Fostering Inclusive Pair Programming with Awareness Tool

Final Project Report

Preliminary Requirements, Design, Implementation & Testing

Kuttal

CSC 492 Team 3

Terrell Dixon
Lindsey Dunkley
Megan Planchock
Declan Smith
Sam Stone

North Carolina State University
Department of Computer Science

December 8th, 2023

Executive Summary

Author(s): Declan Smith
Reviewer(s)/Editor(s): Lindsey Dunkley, Megan Planchock, Sam Stone

The sponsor of this project is Dr. Sandeep Kuttal, an NCSU associate professor who runs the Human-Centric Software Engineering lab. Dr. Kuttal has conducted research showing a difference in outcomes for pair programming with same-gender and mixed-gender pairs due to differences in communication styles. Dr. Kuttal wants a tool to aid in further researching pair programming with same and mixed-gender pairs. The tool should be designed to increase empathy between partners to improve the quality and efficiency of collaboration.

An existing VS Code extension was created in the Summer of 2023 that solves some of these problems. This extension was a combination of a frontend web application and a webview within VS Code. However, this project had a lot of flaws. It did not have a user-friendly interface or any extensive testing. At the beginning of our project, we tried to build the existing extension so we could reuse some of the code, but after hours of work, we were never able to get it to work. The setup instructions were not accurate, and we eventually had to scrap it entirely.

Our proposed solution is to recreate the VS Code extension from scratch which facilitates pair programming using LiveShare and a WebRTC video call while monitoring each participant's facial expressions, code contribution, and vocal interruptions. The extension will provide live feedback to the users during the programming session, as well as a report of the session upon completion of the programming session. The live feedback will consist of lines of code contribution, ratio of contribution as a driver/navigator, and a count of total utterances and vocal interruptions. The end-of-session report will include a collaboration score, the driver/navigator ratio, primary communication style, a final tally of interruptions, leadership style, and a self-efficacy score. We will improve the user interface, and iteratively test our project through unit and acceptance tests as we build the extension.

Initially, we tried to implement our project with everything within the VS Code extension. However, this was a dead end because VS Code extensions cannot access the microphone or camera, which are vital in emotion and utterance detection. In order to proceed, we formulated and executed a new design which utilizes a VS Code extension alongside a separate web application which runs in the browser. This web app accesses the necessary user media devices that were inaccessible from within the extension. With these devices, the web app currently runs a video call between two users and accesses emotion and voice detection models that send speech-to-text utterances and facial expression scores to the database. The extension itself tracks lines of code and contains a webview panel where users can view their live feedback and end-of-session report. In regards to testing, we have 100% of unit tests passing and 83% of acceptance tests passing. The next steps of the project are to implement simultaneous editing by the extension automatically managing the LiveShare session and the handling of role switching with feedback related to it.

Project Description

Author(s): Sam Stone
Reviewer(s)/Editor(s): Lindsey Dunkley

Sponsor Background

The sponsor for the Pair Programming Tool is Dr. Sandeep Kuttal, an associate professor at NC State University in the department of Computer Science. Dr. Kuttal runs the Human-Centric Software Engineering lab, dedicated to studying individuals' behaviors while programming and applying these findings to Human-Computer Interaction. Her research combines the multidisciplinary fields of HCI, Artificial Intelligence, Software Engineering, Education, and Empirical Evaluation. With her research, she designs and develops mixed-initiative programmer-computer systems. Her current research studies how men and women work together in same-gender and mixed-gender pair programming sessions.

Pair programming is a technique used in software engineering where two developers work on the same code together on one computer. This practice consists of both a driver and a navigator. The driver is a developer who is actively writing code and the navigator is the person observing and giving input and reviewing their code in real time. Throughout a pair programming session, the driver and navigator will switch roles to take turns working on the code. Pair programming can promote teamwork, create higher work quality, share best practices, and promote efficiency.

Name: Dr. Sandeep Kuttal
Title: Associate Professor, Department of Computer Science at NCSU
Address: 3264 EB II
Phone: 919-513-7543

Problem Description

The results of Dr. Kuttal's research on same and mixed-gender pair programming sessions has shown that men and women have different communication styles. Based on survey results, lab studies, and interviews, a set of metrics were found that showed the key differences between men and women while pair programming. These metrics include primary communication style (either as a driver or a navigator), leadership style (authoritative or democratic), communication style (verbal or nonverbal), the level of self-efficacy (high or low), rapport building, and number of vocal interruptions (when one person begins speaking while their partner was already talking). Dr. Kuttal wanted a tool that aided in her research to see the different styles of how people work together. Additionally, the tool was created to help increase empathy among partners to help individuals work better together regardless of the gender of their partner.

A VS Code extension was created with this purpose in mind over the Summer of 2023. This implementation monitors the pair programming partners' performance through facial expressions, keyboard strokes, and interruptions throughout the session. During the session, the user can view stats about their pair programming session thus far. At the end of the session, they can view a full report of the session, which provides

feedback on areas they can improve as a partner. The stats are measured based on the metrics from Dr. Kuttal's research. However, this system doesn't cover all of the metrics that Dr. Kuttal's research was monitoring, lacks testing, and doesn't have sufficient usability and functionality based on HCI principles. In addition, the tools implemented to track keyboard strokes and emotions are not sufficient as they provide inaccurate data. Dr. Kuttal would like the tool to measure lines of code written instead of keystrokes as it is a better indicator of the work-balance between partners. Also, the previous VS Code extension's implementation for emotion detection uses the FER 2013 dataset that only has 65-68% accuracy in detecting an individual's emotions. This project lacked any testing and did not have a user-friendly interface.

Proposed Solution & Project Goals/Benefits

To solve the problems of the old implementation, Dr. Kuttal would like the current system to be created from scratch with a focus on creating a tool that has extensive testing in both usability and functionality. The new tool should include most of the previous project's functionality including tracking facial expressions and transcribed vocal lines for interruptions with the additional metric of lines of code written. The new tool should have a robust UI that follows HCI principles and the flow of the functionality should be easy for users of any experience level to use with ease.

The proposed solution is to create a VS Code Extension that can be used for remote pair programming by connecting two users. This will allow for the exchange of the following collected data: facial expressions, speech-to-text dialogue with timestamps, and lines of code written. This data is displayed during the session to each user in order to see how many times they have interrupted their partner, the ratio of time they have spent in each role, how many lines of code they have written as the driver, and the number of utterances they have spoken as the navigator. At the end of the session, each user will receive an individual report created from both users' data that shows scores for their dominant leadership and communication styles along with other metrics including rapport building, self-efficacy, and the ratio of their interruptions to the total interruptions in the session.

The main goal for this extension is to allow two users to receive feedback through data about their performance during the session. This is ultimately intended to foster more inclusive pair programming. This is done in order to address the communication struggles and differences programmers have during a session. This will aid in Dr. Kuttal's research to see how same and different gender pair programming pairs work and communicate together. In addition, it will lead users to be more empathetic toward their partner and to improve their teamwork skills.

The major goals of the project are the following:

- Create a VS Code extension that allows two users to remotely connect to each other and start a pair programming session
- Run recognition models to fulfill Dr. Kuttal's research metrics by collecting data on each user's facial expressions, voice, and lines of code written

- Report real-time feedback on the user's collected metrics compared to their partner's which is displayed within the VS Code extension
- Create a final report that displays the overall performance of the session that the user can view once the session ends
- Create UI and a process flow that abides by HCI principles
- Have a system with end-to-end functionality tested using written acceptance and unit tests to ensure the project is stable for a research setting

Resources Needed

Author(s): Declan Smith, Terrell Dixon, Sam Stone
Reviewer(s)/Editor(s): Lindsey Dunkley, Megan Planchock

Table 1: Resources Needed

Name	**Purpose**	**Status**	**Version**	**Licensing Information**
NCSU VM	Host WebSocket server and signaling server for WebRTC.	Obtained	Ubuntu 22.04 Linux	GPL
Hume AI	Facial and voice recognition models.	Obtained: Using a free trial, but will eventually need access for paid API calls	v0	Proprietary
Deepgram	Voice-to-Text for tracking the utterances, or the spoken sentences and phrases from each user.	Obtained: Using a free trial, but will eventually need access for paid API calls	v1.0	MIT License
React	A Javascript framework for building UI	Obtained	v18.2.0	MIT License
MongoDB	NoSQL database used for storing data	Obtained	v7.0	Server Side Public License

	throughout a session			
VS Code	Lightweight IDE that the extension is being built in	Obtained	v1.84	Open Source

Risks & Risk Mitigation

Author(s): Terrell Dixon, Megan Planchock

Reviewer(s)/Editors(s): Sam Stone

1. Insufficient progress in the allotted time.
    - Risk: Due to unfamiliarity with the technologies necessary to complete the project, especially VSCode and its limitations, it became a race against time to deliver something that fulfills the sponsor's requirements.
    - Mitigation: We developed a flexible mindset towards scope and what would be the best features to implement in our limited time. We made sure to keep the teaching staff aware of our technical difficulties and any misunderstandings we had so that we could receive advice and change plans as necessary. Research before decisions of resources also was a big factor in saving time as we progressed.
2. Slow or desynced transfer of information.
    - Risk: There are multiple components of our project that require internet access to use correctly. Because of this, they can be subject to network latency and lack of synchronization between systems. This can lead to an unsatisfying experience and inaccurate information.
    - Mitigation: Upon successful peer-to-peer connection, our system will send synchronous signals to each user's application to initiate model execution. While guaranteeing exact timing synchronization is difficult, we will optimize our system to maximize performance and minimize the effect latency would have.
3. Biased/Incorrect Results
    - Risk: Machine-learning models for emotion and voice recognition could be trained on data sets that are not diverse which can produce results that are skewed to fit the biases reflected in the data sets, or can be completely wrong.
    - Mitigation: Much research and analysis of various models was necessary to choose a service that was cost efficient, sufficiently accurate, and covers a large and diverse range of people.
4. Sensitive Data Collection:
    - Risk: Collecting voice and facial data poses privacy concerns and may deter users from accessing the extension.

- Mitigation: We prioritize user privacy and will not require users to provide any personally identifiable information. Instead, each user will be assigned a unique ID, and collaboration data will be stored anonymously, using these IDs as references. This approach ensures data security and protects user privacy while still enabling collaboration tracking and analysis.

Development Methodology

Author(s): Lindsey Dunkley
Reviewer(s)/Editor(s): Declan Smith

Our team is following an iterative process for the development of the project. Iteration lengths range from two to three weeks depending on the quantity of features being implemented along with their estimated implementation times. Having two-to-three-week iterations allows us to meet with our sponsor two to three times for the development of major project features as we meet with Dr. Kuttal weekly. This allows us to share both progress and allows for the evaluation of what we are actively working on with time to alter what has been implemented to better suit the needs of our sponsor.

Each iteration has specific features that are both implemented and tested. The reason that an iterative process was chosen for our development methodology is because the project's requirements are individual enough to warrant separating implementation of different features over each iteration. However, because some features eventually overlap in later iterations, being able to organize the order of features being created avoids both potential merge conflicts and relying on dependencies that may be delayed in development, require changing after receiving sponsor or teacher feedback, or that have not yet been implemented at all.

All tasks are tracked in our task planning table on a Google doc with a status, assignee, and due date. We are following acceptance test-driven development where acceptance tests have been pre-written before the implementation process has begun to test that a specific feature is behaving as expected. Unit tests are written after implementation to test for the correct behavior of both function calls and React components.

System Requirements

Author(s): Terrell Dixon, Lindsey Dunkley, Megan Planchock, Declan Smith, Sam Stone
Reviewer(s)/Editor(s): Lindsey Dunkley

Overall View

This project allows two users to create a pair programming session through a VS code extension. When creating the session, each user will be given a unique ID to share with their partner in order to connect the session. After connecting, both users will start as the navigator. Once the session begins, a video call will be initiated between the two users. Throughout the session, data will be collected on both users including: facial expressions, the number of lines of code written and modified but not deleted, and speech-to-text transcripts with beginning and ending timestamps. Additionally, user roles can be switched throughout the session; whoever is actively typing in VS Code is the driver while whoever is not typing is the navigator. If no one is typing in VS Code, they are both navigators. If both are typing in VS Code, then they are both drivers. Upon the completion of a session, the data collected will be presented to both users with scores received individually for primary communication, leadership style, communication style, self-efficacy, rapport building, and interruptions as in when that user began speaking while their partner was already actively talking. Communication style and leadership style are calculated based on the number of utterances, which are defined as segments of speech where a person pauses during or after a sentence.

Functional Requirements

FR1: Create Session

FR1.1. The system shall allow exactly two users to establish a pair programming session.

FR1.2. The system shall begin collecting facial expressions, lines of code written and modified, and speech-to-text transcripts with timestamps at the beginning of a session.

FR2: Detect Roles

FR2.1. The system shall assign both users as the navigator upon starting a session.

FR2.2. The system shall allow at most 2 drivers in one session at a time.

FR2.3. The system shall allow at most 2 navigators in one session at a time.

FR2.4. The system shall reassign a user with the navigator role to the driver role when the user begins to type.

FR2.5. The system shall reassign a user with the driver role to the navigator role when the user stops typing.

FR3: End Session

FR3.1. The system shall allow either user to end a pair programming session.

FR3.2. The system shall end both user's sessions when one of the users has elected to end the session.

FR3.3. The system shall stop collecting users' facial expressions, text-to-speech transcripts with timestamps, and number of lines of code written or modified once the session has ended.

FR4: Video Call

FR4.1. The system shall create a real-time video call with audio between the two users that have started a session at the beginning of the session.

FR4.2. The system shall end the video call between the two users on the session's ending.

FR4.3. The system shall allow the user to hide their video from themselves and their partner.

FR4.4. The system shall allow the user to mute their microphone so their partner cannot hear them.

FR5: Simultaneous Editing

FR5.1. The system shall allow two users to share code files.

FR5.2. The system shall allow two users to simultaneously live edit shared code files.

FR5.3. The system shall allow two users to stop sharing code files when a session ends.

FR6: Line Count

FR6.1. The system shall record the code contribution of each user as a number of lines written.

FR6.2. The system shall allow each user to view the ratio of lines written by themselves to the lines written by their partner during the session.

FR7: Facial Recognition

FR7.1. The system shall identify facial expressions of the user in real time during a session.

FR7.2. The system shall categorize facial expressions into positive, negative, and neutral.

