STeMS Backend Services User Guide

University of Maryland Global Campus

SWEN 670

Summer Team B

Version 1.0

August 5, 2023

Version	Issue Date	Change Description
0.1	07/23/2023	Initial Revision
1.0	08/05/2033	Milestone 4 Submission

Introduction 1
Overview 1
Project Documents 1
Stakeholders 2
Definitions, Acronyms, and Abbreviations 2
Hardware Requirements 4
Software Requirements 4
Getting Started 5
Cloning the Repository 5
Opening the Project in Visual Studio Code 6
Setting up an .env File 8
Creating an Open AI API Key for Development 9
Emulator Settings 10
Starting up the Browser Extension Service (BESie) 11
Configuring ConvoBuddy or backend_test_utility to Connect To Local BESie 11
Running Unit and Integration Tests 12
Unit/Widget Testing 12
Integration testing 13
Features of the Application Backend 16
Importing backend_services_exports.dart 16
Agent Class and Initialization 16
Recordings API 17
Transcription and Diarization API 18
Form Filler Browser Extension API 20
Browser Extension API Lifecycle 20
App Instance Codes 22
Browser Extension Requests 22
ChatGPT Summary API 23
Food Orders Summary API 24
STML Reminders API 25
AWS Transcription Services 25
Troubleshooting 27

AWS Transcript not Received..... 27
Debugging BESie with the Test Page 27
References..... 29

Introduction

The STeMS Backend Services interface with ChatGPT and AWS Transcribe to support methods used to assist those suffering from Short Term Memory Loss, summarize conversations, provide order summaries that are useful for restaurant wait staff, and fill of web forms through a browser extension connected to the ConvoBuddy mobile app through a WebSocket server. The services described here provide a variety of functionality and a basis for implementing additional use cases.

Overview

The purpose of this document is to act as a reference guide for developers intending to utilize the STeMS Backend Services. The STeMS backend services application programming interface (API) consists of the backend_services cross-platform Dart library package and supporting Browser Extension Service (BESie). Instructions on how to configure the user's development environment are outlined as well as the features made available by the backend_services package. The troubleshooting section contains problem-solving steps to resolve common issues that may arise while using the STeMS backend services.

Project Documents

To adequately manage, control, and deliver the STeMS backend application, a suite of vital documents is created including this User Guide. Artifacts that are provided within the document package comprise important information for the application's ongoing support and operation throughout its life cycle. The documents created with the specific project milestones are included within the "Project Documentation Package" in the following table.

Table 1. Project Documentation Package

	Document	Version	Date
1	Project Management Plan (PMP)	4.0	8/04/2023
2	Software Requirements Specification (SRS)	4.0	7/24/2023
3	Technical Design Document (TDD)	3.0	7/24/2023
4	Programmers Guide (PG)	2.0	7/24/2023
5	Deployment and Operations (DevOps)	2.0	8/04/2023
6	User Guide (UG)	1.0	8/05/2023
7	Test Report (TR)	1.0	8/05/2023

Stakeholders

The project stakeholders linked to this undertaking are included in the following table:

Table 2. Project stakeholders

Stakeholder Name	Project Role
Mir Assadullah	Professor, Program Manager
Roy Gordon	Project Mentor
Robert Wilson	DevSecOps Mentor
Babers, David	Project Manager
Nagy, Jonathan	Team Lead
Babers, David Hicks, Collin Iko, Benny Mccrillis, Scott Thomare, Amol	Developers
Ben Lakbir, Mohamed De Souza, Robson Huynh, Johnny Mekonnen, Kidanu Nagy, Jonathan	QA/Testers

Definitions, Acronyms, and Abbreviations

This section covers all definitions, abbreviations, and acronyms used throughout this User Gide document.

