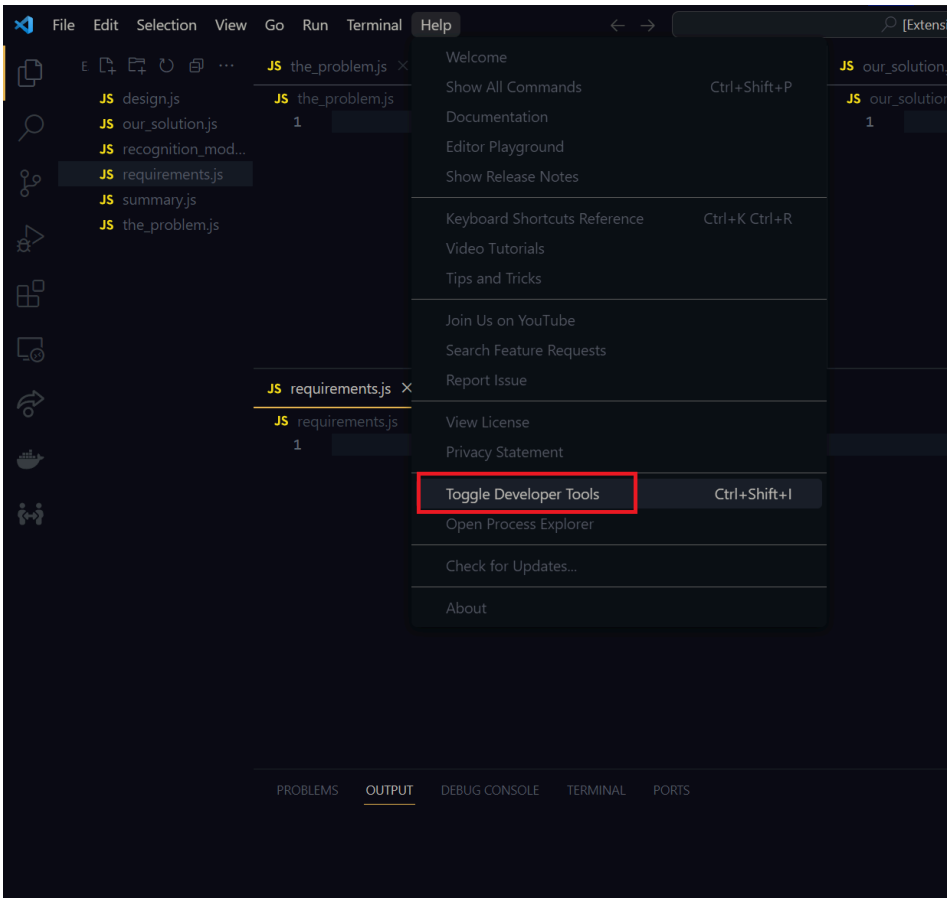

Figure 44: Location of the Option to Toggle Developer Tools

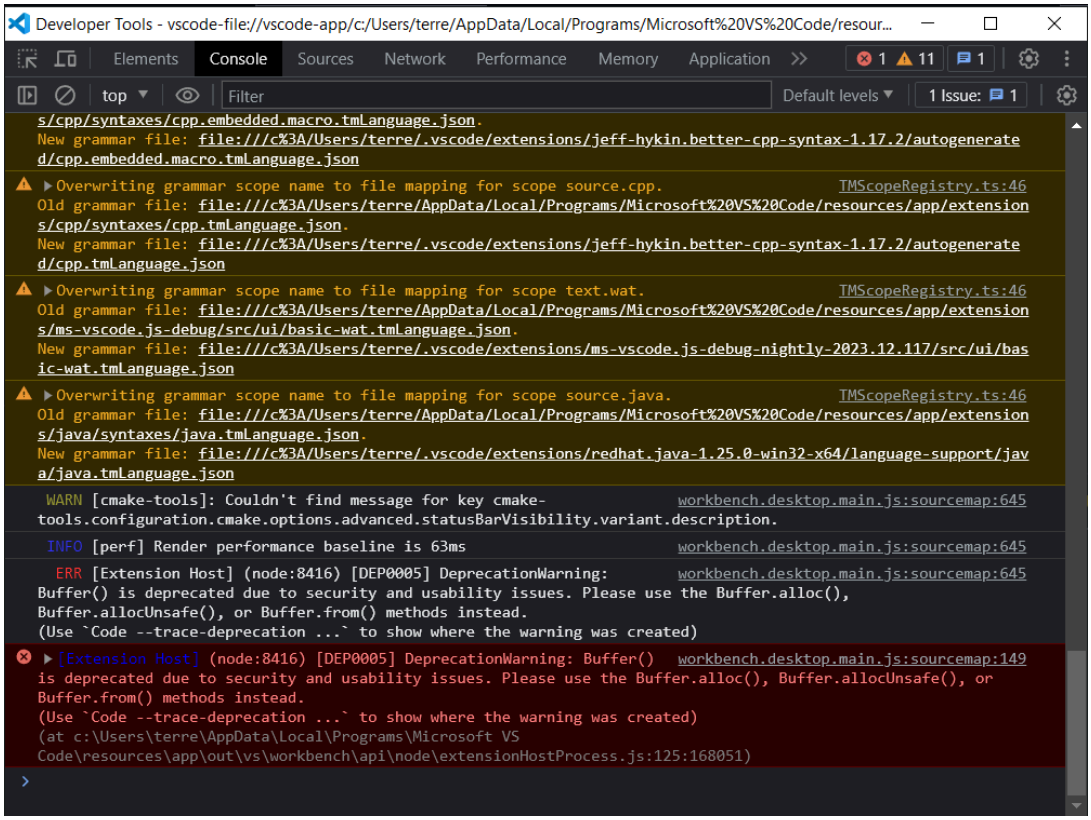

Figure 45: Developer Tools of the Extension Debugger.