FR7.3. The system shall measure the frequency of positive, negative or neutral expressions over time during a session.

FR8: Voice Detection

FR8.1. The system shall record a text-based transcript of a user's speech.

FR8.2. The system shall count the number of utterances from each user by calculating an utterance as a segment of speech when a user is talking to when they pause in their speech.

FR8.3. The system shall detect interruptions when a user begins speaking while the other user is talking.

FR9: User Feedback

FR9.1. The system shall offer real-time individual feedback after a session has begun.

FR9.2. The system shall display the ratio of lines of code they have written compared to their partner throughout an ongoing session.

FR9.3. The system shall display the number of utterances a user has made while having the navigator role throughout an ongoing session.
FR9.4. The system shall display the ratio of time the user has been the driver compared to the time their partner has been the driver.
FR9.5. The system shall display the number of times they have interrupted their partner throughout an ongoing session.
FR9.6. The system shall generate an individual comprehensive summary of each user's session after a pair programming session has ended.
FR9.7. The system shall display the user's primary communication as either driver or navigator in the end of session report.
FR9.8. The system shall display the user's communication style as either verbal or non-verbal in the end of session report.
FR9.9. The system shall display the user's self efficacy as either high or low at the end of session report.
FR9.10. The system shall display the number of times the user has interrupted their partner while talking at the end of the session report.
FR9.11. The system shall display personalized feedback during and after the session on how they can improve to be a better pair programming partner.
FR9.12. The system shall store the end of session report once the session has ended and the final report has been generated and contains the calculated scores for primary communication, leadership style, communication style, self-efficacy, rapport building, and speech interruptions.

Non-Functional Requirements

NFR1. The latency between the detection by any recognition model and the succeeding feedback provided to users shall be no longer than 300 milliseconds.

Constraints

C1. The system must be implemented as a VS Code extension.

Design

Author(s): Terrell Dixon, Lindsey Dunkley, Megan Planchock, Sam Stone
Reviewer(s)/Editor(s): Declan Smith

Design Tradeoffs

After creating our initial design where our major components were an extension and a server, we ultimately had to perform a design overhaul due to VS code extensions having multiple constraints that impacted our ability to create major functionality, leading to persistent roadblocks in the first iteration of implementation. The major change was that multiple subcomponents including our data collection components and video call were now going to be running from a web application rather than from the extension itself. One of the main reasons for doing this was that we were unable to access the camera and microphone through Visual Studio Code inside our extension's panel which

ultimately meant we could not create a video call [FR 4.1] nor could we track facial expressions [FR 7.1] or create a speech-to-text script [FR 8.1].

One alternative to the approach of using a web application was to use a web extension instead. However, this was ultimately decided against as web extensions are not inside the regular VS Code application which is more difficult for the user that has already configured and customized their own VS Code environment.

High-Level Design

Our system architecture can be divided into four primary components: the Visual Studio Code extension, a web application, a server, and a database. Each user will download and build their own local copy of the extension (see Figure 1). Upon activation, the VS Code extension directs the user to the web application running on the VM.

The web application's frontend will have the responsibility of initiating WebRTC calls and activating the recognition models by opening WebSockets to the Deepgram and Hume AI streaming APIs. Then, at set intervals, information is collected from the video stream through the WebSocket. Hume AI returns the emotion of the expression while Deepgram returns information including a transcript of the audio with confidence levels for each word along with start and stop times for the words. WebRTC will utilize the server's WebSocket to establish a connection between the two users.

The frontend components facilitate communication with the extension and the database. As the frontend gets results for the utterances and emotions, these values are sent into the database through the express server's APIs. Additionally, the app communicates starting and closing sessions through a WebSocket from an express router on the express server using messages such as "start" with information like the user's ID. This allows the extension to access the data sent from the app in the database using the API routes combined with the user's ID including interruptions [FR 8.3] from the speech-to-text sentences overlapping timestamps.

The extension will facilitate communication between the extension's user interface (UI) and the database. During a session, the VSCode extension will directly retrieve results from the recognition models through the web application. However, when it comes to interruption data, as neither user's web page is aware of the other's, information about each user's utterances will be stored in the database. The extension will request this data and use the timestamp attribute to determine if interruptions occurred during the session.

Upon the session's completion, the extension will ensure data persistence by inserting the end session results [FR 9.12] with the user's ID into the database through a POST API endpoint running on the express server.

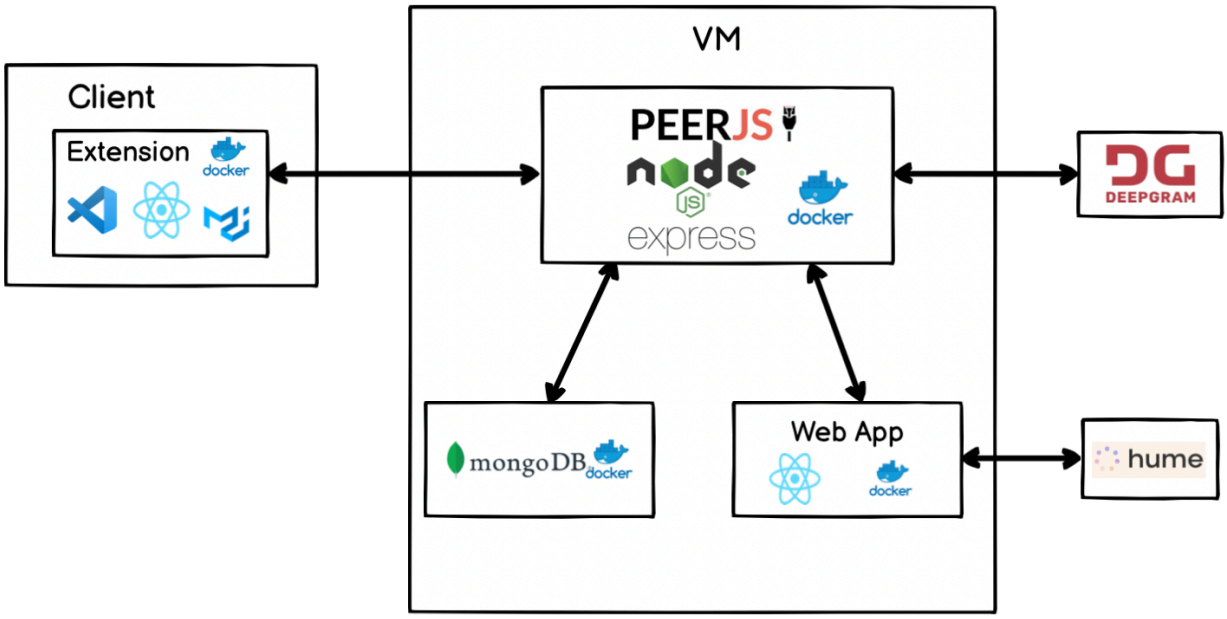


Figure 1: High Level System Architecture

Low-Level Design

Two of the primary components in the high-level design are the extension and the web application, both of which will be constructed using React applications. The connection between components for the web application are shown in Figure 2. The entry point for this application is App.js, which was accessible at https://sd-vm01.csc.ncsu.edu/, however future teams will need to access the application at https://<hostname>/. Upon entry the App.js file renders the Waiting component simply displays a waiting for media permissions message. Once the user grants the request, the app switches the page to the Create Session component.

In the CreateSession component, a unique ID is generated for the user, and the user inputs their partner's ID. This component is responsible for initiating the start of a session and manages communication over a WebSocket to ensure the validity of the partner's ID.

Once the IDs are paired, the App switches from Create Session to the Video Call. The user's ID and their partner's ID will be passed as props to this component. The VideoCall component is rendered and responsible for initiating the WebRTC call using the WebRTC data channel. Time synchronization between the users will be managed from this WebRTC data channel. The VideoCall component returns the layout with both users' video feeds displayed, and supports allowing the users to mute their microphone and hide their camera. Additionally, the VideoCall component executes the voice and emotion recognition models through the Voice and Emotion components which receive the audio and video streams respectively from the VideoCall component. The VideoCall, Voice, and Emotions components will remain active and mounted in the web browser until the session ends.

The Emotions component takes screenshots of the videos at half-second intervals, and sends the image as a base64-encoded string through the WebSocket the Emotions component establishes with the Hume streaming API. The API endpoint provides an emotion, such as "happy," "confused," or "frustrated." The Emotions component within the app interprets these emotions and assigns a numerical value of -1, 0, or 1, representing negative, neutral, or positive sentiments, respectively. These mappings are explicitly defined in the constants.js file, located in the components folder of the app directory, with specific values assigned for each of the 48 emotions that Hume can detect. Over five second intervals, these numerical values are averaged. At the end of the interval, the Emotions component updates the User's expression score with the average and the average resets.

The Voice component establishes a WebSocket connection to the transcription.js file located in the server/recognitionModels directory, and sends the user id to the transcription file. The transcription file is hosted externally because Deepgram is not compatible with React. Subsequently, the Voice component sends the audio stream through the WebSocket as soon as the data becomes available. Within the transcription.js file, a new Deepgram client is instantiated, and sends the audio stream from the Voice component to the Deepgram client. The Deepgram client processes the input and returns a transcript, encapsulating all the words uttered in the spoken phrases. Upon receiving the transcript, the transcription file takes the output and stores it in the MongoDB database for further reference.

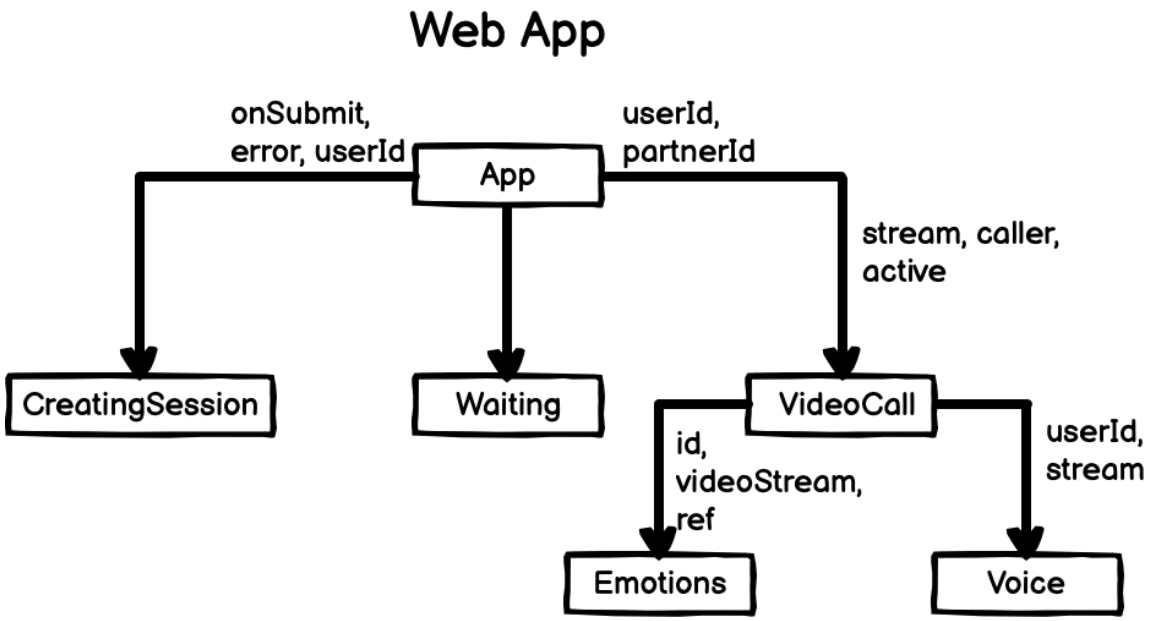


Figure 2: Web Browser Component Diagram

In Figure 3, the components needed for the extension are included with the relationship between them. The components for the extension encompass all the HTML elements

rendered in the root div of the HTML side panel within the Visual Studio Code environment. The communication between the user's extension and the app occurs using a WebSocket on express routers running on an express server. The extension has its own WebSocket route that it communicates information such as the extensionId, when a session ends, and when the user clears their report. This allows for the extension panel to be open and closed without losing the status of the session. In addition, information is communicated from the extension to the database using axios with API calls.

Similar to the web application, the entry point for the extension is a file called App.js. While a valid video call is in progress between the users, the extension will navigate to the Session component. When a user ends the call, the Session component, which has a switchPage function passed to it as a prop, will render the EndSession component. The roles and responsibilities of the Session and EndSession components are similar. Session retrieves information about interruptions and lines of code through the APIs running on the express server, displaying real-time feedback using components like accordions and pi charts. EndSession retrieves information on interruptions, utterances, lines of code, and emotion data through API calls to the same express server. EndSession then conducts analysis on these values to calculate the user roles, self-efficacy, and more, displaying feedback using generic UI components like accordions and pi charts. When users conclude a session, the EndSession component is rendered and summarizes the calculated scores for the session (specifics on what scores available in [FR 9]) while updating the database with the calculated values from the summary.