Table 31. Abbreviations and acronyms used throughout the User Guide

Abbreviation/Acronym	Definition
ADB	Android Debug Bridge
APIG	Application Programming Interface Gateway
AI	Artificial Intelligence
API	Application Programming Interface
ATS	Amazon Transcription Services
AWS	Amazon Web Services

BESie	Browser Extension Service
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
I/O	Input and Output (computer hardware)
JDK	Java Development Kit
JSON	JavaScript Object Notation
PC	Personal Computer
QA	Quality Assurance
SDK	Software Development Kit
S3	Amazon Simple Storage Service
STeMS	Short-Term Memory System
STML	Short-Term Memory Loss
STT	Speech-to-Text
URL	Universal Resource Location
UUID	Universal Unique Identifier
VM	Virtual Machine
YAML	Yet Another Markup Language

Table 42. Terms used throughout the User Guide

Term	Definition
App Instance Code	A code generated by the STeMS backend system to identify ConvoBuddy application instances and used with the Browser Extension.
ChatGPT	A language model developed by OpenAI that uses deep learning techniques to generate human-like responses in natural language conversations.
ChatGPT API	An interface provided by OpenAI for developers to integrate ChatGPT into their applications
Diarization	The application of an algorithm to separate individual speakers in an audio stream, so that a speaker can be identified with each utterance in a textual transcript.
Local Storage	Storage provided on a user's device, sandboxed on mobile operating systems for security.
Speech-to-Text (STT)	Recognition and translation of spoken language into text. Also known as ASR or computer speech recognition.
WebSocket	A technology used to provide two-way communication between a web client and a server, allowing event driven responses without having to poll for new messages.

Hardware Requirements

The hardware required for this application would be:

- A physical mobile device or emulator that has either an Android or iOS operating system
  - Android Lollipop (API Level 21) or later
  - iOS Version 11 or later
- A computer that has a browser that supports browser extensions, tested with Chrome 115 version

Software Requirements

The software required for this application would be:

- Android or iOS emulator software (if not using a physical device)
  - Android Emulator 32.1.13
  - iOS Simulator (Xcode 14)
- A browser that can support browser extensions such as Google Chrome, Firefox, or Microsoft Edge
- A Code Editor such as Visual Studio Code or Android Studio
- Android SDK Command Line Tools for Android development
- The Java Development Kit (JDK) 17 development environment for Java coding
- Flutter SDK (Version 3.10.5)
- Dart SDK (Version 3.0.5, included with Flutter SDK 3.10.5)
- Flutter DevTools (Version 2.23.1, included with Flutter SDK 3.10.5)
- Flutter and Dart extensions
  - Android SDK (SDK Level 21 (L) or higher)
  - Windows 10 or higher

Getting Started

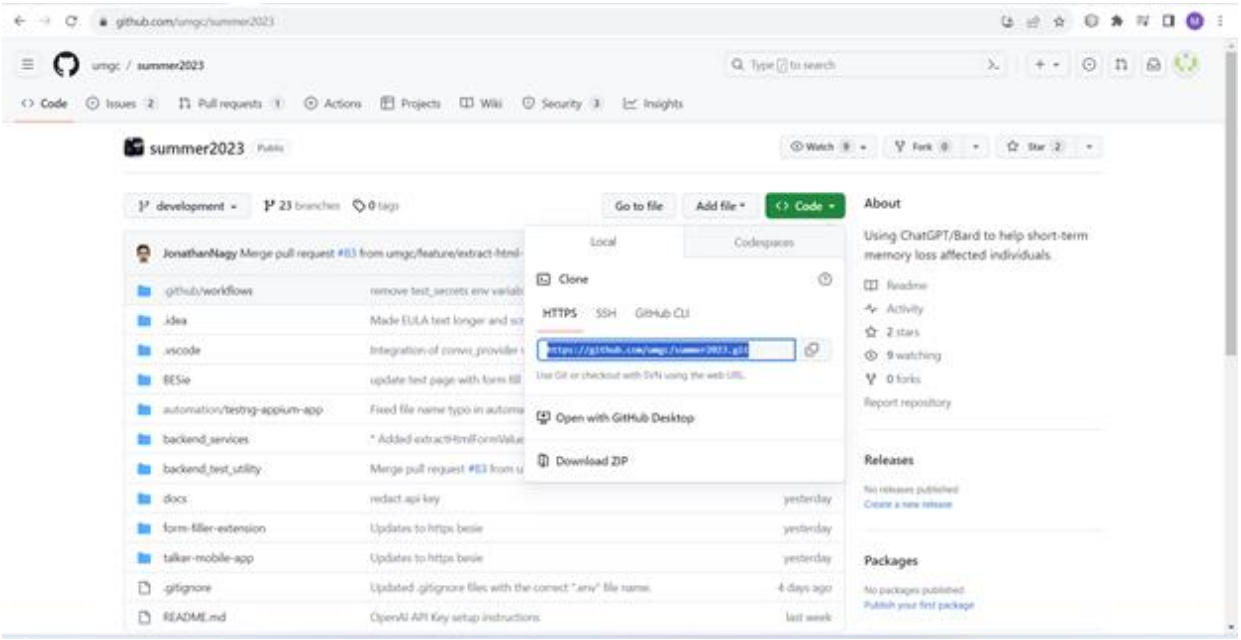
The following section discusses prerequisites to set up the development environment for use with STeMS Backend Services.

Cloning the Repository

To clone the summer2023 GitHub repository to get the application ConvoBuddy running, some instructions must be followed to get a copy transferred into a local machine as follows.

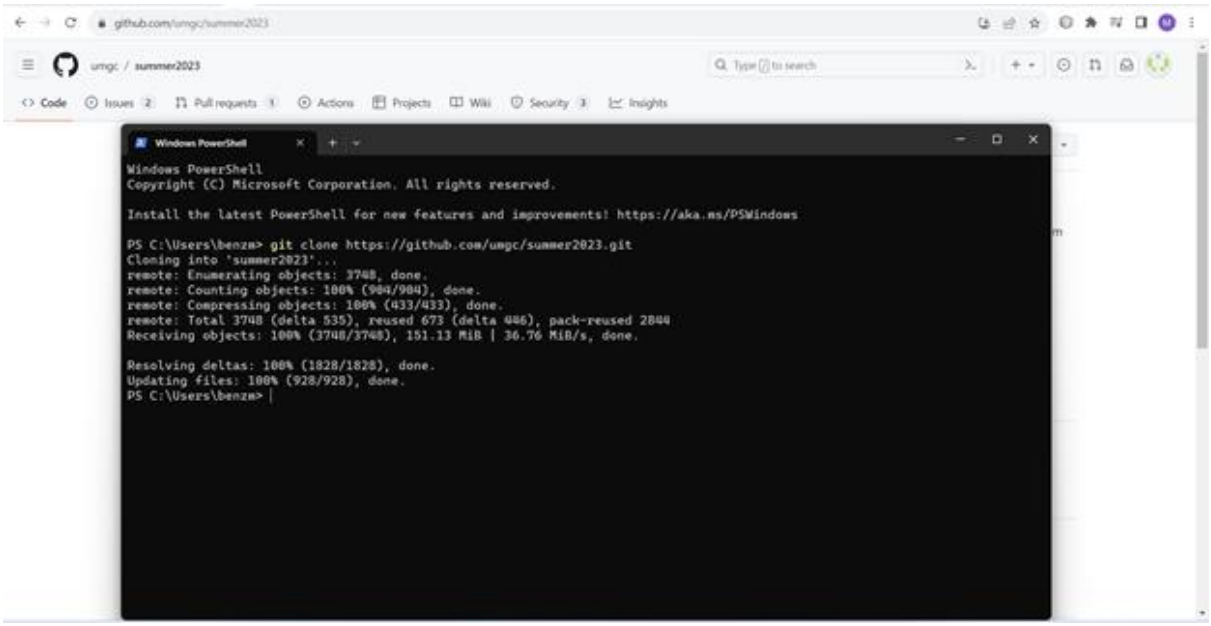
- Open GitHub and choose “umgc / Teams / Summer2023”.
- Select “Repositories” from the menu bar.
- Select “umgc/summer2023” link.
- Select the green [< > Code] button in the upper right corner.
- Select HTTPS.
- Copy the url link, https://github.com/umgc/summer2023.git, for the entire repository.