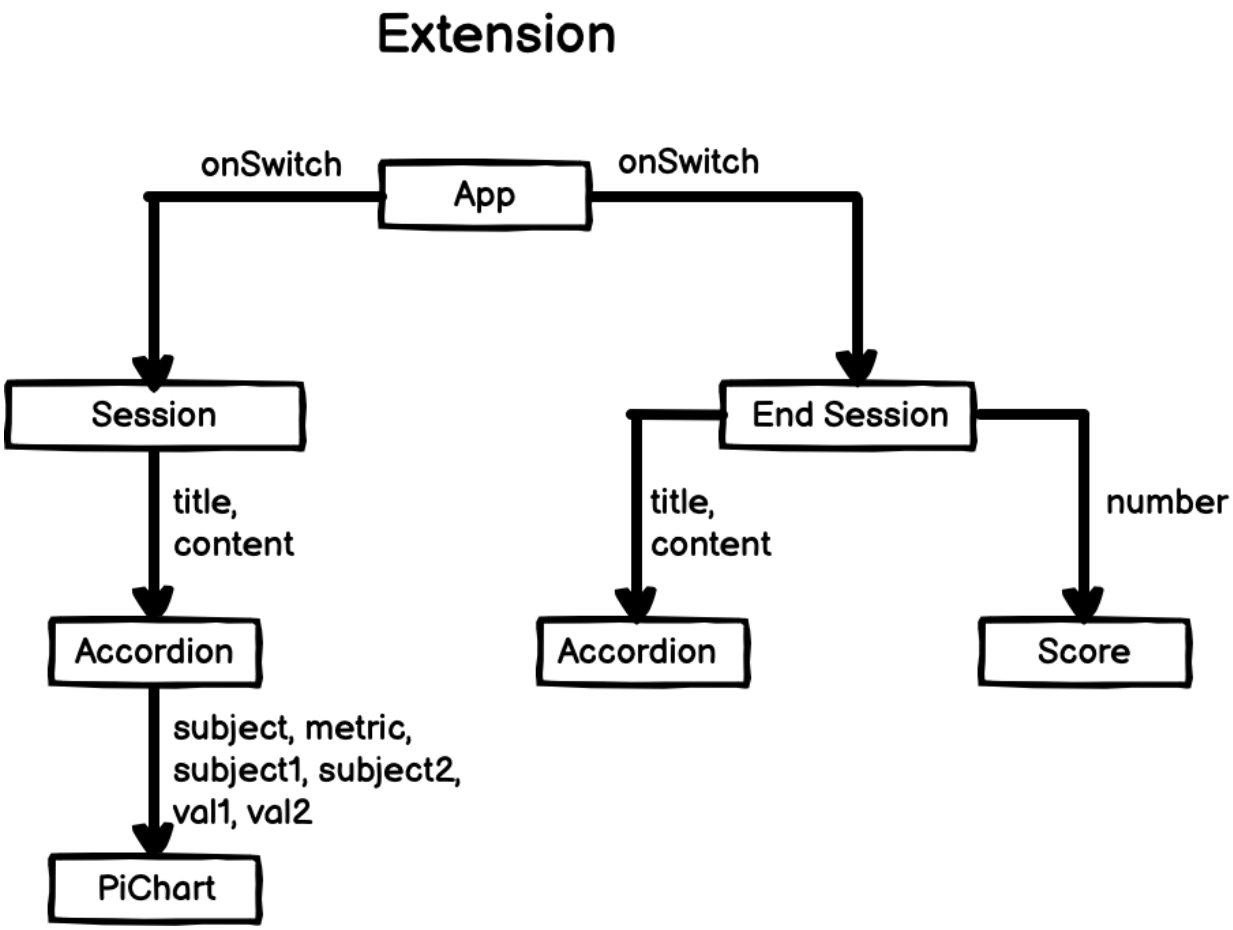


Figure 3: Extension Component Diagram

The database we are using is MongoDB, which is a NoSQL database that allows for flexible schemas. Our database has 4 collections, a sessions collection, utterances collection, users collection, and report collection. Each of the collections have a user_id field so the documents can be queried to find the user the data is associated with. They also all have an _id field, which is automatically generated by MongoDB, where each document has its own unique id.

The sessions collection contains the string id's of both users who have been paired together in a session. The users collection contains all of the information that is collected about a user throughout the session. The num_role_changes field keeps track of the number of times a user has switched between driver and navigator. The expression_scores field is an array of Numbers that keeps track of the user's average emotional valence (intensity of positive/negative emotions) at every 5 second interval throughout the session.The num_interruptions field is a count of how many times the user has interrupted their partner. The num_utterances field is the number of utterances a user has made throughout the session.

The utterances collection contains information about an utterance that the user makes. A user will have many utterance documents throughout the course of the session, whereas in every other collection, the user will just have one instance of it. The start_time and end_time fields denote the timestamp when the utterance started and ended. These fields are used to calculate interruptions between partners. The transcript field is the speech-to-text transcript of what a user said during that utterance.

The reports collection represents the stats shown at the end of a session and contains all of the metrics that were calculated based on a user's performance as a partner. The primary_communication field represents if the user was more of the driver or the navigator. The leadership_style field represents if a user was more authoritative or democratic. The communication_style field represents if a user was more verbal or nonverbal. The self_efficacy_level field is either high or low based on if the user had an increase or decrease in positive emotions.

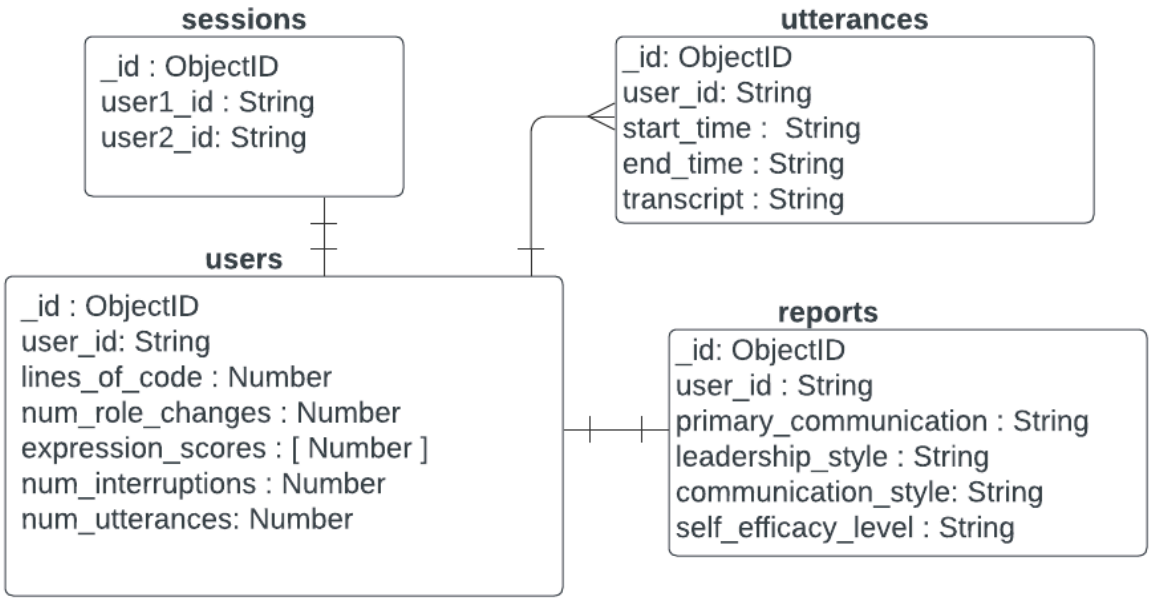


Figure 4: Database Document Structure

Figure 5 shows the key for the sequence diagrams. They are grouped by what part of the application the file is in, either the extension, web application, server, or the database. Figures 6-8 use this key.

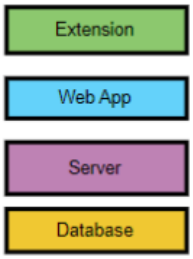


Figure 5: Sequence Diagram Key

Figure 6 shows how the session is started by the CreateSession component in the web app. Once the session is started, this component will render the VideoCall component with each partner's id, the MediaStream (both Video and Voice), the caller and active props. The caller prop is the person who enters the id to start the session. The active prop is true when the session is actively running. Once the session is running, the VideoCall component will insert a session document into the database. This document will contain both of the user's ids. It will then render both the Emotion and Voice components by sending in the user's id and either the VideoStream for Emotion, or the MediaStream for Voice. Both the Emotion and Voice components are in the web app, so the stream can be sent directly. The transcript.js file, which runs the utterance detection, has to run on the server, which is why the Voice component then sends the Voice stream to it (through a WebSocket).

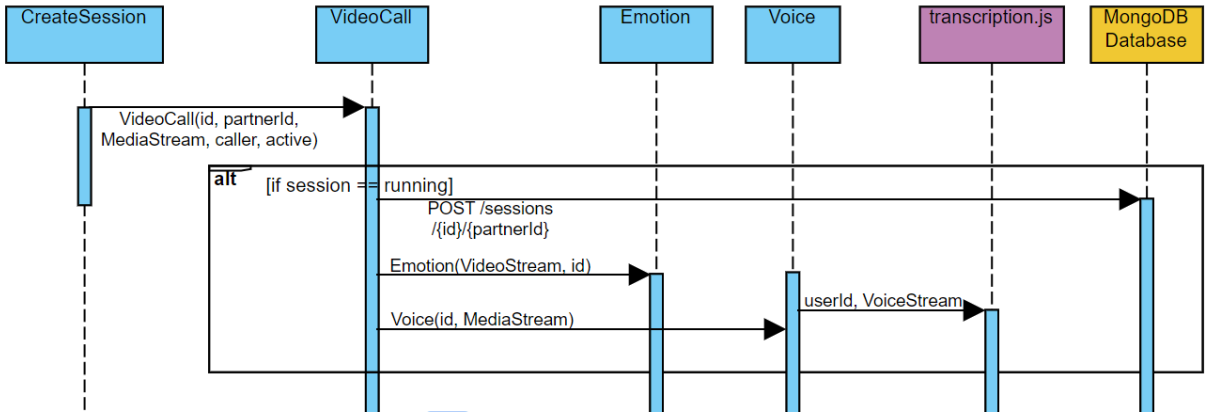


Figure 6: Start Session Sequence Diagram

The next part of the sequence diagram shown in Figure 7 is what happens continuously during a session. The extension.js file calculates the lines of code a user writes, then updates the user's user document with the current lines of code. The transcript.js file inserts the utterances into the database. The Emotion component detects the user's emotions, calculates the average positive/negative score, and adds that score to the

user's user document. The Session component displays the lines of code and number of interruptions. Both of these stats are retrieved from the database with lines of code by accessing the GET /utterances/interruptions/{userId}/{partnerId} and by retrieving the user's user collection which contains a lines of code field.

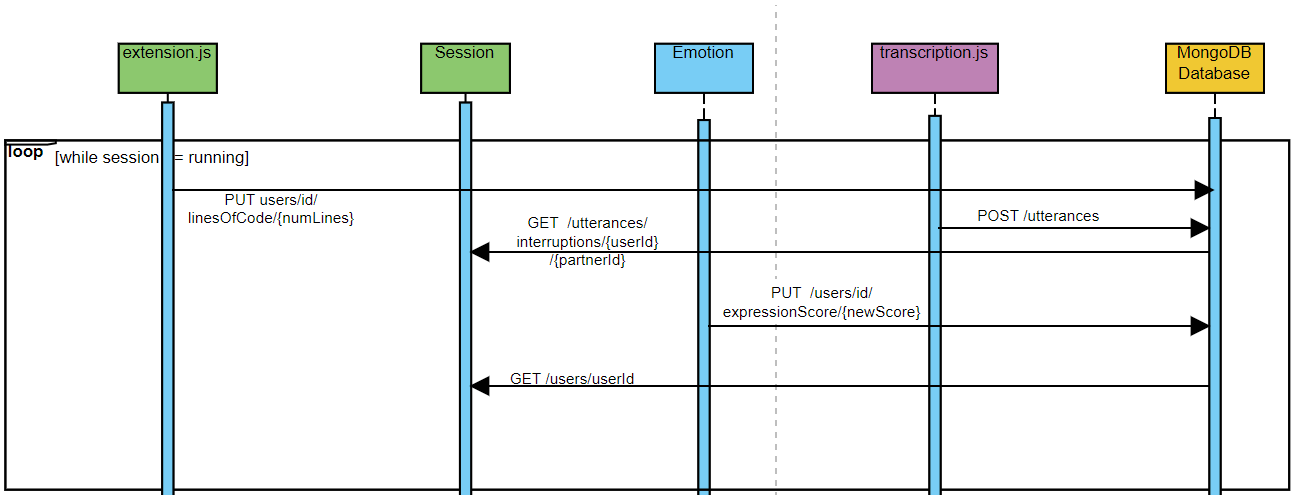


Figure 7: During Session Sequence Diagram

Figure 8 shows what happens when a user clicks the "End Session" button inside of the extension, the Session component renders the End Session component as true. The End Session component displays the final report to the user by calculating information that was gathered during the session. It retrieves the number of utterances and user documents by calling API endpoints. Once all of the metrics are calculated using utterances and the fields stored in the user document, it inserts the report into the database. Lastly, all of the user's utterances are deleted as they are not needed anymore, contain sensitive information, and take up too much space in the database.

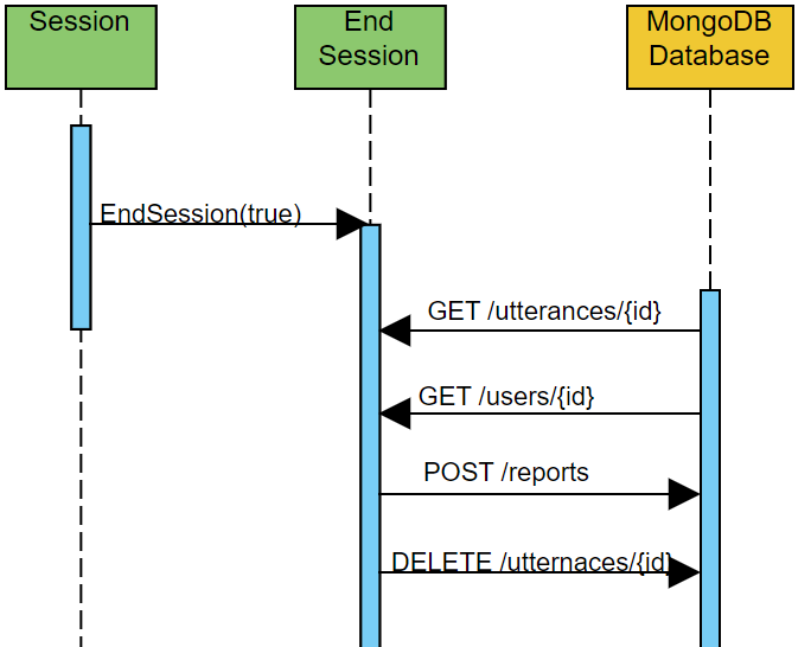


Figure 8: End Session Sequence Diagram

REST API Calls

Note: All of the REST API endpoints precede with "http://hostname/server/api," where the hostname is the hostname where the VM is running.