Figure 1. URL link for the entire repository



- Open up Command Prompt, Git Bash, Visual Studio Code Terminal, or Windows PowerShell in a local machine.
- Execute the git clone command; git clone https://github.com/umgc/summer2023.git.

Figure 2. Execution of git clone command on Windows PowerShell



Once the execution of the clone command is done, the local machine gets the entire repository - all files, all branches, and all commits (GitHub, 2022). The development branch is set active by default.

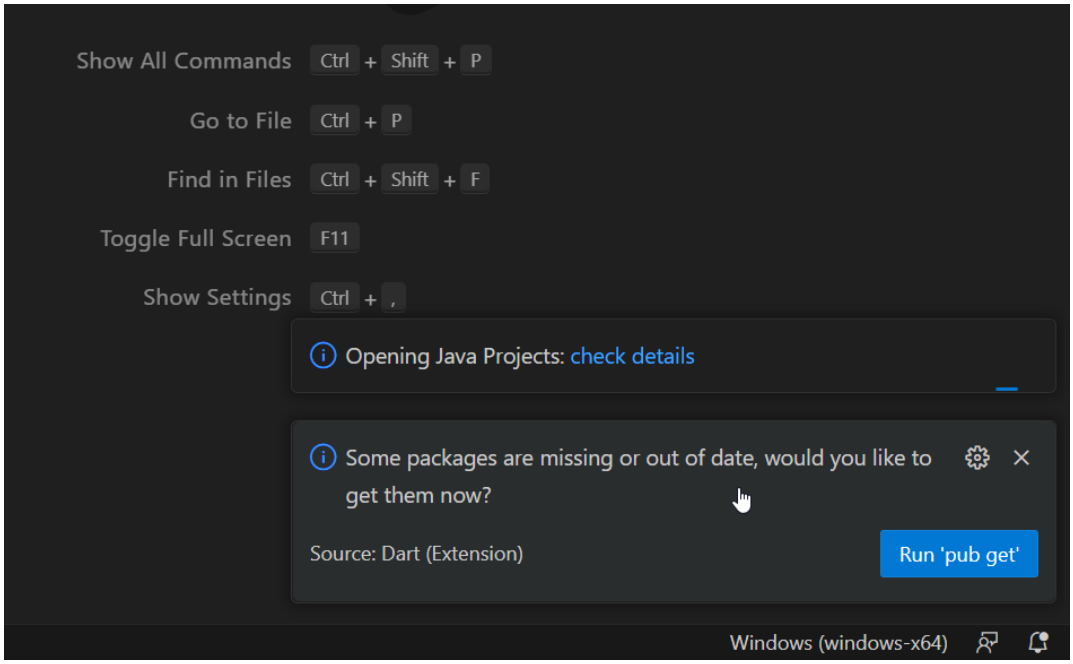
Opening the Project in Visual Studio Code

1. Open Visual Studio Code
2. Type `Ctrl-Shift-X` to open the Extensions Marketplace
3. Search for Flutter and install the official Flutter extension from dartcode.org