Insert a Session and Users

- Endpoint: /sessions/{user1_id}/{user2_id}
- HTTP Verb: POST
- Request Payload
  - Route Params: user1_id, user2_id
- Response Payload
  - HTTP Status Code: 200
  - Conditions for Response: Success
  - Body Content:

```
{
  user1_id: user1_id,
  user2_id: user2_id,
  _id: ObjectId,
  __v: Number
}
```

  - HTTP Status Code: 409
  - Conditions for Response: Users already have a session registered
  - Body Content:

```
{
  “Session already exists with these users”
}
```

- HTTP Status Code: 500
- Conditions for Response: Failure
- Body Content:

```
{
  “Failed to insert Session”
}
```

Insert an Utterance

- Endpoint: /utterances
- HTTP Verb: POST
- Request Payload
  - Body Content:

```
{
  user_id: String,
  start_time: Number,
  end_time: Number,
  transcript: String
}
```

- Response Payload
  - HTTP Status Code: 200
  - Conditions for Response: Success
  - Body Content:

```
{
  user_id: user_id,
  start_time: start_time,
  end_time: end_time,
  transcript: transcript,
  _id: ObjectId,
  __v: Number
}
```

  - HTTP Status Code: 409
  - Conditions for Response: User has not been registered in a session
  - Body Content:

```
{
```

```
  "A session does not exist with
these users"
}
```

- HTTP Status Code: 500
- Conditions for Response: Failure
- Body Content:

```
{
  "Failed to insert Utterance"
}
```

Insert a Report

- Endpoint: /reports
- HTTP Verb: POST
- Request Payload
  - Body Content:

```
{
  user_id: String,
  primary_communication: String,
  leadership_style: String,
  communication_style: String,
  self_efficacy_level: String
}
```

- Response Payload
  - HTTP Status Code: 200
  - Conditions for Response: Success
  - Body Content:

```
{
  user_id: user_id,
  primary_communication: primary_communication,
  leadership_style: leadership_style,
  communication_style: communication_style,
  self_efficacy_level: self_efficacy_level,
  _id: ObjectId,
  __v: Number
}
```

  - HTTP Status Code: 409

- Conditions for Response: User has not been registered in a session
- Body Content:

```
{
  “A session does not exist with these users”
}
```

- HTTP Status Code: 500
- Conditions for Response: Failure
- Body Content:

```
{
  “Failed to insert Report”
}
```

Retrieve a User

- Endpoint: /users/{user_id}
- HTTP Verb: GET
- Request Payload
  - Route Params: user_id
- Response Payload
  - HTTP Status Code: 200
  - Conditions for Response: Success
  - Body Content:

```
{
  user_id: String,
  Lines_of_code: Number,
 num_role_changes: Number,
 expression_scores: [Number],
 num_interruptions: Number,
 num_utterances: Number
}
```

  - HTTP Status Code: 409
  - Conditions for Response: User does not exist in the database
  - Body Content:

```
{
  “{userId} does not exist”
}
```

- HTTP Status Code: 500
- Conditions for Response: Failure
- Body Content:

```
{
  “An error has occurred
retrieving {userId}”
}
```

Retrieve a User’s Report

- Endpoint: /reports/{user_id}
- HTTP Verb: GET
- Request Payload
  - Route Params: user_id
- Response Payload
  - HTTP Status Code: 200
  - Conditions for Response:
  - Body Content:

```
{
  user_id: user_id,
  primary_communication:
primary_communication,
  leadership_style: leadership_style,
  communication_style: communication_style,
  self_efficacy_level: self_efficacy_level,
  _id: ObjectId,
  __v: Number
}
```

  - HTTP Status Code: 409
  - Conditions for Response: User does not exist in the database
  - Body Content:

```
{
  “A report for {userId} does not
exist”
}
```

  - HTTP Status Code: 500

    - Conditions for Response: Failure
    - Body Content:

```
{
  “Failed to retrieve Report”
}
```

Delete a user’s utterances

- Endpoint: /utterances/{user_id}
- HTTP Verb: DELETE
- Request Payload
    - Route Params: user_id
- Response Payload
    - HTTP Status Code: 200
    - Conditions for Response: Success
    - Body Content:

```
{
  “Successfully deleted {user_id}’s utterances”
}
```

    - HTTP Status Code: 409
    - Conditions for Response: User does not exist in the database
    - Body Content:

```
{
  “{user_id} does not exist”
}
```

    - HTTP Status Code: 500
    - Conditions for Response: Failure
    - Body Content:

```
{
  “Failed to delete utterances”
}
```

Updates a user’s lines of code

- Endpoint: '/users/{user_id}/linesOfCode/{line_count}
- HTTP Verb: PUT
- Request Payload
    - Route Params: user_id, lines_of_code
- Response Payload

    - HTTP Status Code: 200
    - Conditions for Response: Success
    - Body Content:

```
{
  “Successfully updated {user_id}’s lines of code”
}
```

    - HTTP Status Code: 409
    - Conditions for Response: User does not exist in the database
    - Body Content:

```
{
  “{userId} does not exist”
}
```

    - HTTP Status Code: 500
    - Conditions for Response: Failure
    - Body Content:

```
{
  “Failed to update{userId}’s lines of code”
}
```

Updates a user’s expression score

- Endpoint: /users/{user_id}/expressionScore/{new_score}
- HTTP Verb: PUT
- Request Payload
    - Route Params: user_id, new_score
- Response Payload
    - HTTP Status Code: 200
    - Conditions for Response: Success
    - Body Content:

```
{
  “Successfully updated {user_id}’s expression score”
}
```

    - HTTP Status Code: 409
    - Conditions for Response: User does not exist in the database

    - Body Content:

```
{
  “{user_id} does not exist”
}
```

    - HTTP Status Code: 500
    - Conditions for Response: Failure
    - Body Content:

```
{
  “Failed to update {user_id}’s
expression score”
}
```

Calculates the number of times user1 has interrupted user 2

- Endpoint: /utterances/interruptions/{user_id}/{partner_id}
- HTTP Verb: GET
- Request Payload
    - Route Params: user_id, partner_id
- Response Payload
    - HTTP Status Code: 200
    - Conditions for Response:
    - Body Content:

```
{
  numberOfInterruptions
}
```

    - HTTP Status Code: 409
    - Conditions for Response: User does not exist in the database
    - Body Content:

```
{
  “A session does not exist between
these users”
}
```

    - HTTP Status Code: 500
    - Conditions for Response: Failure
    - Body Content:

```
{
  “Failed to retrieve interruptions
for {user_id}”
}
```

Websocket Protocols

Note: All of the Non REST API endpoints that connect with the remote express pairing server use a different host name for their endpoint: which is based on the device that is hosting the server. All of the connections start with “wss://sd-vm01.csc.ncsu.edu.”

Server and devices confirm connection

- Endpoint: /server/ws
- Action Verb: hello
- Request Payload
    - Route params:
- Response Payload
    - Response action: hello
    - Conditions for Response: the connection is established
    - Body content:

Registers Extension ID with the Express Pairing Server

- Endpoint: /server/extension/ws
- Action Verb: “extensionID”
- Request Payload
    - Route params: extensionId

Registers App ID with the Express Pairing Server

- Endpoint: /server/ws
- Action Verb: “id”
- Request Payload
    - Route params: id, eid
- Response Payload
    - Response Action: error
    - Conditions for Response: The app id is already registered in the server
    - Body content:

```
{
   “Id already exists”
}
```

    - Response Action: registered
    - Conditions for Response: The id is valid and has not been registered

- Body content:

```
{
   id
}
```

Pairing with another user

- Endpoint: /server/ws
- Action verb: pair
- Request payload:
  - Route params: id, partnerId
- Response Payload:
  - Response action: error
  - Conditions for Response: One or both of the users mentioned in the params are already paired
  - Body Content:

```
{
   “One or both partners is already paired.”
}
```

  - Response action: error
  - Conditions for Response: One or both of the users has already been paired.
  - Body Content:

```
{
   “One or both partners has not registered yet.”
}
```

  - Response action: start
  - Conditions for Response: Both given user ids are valid and have not been paired yet.
  - Body Content:

```
{
   partnerId
}
```

Starting the session after pairing

- Endpoint: /server/ws
- Action verb: pair
- Request payload:
    - Route params: id, partnerId
- Response Payload:
    - Response action: error
    - Conditions for Response: One or both of the users mentioned in the params are already paired
    - Body Content:

```
{
   “One or both partners is already paired.”
}
```

    - Response action: start
    - Conditions for Response: Both user ids are valid, and are not already paired.
    - Body Content:

```
{
   partnerId
}
```

Sending lines of code from the extension to the server

- Endpoint: /server/extension/ws
- Action verb: loc
- Request payload:
    - Route params: id, count

Closing (extension)

- Endpoint: /server/extension/ws
- Action verb: close
- Request payload:
    - Route params: eid, id
- Response Payload:
    - Response action: close
    - Conditions for Response:
    - Body Content:
    -

```
{
   Eid, id
}
```

Closing (app)

- Endpoint: /server/extension/ws
- Action verb:
- Request payload:
    - Route params:
- Response Payload:
    - Response action:
    - Conditions for Response:
    - Body Content:

Keepalive (extension)

- Endpoint: /server/extension/ws
- Action verb: keepalive
- Request payload:
    - Route params: eid
- Response Payload:
    - Response action: keepalive
    - Conditions for Response: The connection is still established
    - Body Content:

Keepalive (app)

- Endpoint: /server/ws
- Action verb: keepalive
- Request payload:
    - Route params: id
- Response Payload:
    - Response action: keepalive
    - Conditions for Response: The connection is still established
    - Body Content:

GUI Design

Figure 9 shows the flow of opening our web application from the extension. When the “Create New Session” button is clicked inside the panel of our extension, the web app is opened in the browser on a localhost where the component for creating a new session is rendered. This prevents the web page from simply opening upon the activation of the extension, but gives the user control over when they want to open the webpage to actively start a new pair programming session.

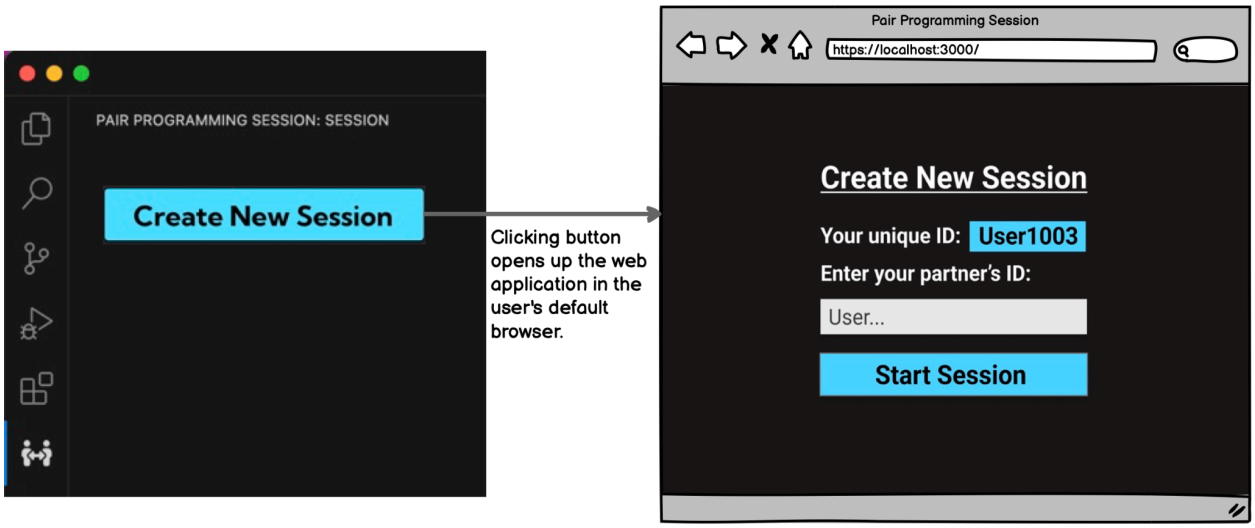


Figure 9: Open Create Session Page in Browser

Figure 10 shows the two possible flows a user may take when interacting with the “Start Session” page. The precondition to Figure 10 is opening the web app in the user’s browser as seen in Figure 9. If the user enters an invalid partner ID and clicks “Start Session,” an error message will be returned stating “ERROR: Please check your partner’s ID and try again” along with a thick red outline around the input to allow the user to have their attention drawn to the problem area.

Otherwise, in Figure 10, when the user enters a valid ID and submits it or when their partner has submitted their valid ID, the user will begin to be redirected into a video call. First, the web page will have a pop-up that requests access to the microphone and video. If the user clicks “Allow,” the call container will pop out of the web browser with the bottom video being the user’s and the top video being their partner’s. The user will be allowed to move this container around. The web application window specifies to not close the tab as the call would end.

The user clicking “Deny” will completely halt and stop the start of a session. If there is no access to the camera and microphone, the data cannot be collected, the video call cannot occur, and the session simply will not start. This is considered “unexpected behavior” or not a part of the expected user flow. However, it will keep the user on the “Start Session” page where they can restart the process for connecting if necessary.

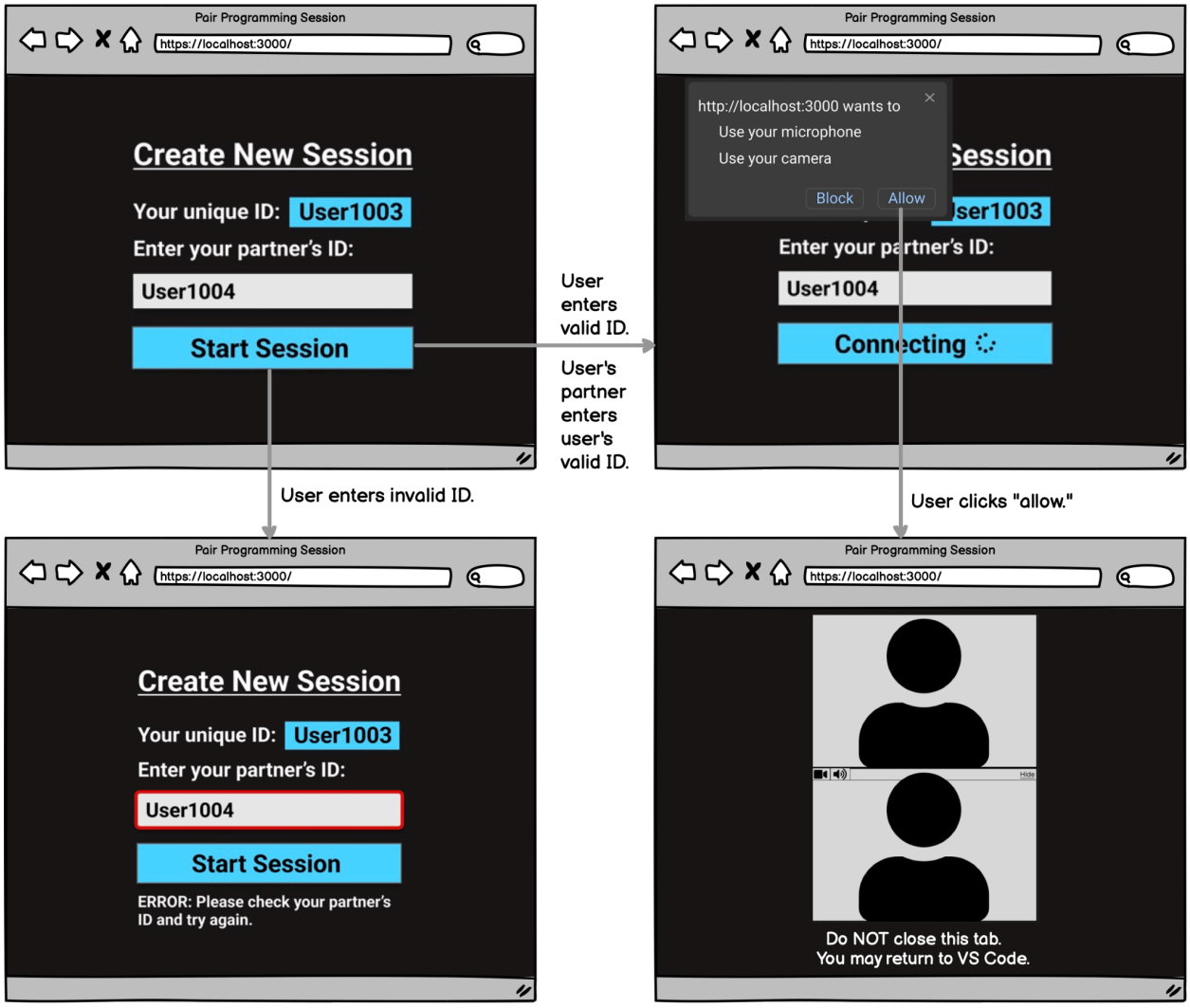


Figure 10: Pairing A Session

Figure 11 is a GUI mockup that exists to show the possible layout of the popout video call over top of VS Code application with our extension running. We did not implement a pop out window for the call, but Firefox does have a built in popout feature for video elements that can be used. On the left of Figure 11, the panel will display all in-session feedback with the current collaboration metrics and will allow either user to end the session. Role switches and the role itself are no longer explicitly defined in the UI as requested by Dr. Kuttal as previously designed in Figure 24 and 25 in the Appendix.

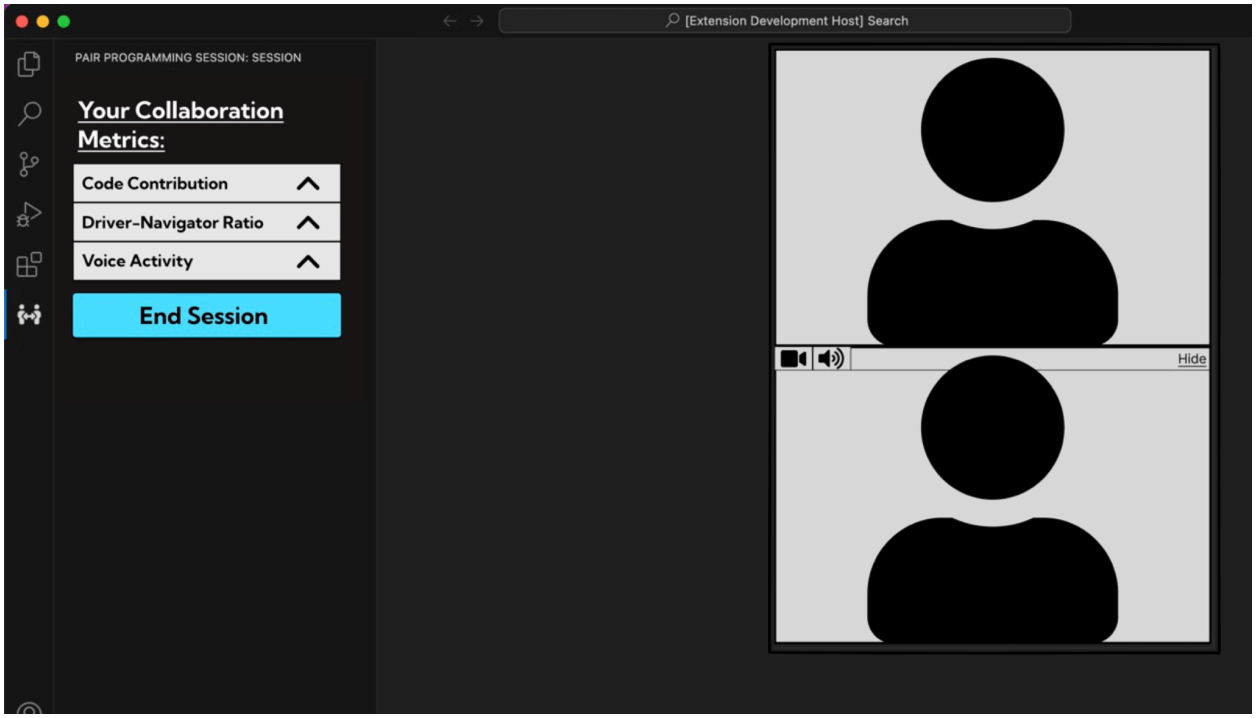


Figure 11: Video Call in Extension

Figure 12 shows the way that in-session feedback will be shown to the user. The main reason for using accordions is because Dr. Kuttal requested that the information and collaboration tips not be something the user is forced to see during the session. As such, opening each accordion with the collaboration data is an optional choice each user has throughout the session. The reason for using the pi charts was to have a more user-friendly way of displaying data, specifically when the data involves comparing two related values. The user will still be able to see the specific number of lines of code that they have written by hovering over the blue section of the pi chart under “Code Contribution” in Figure 12. This way the data is presented in a way that both shows a comparison of the overall contribution, but also allows the user to optionally view more details if desired.

One of the criticisms for the initial GUI designs (Figure 24 and 25 in the Appendix) was that the designs did not look like a VS Code extension. Figure 12 was created as if the UI is displayed within a panel in VS Code which is more like an extension than the previous browser approach of Figure 24 and 25. The choice to use a panel also abided by Dr. Kuttal’s request to not force the user to view their stats as the user is able to close the panel when they no longer want to view any session data, but are able to open the panel throughout the session whenever they please.

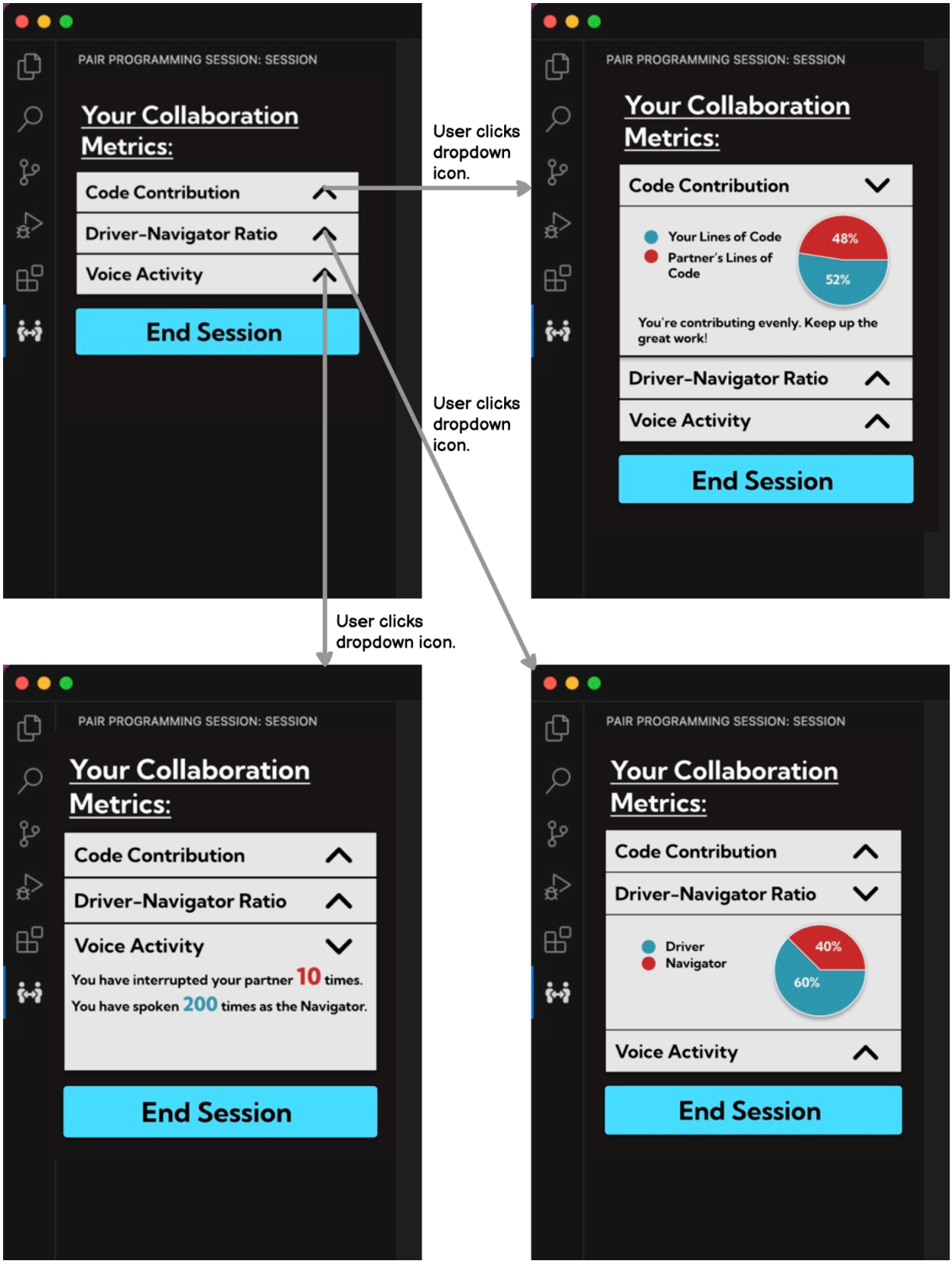


Figure 12: Panel Feedback

Figure 13 is to show the user flow when one user elects to end a session. If the user clicks to end the session using the “End Session” button or their partner chooses to end the session through their own extension, the user’s extension panel will stop rendering the in-session collaboration metrics and begin rendering a final report that displays the collaboration score along with more specifics of their contributions including communication style, role ratio, interruptions, leadership style, and the results of their self-efficacy. It also shows that when a session is ended, the video call is simultaneously ended as well.

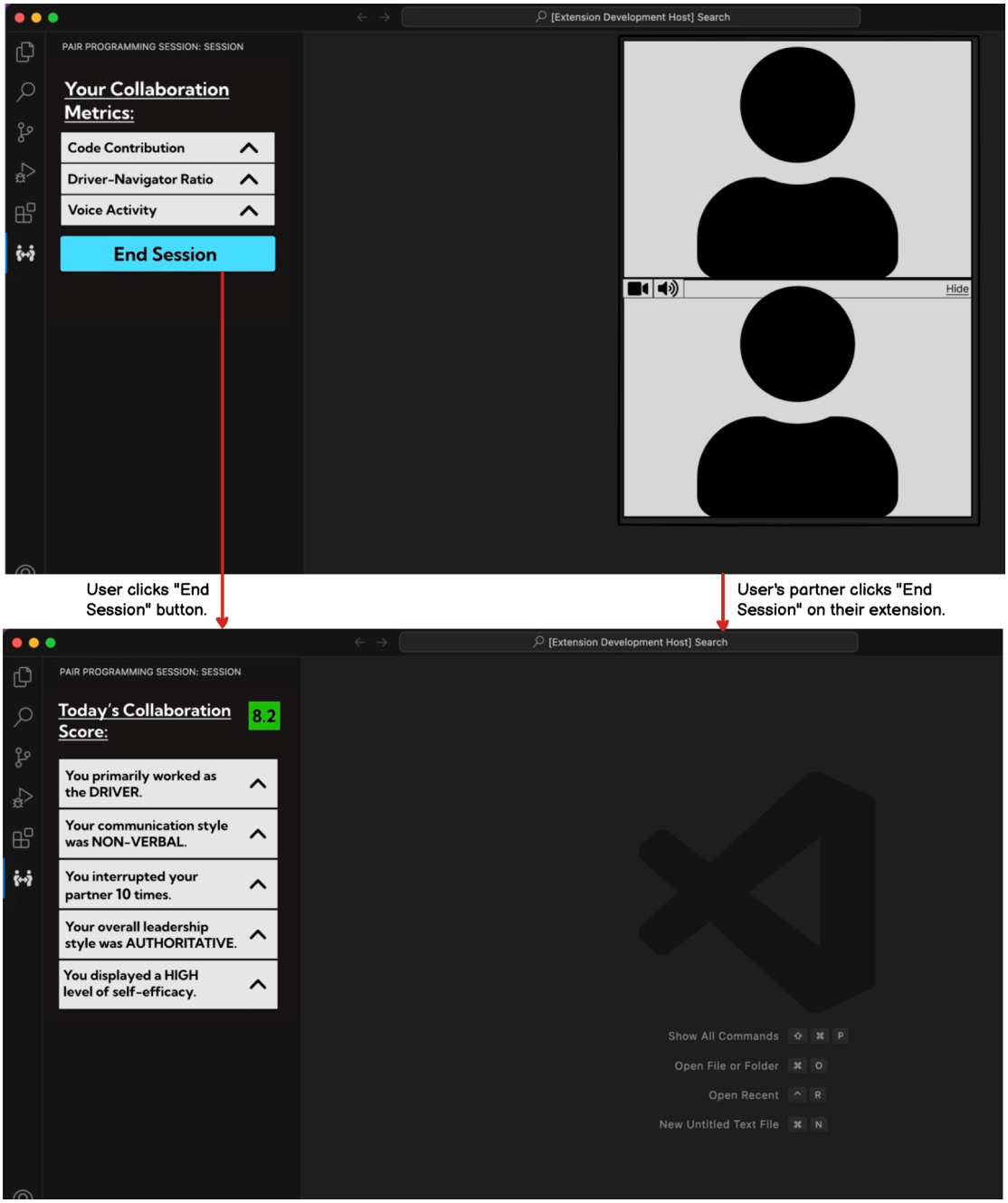


Figure 13: End Session

Figure 14 shows how a user is able to view further specifics of their end-of-session report. The use of accordions is to prevent an excessive display of information at one time that may be difficult for the user to read or follow. Instead, the user can click on the dropdown icon for each accordion when or if they want to view more information about that specific metric.