Figure 3. Install VS Code Flutter Extension

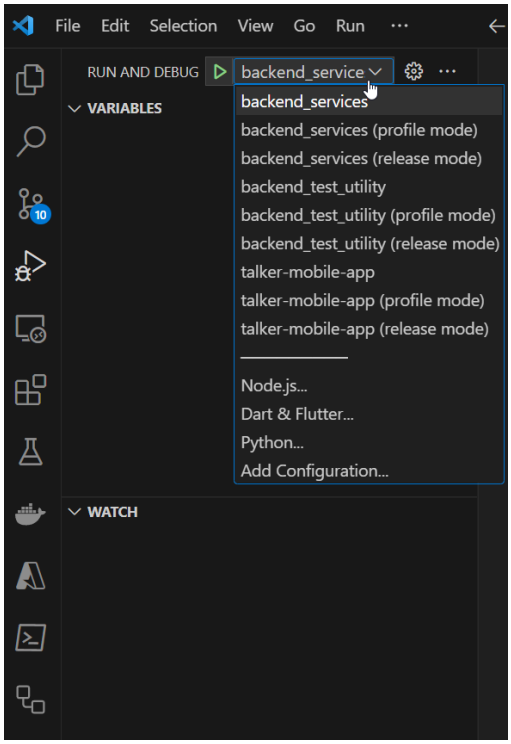
4. Open the File menu and select Open Folder…
5. Browse to the folder cloned through git, by default it is named summer2023.
   Note: Opening the root allows a developer to browse all the Flutter projects at once.
6. When a notification appears at the bottom right (see Figure 4), it can be clicked to update the package dependencies for all Flutter projects in the folder hierarchy. If the notification is dismissed or dependencies need to be updated later, `Ctrl-~` will open a terminal, the directory can be changed to the projects folder (such as "`cd backend_services`") and "`flutter pub get`" will download the Flutter package dependencies.

Figure 4. Run 'pub get' Notification



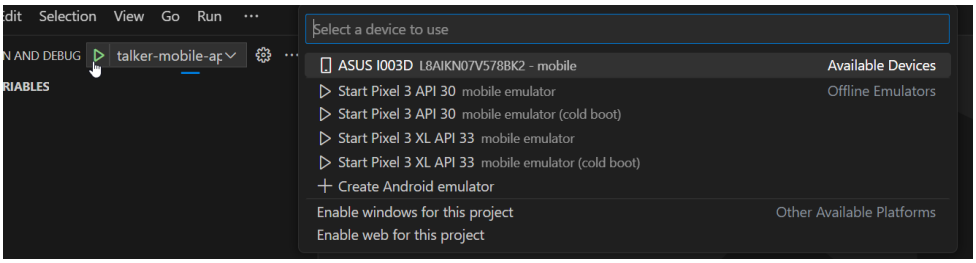
Either the talker-mobile-app or backend_test_utility can be debugged by going to the Debug tab and selecting one of the projects from the “RUN AND DEBUG” dropdown list.

Figure 5. Run and Debug Options



Once a project is chosen, an emulator or connecting to a physical device will need to be selected to run the app on. If a physical device is connected, it will be selected automatically.

Figure 6. Select Device



The “(cold boot)” options may be helpful if an emulator has crashed and is not responsive. Normally Android Emulator loads from a saved snapshot, but if the snapshot is invalid or contains a device in a bad state, a cold boot will start without loading a snapshot, “just like powering on a device” (Android Studio, 2023).

Setting up an .env File

ConvoBuddy and the backend_services package use a .env file deployed with the application to configure certain items such as AWS URLs, BESie topics, and the OpenAI API key. As the API key must be private, it cannot be pushed to GitHub. Thus the .env file is in the .gitignore and

is not to be checked in. To build and run the project with full functionality, a complete copy of the .env file is required. A shared copy of the base .env file can be found in the Team B folder in Teams. It is also provided here for convenience and future reference.