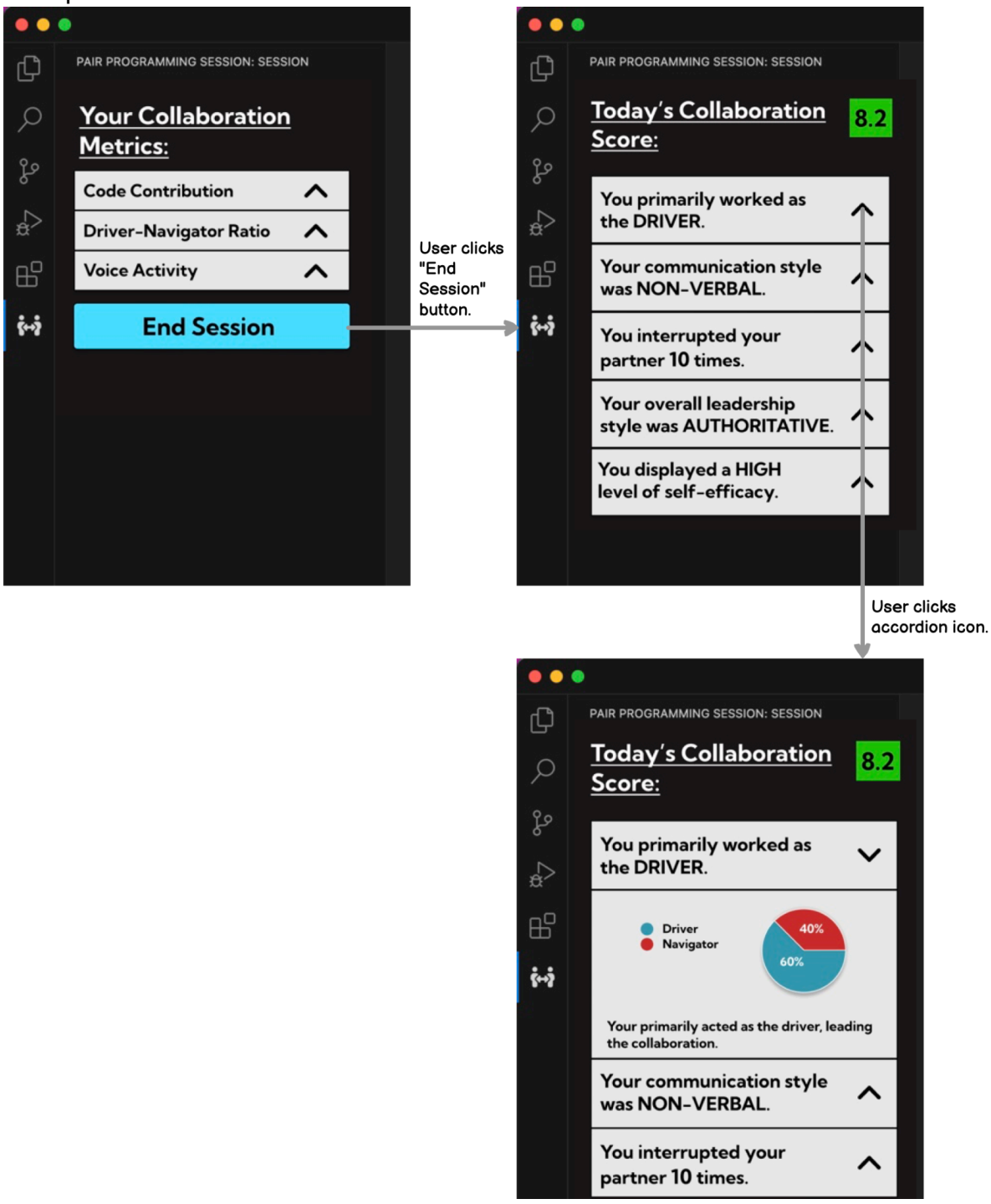


Figure 14: End Session

Implementation

Author(s): Sam Stone, Terrell Dixon, Lindsey Dunkley
Reviewer(s)/Editor(s): Declan Smith

Iteration Definition & Current Status

Iteration 0

Iteration 0 was created for setting up our project including configuring technologies like React with VS Code. FR 1.1 was completed because we created the ability to create a session by pairing two users over a WebSocket server. It started on September 18, 2023, but did not finish until October 6, 2023, largely due to changes in our design.

Iteration 1

Iteration 1 started on October 6th, 2023 and was completed on October 25th, 2023. In this Iteration, create session [FR 1] and video call [FR 4] were the requirements completed.

Iteration 2

Iteration 2 started on October 25th, 2023, and was officially completed on November 20th, 2023. In this iteration, facial recognition [FR 7] and voice detection with speech-to-text transcripts [FR 8] was completed.

Iteration 3

Iteration 3 was the last iteration completed. Iteration 3 officially started on November 20th, 2023, and was completed on December 5th, 2023. During this iteration, the following requirements were completed:

- End Session [FR 3]
- Line Count [FR 6]
- User Feedback
    - [FR 9.1 - 9.2]
    - [FR 9.5 - 9.10]

Iteration 4

Iteration 4 was never started. However, this iteration would have focused on the completion of the following requirements:

- Role switching [FR 2]
- Simultaneous editing [FR 5]
- User Feedback
    - [FR 9.3]
    - [FR 9.4]
    - [FR 9.12]

Security Considerations

The purpose of our system is for research, which means that all information will be stored anonymously with no identifying information, and all information generated from the pair programming session is considered the property of Dr. Kuttal. For our purposes, since the system will be used with no proprietary or identifying data and with limited participants, very few security considerations will be implemented. However, if the extension were to be published and added to the public VSCode Extension Marketplace, multiple security measures would need to be implemented.

Confidentiality:
The information that will be stored using this system will be the ids of the users for each session, and then information for each session based on the user. This information includes lines of code, the number of interruptions, the number of facial expressions and their scores, and the end-of-session report. Since the information will not be saved with any distinguishing information that could be used to identify between specific users, it is not as important to secure.

Integrity:
The system is based on non-sensitive information for research purposes, so we will not implement additional protections to ensure the integrity of the information. Websockets ensure the integrity of traffic between components. Currently, assuming an actor has access to the database where data is likely to be stored unencrypted, and access to the format of the messages being sent to modify the database then that actor can use the ids they find to change the data stored within.

Availability:
Since the system will primarily be used for small-scale research, at least within the scope of our project, maintaining availability is not a significant concern. The extension will be hosted locally, but a separate machine will be necessary to host a server to pair users, the database, and the app running the recognition models. Currently, this functionality is hosted on a remote virtual machine which is accessible on the NC State network. In deployment, our sponsor may wish to make the server more publicly accessible to allow a broader selection of participants. However, it is perfectly acceptable for the server to be started and stopped as needed by those conducting the research study. As such it is not essential to implement functionality to ensure the system is available at all times or scalable to more than a handful of simultaneous connections.

Identification:
All traffic between components of the system, between clients, and to/from the server will contain the id of the current or partnered user. This will help to identify where the information came from and to whom the information is referring to. Websockets, the main method of transferring information, also has functionality that allows the connection information for each websocket to be viewed if necessary.

Authentication:
Since information is stored in the database anonymously, it is not necessary to implement login credentials or functionality to authenticate users. The capabilities of each user are the same for every other user of the extension and the ids to be used in the session are generated at the start of every session. Eventually, the extensions may be adapted to allow users to compare feedback from older sessions to the feedback from more recent sessions to show how they may have improved over time. If this were to be implemented, more robust authentication and identification protocols would need to be considered. Video and audio streams are not being stored anywhere and are just being used for real-time processing. The only information being stored related to the media streams are the analytics such as facial expression scores, and utterance transcripts that are derived from the recognition models.

Accountability:
The main method of transferring information between components of the full system is using websockets. An area that may be vulnerable to accountability issues is when managing the database. Assuming an actor has access to the database, which will likely be stored unencrypted, and access to the format of the messages being sent to modify the database then that actor can use the id's they find to change the data stored within. If they are able to do so, there are no systems in place to check who is the one making changes. In the future, it would likely be necessary to keep logs of changes and require some sort of authentication before a user is able to make changes to the database.

Privacy:
The system will preserve privacy by storing no personal or identifiable information in the database. All video and audio streams will be processed locally in realtime and never recorded. Furthermore, participants in the research that utilizes this tool will be fully informed of all data being gathered, and will only participate in the research if they consent to those terms.

Project Folder Structure

At the root level, our project has 2 folders- an extension folder and a vm folder. The extension folder contains all of the files that are running within the extension itself. The vm folder con `tains all of the files that are running on the vm- the web app and the Express server.

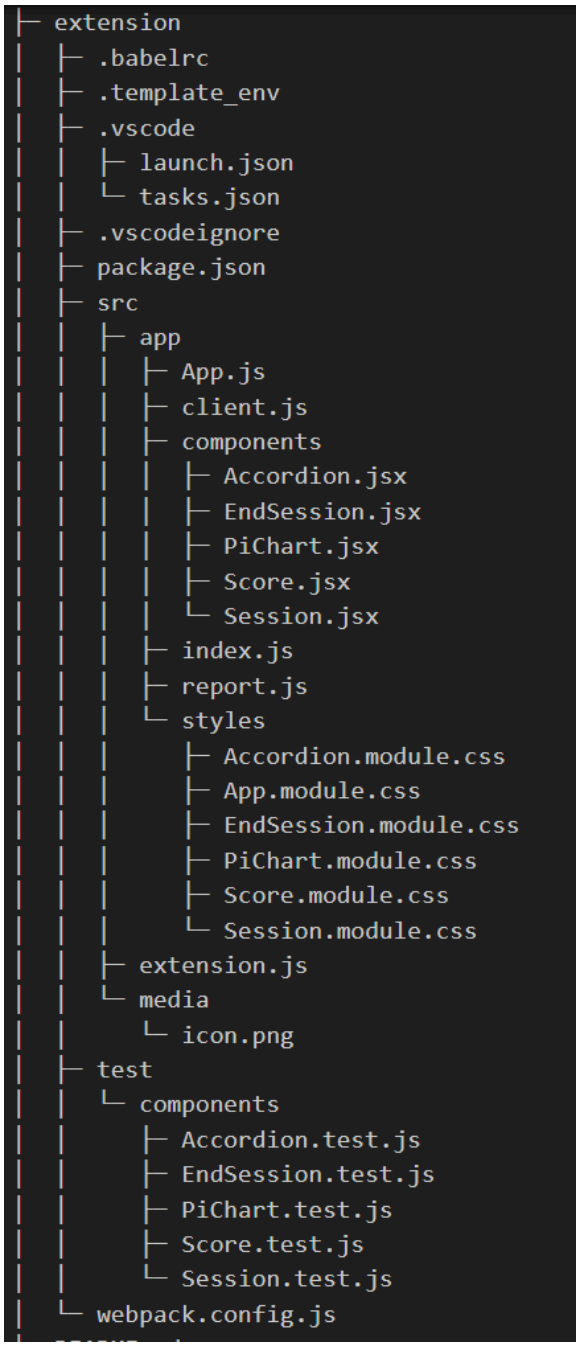


Figure 15: Extension Directory

In the extension folder, as shown in Figure 15, it contains a .vscode folder, a src folder, and a test folder. The .vscode folder contains instructions on how to launch the VS Code extension. The src folder contains all of the directories and files with the extension's components. The test folder contains all of the tests for the extension's components. Within the /src/app folder, it contains a components folder for all of the components within the extension that display the live feedback. The styles folder contains all of the css styling for the components.

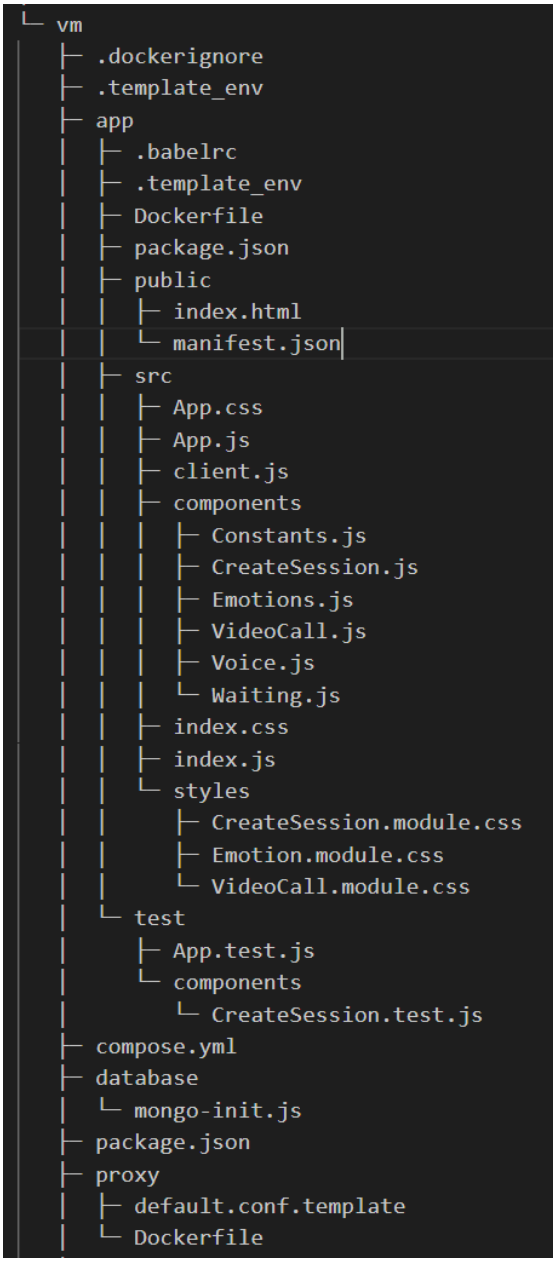


Figure 16: Web App Directory and Root Server Directory

Figure 16 shows the directory for the web application, which is in the /vm/server director. The app folder contains the Dockerfile for the web application. In /src it contains the components directory, which contains the components of the web application to start a session, video call, splitting the media streams, and calculating the emotion scores. The styles folder contains the css for the components. The test folder contains the test for the CreateSession component.