Figure 7. Example .env File

```
RECORDING_S3_BUCKET = 'https://testrecordingsswenv2.s3.amazonaws.com'
TRANSCIPTION_S3_BUCKET='https://transcriptsswenv1.s3.amazonaws.com'

# BESie on the internet
WS_URL = 'https://besie.servehttp.com/ws'
WS_CONNECTION_TIMEOUT_MS = '10000'
FORM_FILL_REQUEST_TOPIC = '/topic/form-model'
FORM_FILL_RESPONSE_TOPIC = '/app/filled-form'
TRANSCRIPT_RESPONSE_TOPIC = '/topic/transcript'

# Do not check into github! OpenAI will disable any keys found in a source repository on
# github.
# Set your OpenAI API key below. There are instructions on registering for a key in the
# "talker-mobile-app/README.md" file
# (https://github.com/umgc/summer2023/blob/development/talker-mobile-app/README.md).
OPENAI_API_KEY = 'openai-api-key-here'
```

Copy the above into a .env file in the talker-mobile-app to configure ConvoBuddy. Create a copy of the file in backend_test_utility folder to configure the backend_test_utility.

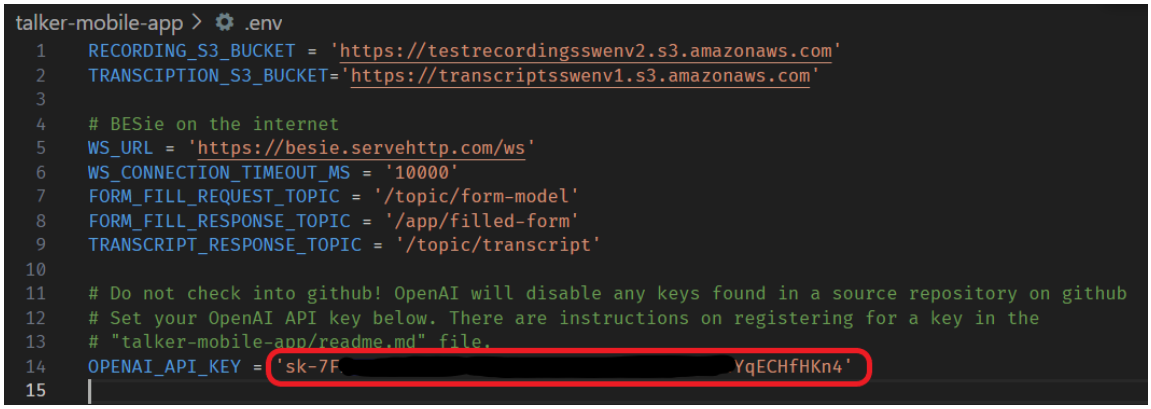
Creating an Open AI API Key for Development

This short video explains the process of creating a new key: https://youtu.be/EQQjdwdVQ-M?t=24.

The following outlines setting up a new key and adding it to the .env created with the instructions above.

1. Go to https://openai.com and Login or Sign up for a new account.
2. Click on API
3. At top right, click on Personal and then View API Keys menu item.
4. Click "+ Create new secret key" button
5. Give the key a name relating to ConvoBuddy or SWEN670
6. The new secret key will appear in a text box with a copy icon button next to it.
7. Copy the key and paste it into the .env file.

Figure 8. Populated .env File with API Key Redacted



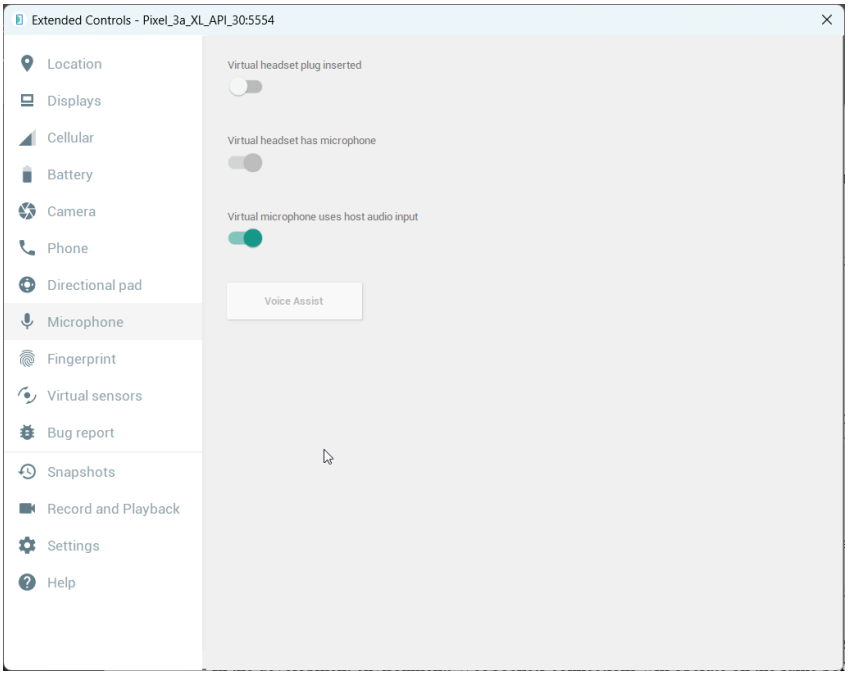
8. Save the key somewhere like OneNote where it will be safe. It cannot be retrieved once the dialog is closed. Additional keys can be made, however.
9. Click done.

A new account should have $5 in credits to start with, which is ample for testing with the ChatGPT model 3.5 ConvoBuddy uses.

Emulator Settings

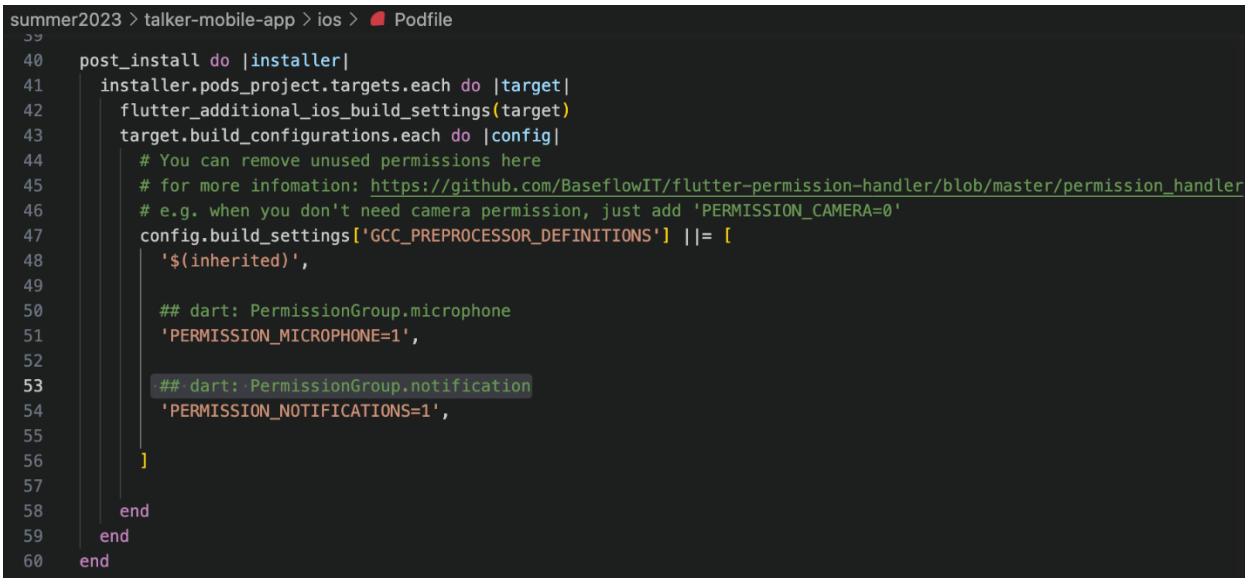
To develop with the backend_services package, either a mobile device (Android or iOS) or an emulator will be needed. To make use of the recording capability of ConvoBuddy, the microphone must be enabled and microphone source set to a valid input such as the host PC. Android Emulator has a Microphone settings tab in device settings. Enable “Virtual microphone uses host audio input” to allow a microphone connected to the host PC to be used as input.