In the root of the server directory, it contains the database directory, which contains the configuration for creating the root user in our MongoDB database. The proxy directory contains the configuration file and Dockerfile for the proxy to move traffic from http to https.

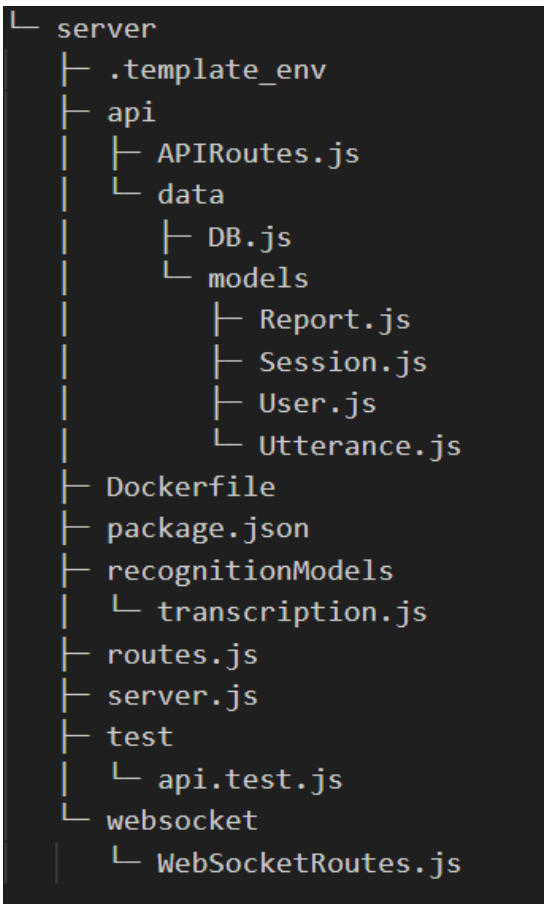


Figure 17: Server Directory

Figure 17 shows the directory where all the server files are which is in the /vm/server directory. It contains the api folder which has all of the API endpoints for inserting, deleting, retrieving, and updating documents in the database. Within the api directory, it contains a data folder with files to connect to the database and a models folder with all of the database collection schemas.

In the root of the server directory, there is a recognitionModels folder for the AI models that run on the server. The only file is for the utterance detection.

Project configurations/settings

There are 4 .env files in the project, 3 on the vm, and 1 in the extension. There are template .env files in the github repo which will need to be filled in for a fresh installation.

The vm/.env contains the information Docker needs to initialize the Database.

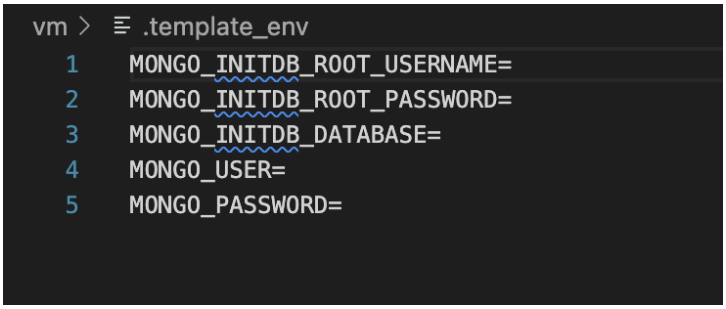


Figure 18: VM .env file for database

The vm/app/.env file contains the information for Hume Ai and Peer.js signaling.

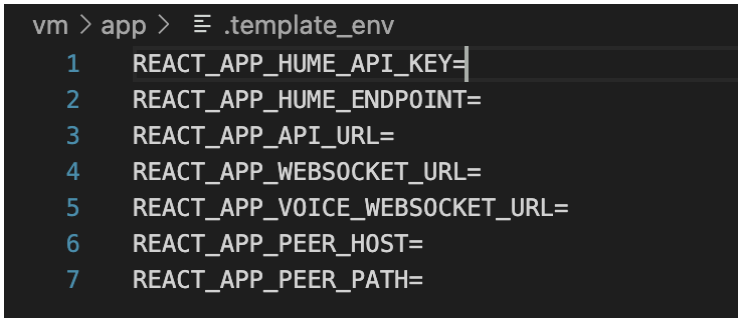


Figure 19: VM Web App .env file

The vm/server/.env file contains the login information to authenticate to the database, the urls for the API routes and database, and the Deepgram API key.

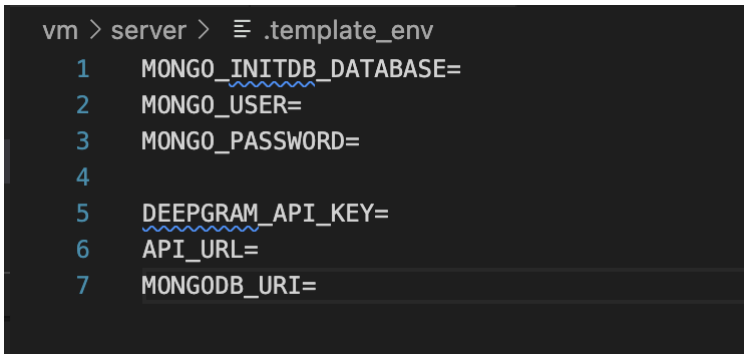


Figure 20: VM Server .env file

Testing

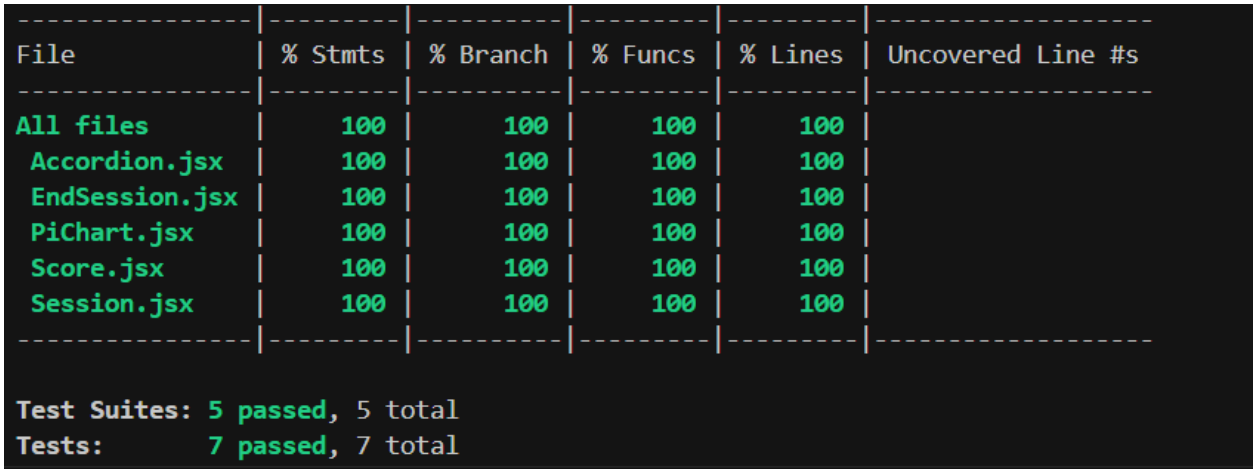
Author(s): Terrell Dixon, Sam Stone
Reviewer(s)/Editor(s): Declan Smith

Overall View

We will predominantly test our project with unit tests written in Jest, and black box system tests.Emotion and utterance detection is a large part of the required functionality, but the user input for these tests involve a person's face and voice, which is hard for unit and acceptance tests. We will also rely on the assumption that these recognition models have already been thoroughly tested themselves. We also were unable to do unit testing on the Video Call component as it is difficult to mock a video stream.

Unit Testing

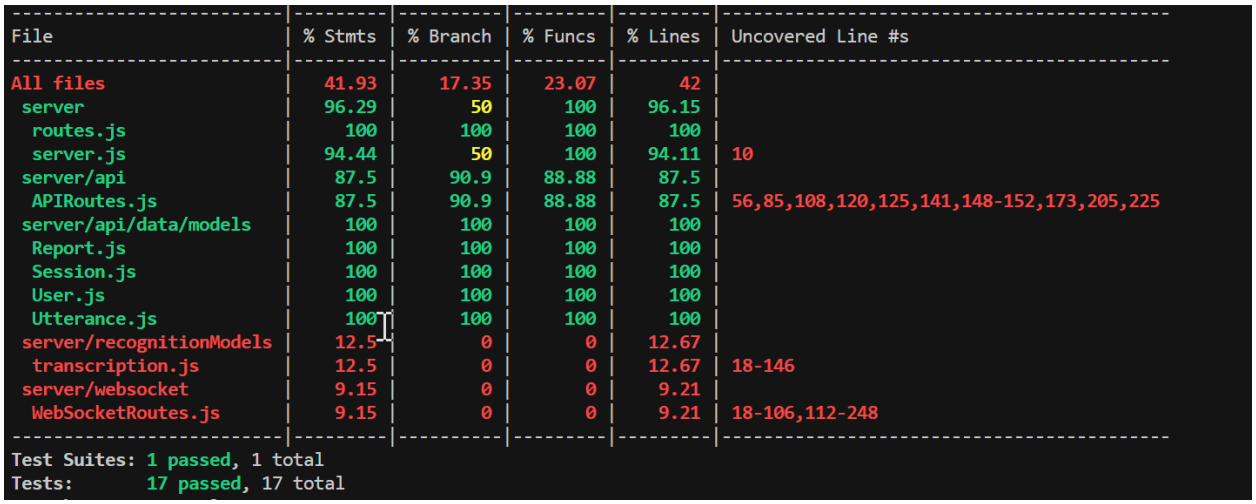
For unit testing we aimed to reach at least 70% coverage across all Javascript files. We were able to test all of the API endpoints with 87.5% statement and line coverage. We were also able to test all of the components in both the extension and the web application with 100% statement and line coverage. We were not able to test any of the App.js files or the websocket routes.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
Accordion.jsx	100	100	100	100	
EndSession.jsx	100	100	100	100	
PiChart.jsx	100	100	100	100	
Score.jsx	100	100	100	100	
Session.jsx	100	100	100	100	

Test Suites: 5 passed, 5 total
Tests: 7 passed, 7 total

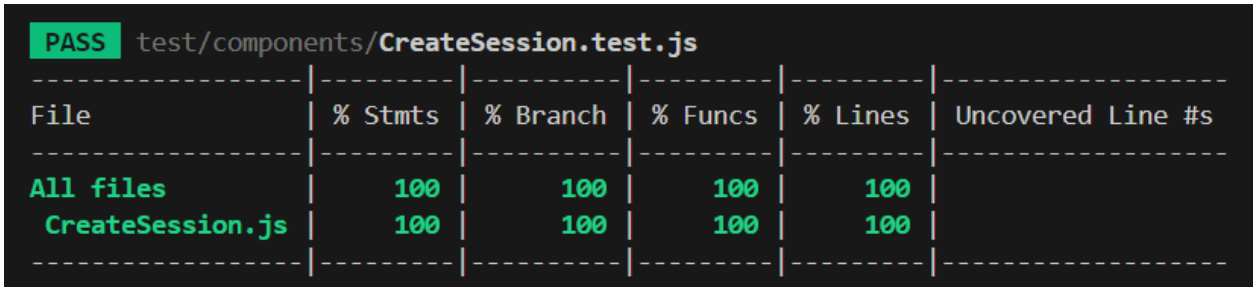
Figure 21: Extension Unit Testing



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	41.93	17.35	23.07	42	
server	96.29	50	100	96.15	
routes.js	100	100	100	100	
server.js	94.44	50	100	94.11	10
server/api	87.5	90.9	88.88	87.5	
APIRoutes.js	87.5	90.9	88.88	87.5	56,85,108,120,125,141,148-152,173,205,225
server/api/data/models	100	100	100	100	
Report.js	100	100	100	100	
Session.js	100	100	100	100	
User.js	100	100	100	100	
Utterance.js	100	100	100	100	
server/recognitionModels	12.5	0	0	12.67	
transcription.js	12.5	0	0	12.67	18-146
server/websocket	9.15	0	0	9.21	
WebSocketRoutes.js	9.15	0	0	9.21	18-106,112-248

Test Suites: 1 passed, 1 total
Tests: 17 passed, 17 total

Figure 22: Endpoint Unit Testing



PASS test/components/**CreateSession.test.js**

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
CreateSession.js	100	100	100	100	

Figure 23: Web App Testing

Acceptance Testing

Table 3: Start and Join Session

Test ID	Description	Expected Results	Actual Results
openExtensionPanel	Precondition: ● Extension is installed and is running. 1. User clicks on the pair programming tool icon.	Extension panel on the side toolbar of VS Code opens, displaying a “Create New Session” button.	Extension panel opens, displaying a “Create New Session” button.
clickCreateNewSessionButton	Precondition: ● Extension is installed and is running. ● The Pair Programming panel is open. 1. User clicks the “Create New Session” button.	The webpage displays “Waiting for permissions..”.	The webpage displays “Waiting for permissions..”.
allowAccessToMediaDevices	Precondition: ● Extension is installed and is running. ● Partner A and B open up the Create New Session webpage. 1. The user allows the webpage to	A browser tab is opened in the user’s default web browser. The webpage displays the user’s session ID, a text input labeled “Enter your partner’s ID,” and a “Start Session” button.	A browser tab is opened in the user’s default web browser. The webpage displays the user’s session ID, a text input labeled “Enter your partner’s ID,” and a “Start Session” button.

	access their camera and microphone.		
enterIncorrectPartnerId	Precondition: ● Extension is installed and is running. ● Partner A and B open up the Create New Session webpage. ● Both partners have allowed microphone and camera access. 1. Partner A enters in the session ID “WrongID” 2. Partner A clicks the Join Session button	An error message appears for Partner A that says “ERROR: Please check your partner’s ID and try again”.	An error message appears for Partner A that says “ERROR: Please check your partner’s ID and try again”.
enterCorrectPartnerId	Precondition: ● Extension is installed and is running. ● Partner A and B open up the Create New Session webpage. ● Both partners have allowed microphone and camera access.	The user's browser page is changed to show two video streams; theirs, and their partners.	The user's browser page is changed to show two video streams; theirs, and their partners.

	1. Partner A sees their session ID displayed on the webpage 2. Partner B enters Partner A's session ID 3. Partner B clicks the Start Session button		
yourPartnerEntersInIdThatIsIncorrect	Precondition: ● Extension is installed and is running. ● Partner A and B open up the Join Session webpage. ● Both partners have allowed microphone and camera access. 1. Partner A sees their session ID displayed on the webpage 2. Partner B enters an ID that *is not* Partner A's session ID 3. Partner B clicks the Join Session	Partner A sees no change in their webpage.	Partner A sees no change in their webpage.

	button		
enterPartnerIdOfUserAlreadyInSession	Precondition: ● Extension is installed and is running. ● Partner B and C have started a session together. ● Partner A has opened up the Join Session webpage. ● Both partners have allowed microphone and camera access. 1. Partner A enters in Partner B's session ID 2. Partner A clicks the Join Session Button	An error message appears for Partner A that says "ERROR: Please check your partner's ID and try again".	An error message appears for Partner A that says "ERROR: Please check your partner's ID and try again".
userTriesEnteringYourIDWhenYoureAlreadyInASession	Precondition: ● Extension is installed and is running. ● Partner A and C have started a session together. ● Partner B has opened up the Join Session webpage.	Partner A and Partner C see no change in their webpage or VS Code extension. Partner B gets an error message that says "ERROR: Please check your partner's ID and try again".	Partner A and Partner C see no change in their webpage or VS Code extension. Partner B gets an error message that says "ERROR: Please check your partner's ID and try again".

	1. Partner B enters in Partner A's session ID 2. Partner B clicks the Join Session Button		

Table 4: Lines of Code and Role Switching

Test ID	Description	Expected Results	Actual Results
testEnteringALineOfCode	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A types <print("Hello World");> in the file and presses enter. 2. Partner A opens the extension.	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution accordion, there is a pie chart that says 100% Your Lines of Code.	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution accordion, there is a pie chart that says 100% Your Lines of Code.
viewingYourPartnersCodeContribution	Precondition: ● Extension is installed and is running. ● Partner A and B have	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution

	started a session together and have opened up a file in VS Code. 1. Partner B types <print("Hello World");> in the file and presses enter. 2. Partner A opens the extension.	accordion, there is a pie chart that says 100% Your Partner's Lines of Code.	accordion, there is a pie chart that says 100% Your Partner's Lines of Code.
viewingEqualCode Contribution	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A types <print("Hello World");> in the file and presses enter. 2. Partner B types <print("Hello Universe");> in the file and presses enter. 3. Partner A	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution accordion, there is a pie chart that says 50% Your Partner's Lines of Code and 50% Your Lines of Code.	Partner A sees Your Collaboration Metrics in the extension panel. Under the Code Contribution accordion, there is a pie chart that says 50% Your Partner's Lines of Code and 50% Your Lines of Code.

	opens up the VS Code extension panel		
viewingCodeContributionWithoutTypingAnything	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A opens up the VS Code extension panel	Under the Code Contribution accordion it displays, “No data to display!”	Under the Code Contribution accordion it displays, “No data to display!”
viewingCodeContributionEndSessionReport	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A clicks the “End Session” Button in the VS Code extension panel	Partner A sees the Today’s Collaboration Score page that pops up from the extension panel. The final results say “You primarily worked as the Navigator” and the pie chart in this drop down shows 0% driver, 0% Navigator.	No driver/navigator panel is shown

Table 5: Video Call

muteVideoCall	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together. ● Each partner has the webpage open and on the page there are two video streams, their own as well as the partners. 1. Partner A clicks the mute button at the top left of their video stream.	Partner A's mute button now has a line through it and Partner B should no longer be able to hear Partner A.	Partner A's mute button now has a line through it and Partner B should no longer be able to hear Partner A.
hideVideoCallCamera	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together. ● Each partner has the webpage open and a video all connected to the other partner.	Partner A's camera button should now have a line through the button. Partner B should no longer be able to see video from Partner A's camera.	Partner A's camera button should now have a line through the button. Partner B should no longer be able to see video from Partner A's camera.

	1. Partner A clicks the camera button at the top left of their video stream.		

Table 6: Voice

Test ID	**Description**	**Expected Results**	**Actual Results**
interruptPartner	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner B starts talking and says “Hello how are you?” 2. Once Partner B says “Hello”, Partner A says “Hi!”. 3. Partner A opens up the VS Code extension panel	In the panel under Voice Activity it says “You have interrupted your partner 1 time.”	In the panel under Voice Activity it says “You have interrupted your partner 1 time.”
testLeadershipStyle Democratic	Precondition: ● Extension is installed and is running.	Partner A sees the Today’s Collaboration Score on the extension	Partner A sees the Today’s Collaboration Score on the extension

	● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A says “Nice to meet you!” 2. Partner B says “Nice to meet you too!” 3. Partner A clicks the “End Session” button in the VS Code extension panel	panel. The results say “Your overall leadership style was Democratic”.	panel. The results say “Your overall leadership style was Authoritative”.
testLeadershipStyle Authoritative	Precondition: ● Extension is installed and is running. ● Partner A and B have started a session together and have opened up a file in VS Code. 1. Partner A says “Nice to meet you!” 2. Partner B clicks the “End session button” in the	Partner A sees the Today’s Collaboration Score on the extension panel. The results say “Your overall leadership style was Authoritative”.	Partner A sees the Today’s Collaboration Score on the extension panel. The results say “Your overall leadership style was Democratic”.

	VS Code extension panel		

Acceptance Testing Status
We are currently passing 15 out of 18 acceptance tests. The tests we currently are not passing is due to role switching not being implemented and a difference in calculating the leadership style (currently using lines of code instead of number of utterances).

Other Testing
The only testing being conducted are the unit tests and acceptance tests.

Suggestions for Future Teams
Author(s): Declan
Reviewer(s)/Editor(s): Sam, Lindsey

For simultaneous editing [FR 5], VS Code Live Share extension has the potential to handle most of the requirements. However, we were unable to get it to work. As such, we suggest planning additional time to pursue other options. Specifically, pursuing IDEs other than VS Code that provide more options than 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 unable to get it to import in our extension.

Another metric which we discussed with Dr. Kuttal, but determined to be out-of-scope, is rapport building. We did not formally define the requirements for rapport building, but relevant things to track for it are laughing, mimicking, sharing of ideas, asking for help, or having friendly conversations unrelated to the programming task. Deepgram's topic detection and keyword searching features may be useful for this. There is thorough documentation on Deepgram's website: https://developers.deepgram.com/docs/topic-detection. Hume AI also has features which may be useful for rapport building, such as detecting laughter and sentiment analysis for audio clips.

Appendix

The following GUI Designs (Figure 24 and Figure 25) are old wireframes that we scrapped due to changing requirements in how roles are being switched as well as our sponsor wanting our design to have more emphasis on Human-Computer Interaction principles to minimize gender bias and maximize usability.

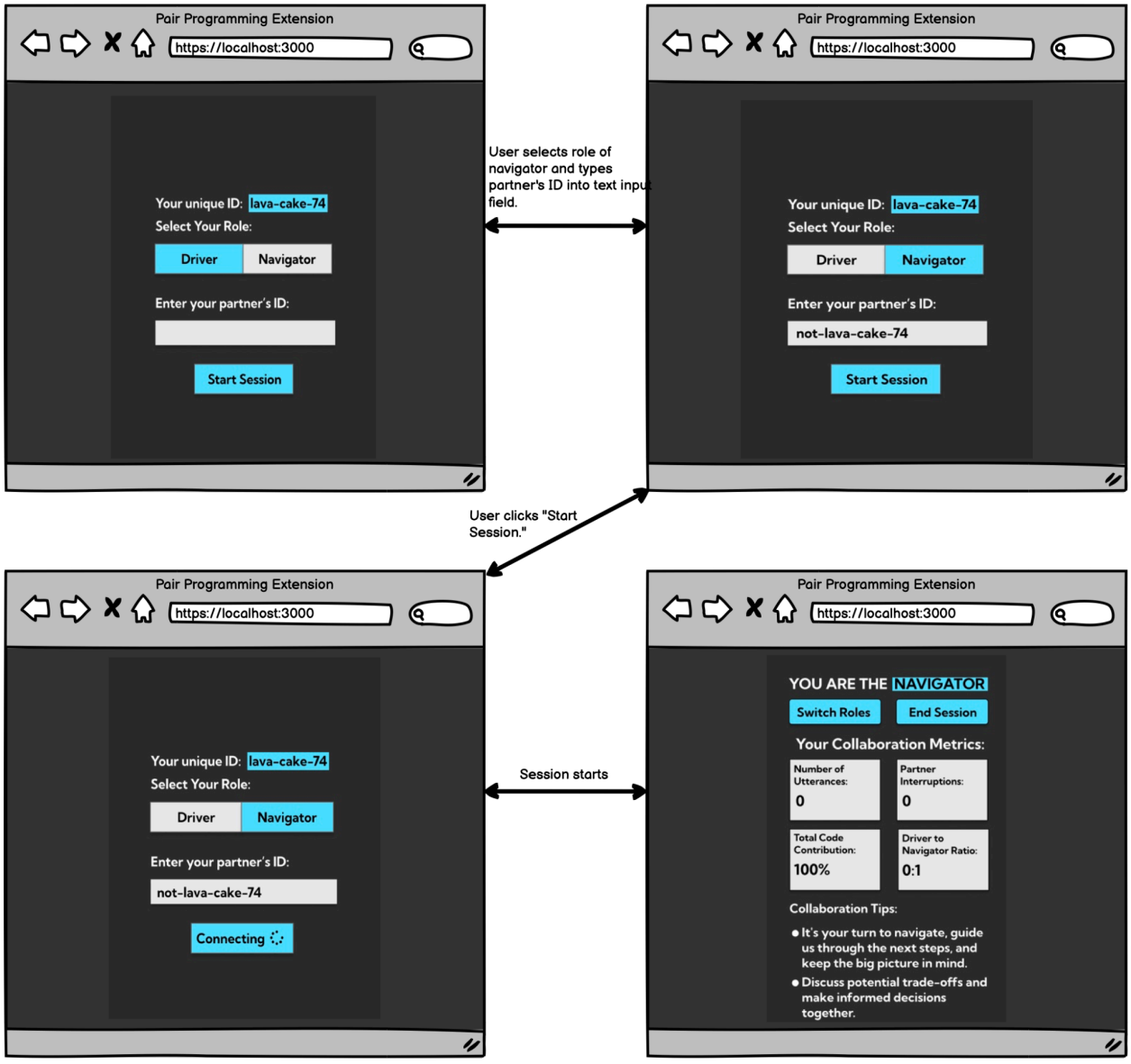


Figure 24: Start Session As Navigator

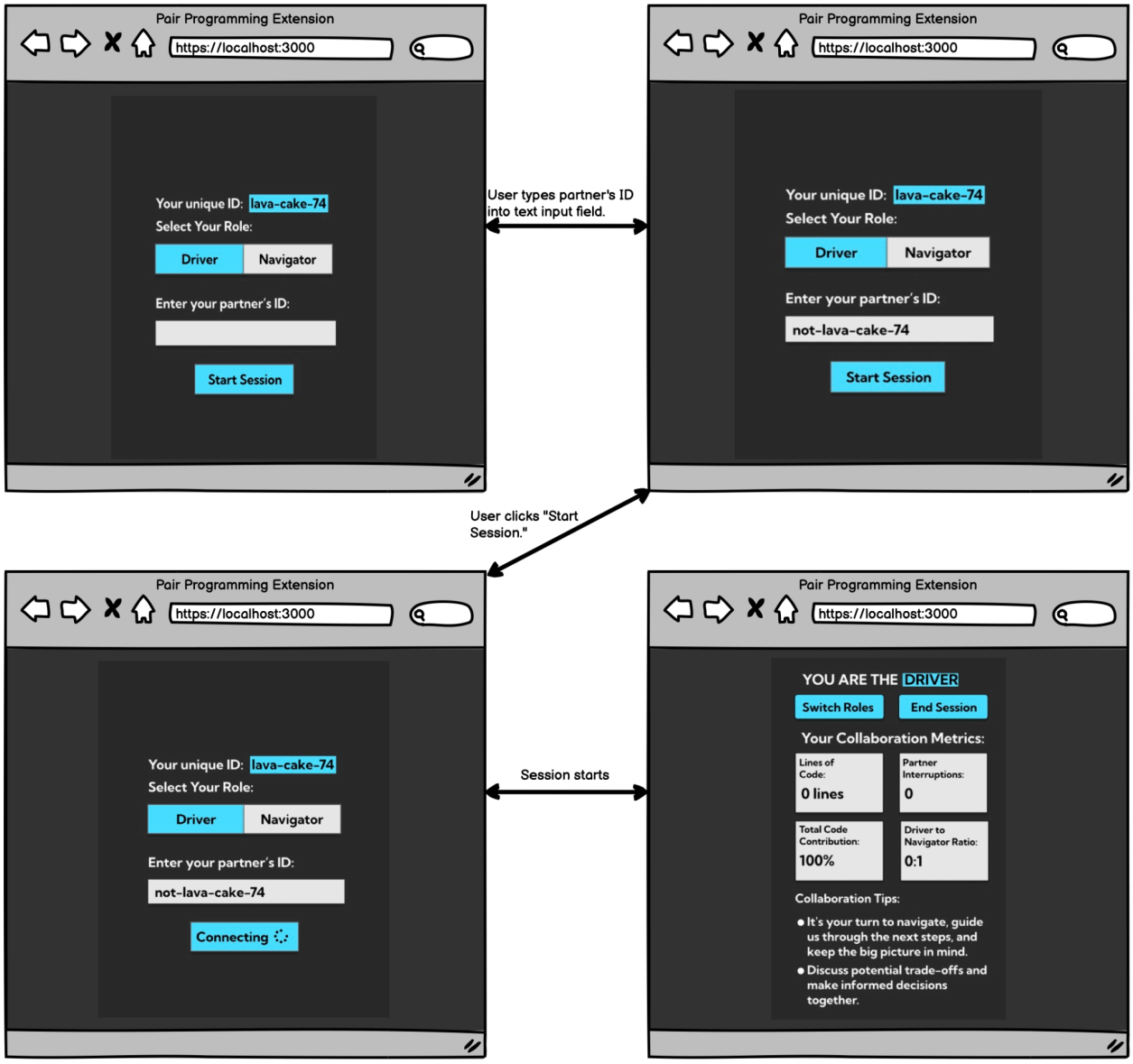


Figure 25: Start Session As Driver