Figure 9. Emulator Microphone Settings



For iOS, choose a device from menu item Hardware > Audio Input. The System option can be selected to direct Mac input to the Simulator (Apple, n.d.).

For iOS, all permissions are disabled by default. The ios/Podfile (untracked) must explicitly specify GCC_PREPROCESSOR_DEFINITIONS for the microphone to prompt the user for permission.



Starting up the Browser Extension Service (BESie)

The Browser Extension Service (BESie) is used for communication between the Flutter mobile app and the Form Filler Browser Extension. It is also used to send transcription results back to the mobile app from AWS Lambda service and AWS Transcribe. To start a development instance of BESie:

1. Open a Windows command prompt.
2. Change directory to BESie folder.
3. Execute "`mvnw.cmd clean package`" to build BESie on Windows or "`./mvnw clean package`" on Linux. Verify there are no errors before proceeding.
4. Execute "`mvnw.cmd spring-boot:run`" to start BESie on Windows or "`./mvnw spring-boot:run`" on Linux.

BESie will listen on port 8080 using HTTP protocol and hosts a test page at http://localhost:8080/. In the development environment, WebSockets connections will operate on the same port 8080.

Configuring ConvoBuddy or backend_test_utility to connect to local BESie

By default, the .env file in both talker-mobile-app and backend_test_utility is set up to use an instance of BESie on the internet. To allow ConvoBuddy or backend_test_utility to connect to BESie locally, the WS_URL property can be set to http://localhost:8080/ws. Then the following Android Debug Bridge command should be typed:

```
> adb reverse tcp:8080 tcp:8080
```

This will allow the currently connected Android mobile device or emulator to connect to BESie on the development machine.

Running Unit and Integration Tests

Tests help ensure that the built app continues to work as more features are added or existing ones are altered. There are three test types that Flutter supports; unit test, widget test, and integration test. A unit test verifies the behavior of a method or class. A widget test verifies the behavior of Flutter widgets without running the app itself. An integration test (also called end-to-end testing or GUI testing) runs the full app (Flutter, 2023.b).

Unit/Widget Testing

Unit tests are adequate to validate the behavior of a single function, method, or class. The test package provides the core framework for writing unit tests, and the flutter_test package provides additional utilities for testing widgets.

To perform unit testing, the mean features provided by Flutter test package fits these steps:

1. Add the test or flutter_test dependency.
2. Choose from the project lib folder a class to test.
3. Create a test file.
4. Write a test for each function for the chosen class.
5. Combine multiple tests in a group.
6. Run the tests.

To add the test package as a dev dependency, the following command should be executed on Terminal:

```
flutter pub add dev:test
```

To create a test file (classname_test.dart), select “test” folder from Explorer menu, right click, select New File, tap the test file; **classname_test.dart**, then hit Enter.

```
projectname_app/
  lib/
    classname.dart
  test/
    classname_test.dart
```

Tests are defined using the top-level test function, and it is easy to check if the results are correct by using the top-level expect function. Both functions come from the test package. To write the first unit test inside the test file, these actions must be taken:

```
// Import the test package and class
import 'package:classname_app/classname.dart';
import 'package:test/test.dart';
```

```
// then write a test for each function inside main method after test function
provided by the test package
void main() {
  test('What the test should be intended for', () {
// code to write
  });
} // then run the test
```

Once each single test passes, tests that are related to one another should be combined using the group function provided by the test package inside the main method.

```
import 'package:counter_app/counter.dart';
import 'package:test/test.dart';

void main() {
  group('What to test or meaningful message', () {
    test('first test', () {
      …;
    }); // end of first test
.
.
.
    test('last test', () {
      …;
    }); // end of last test
  }); // end of group test
} // then run the test
```

(Flutter, 2023.a).

Integration testing

This section demonstrates how to set up integration tests, how to verify if a specific text is being displayed by the app, how to tap specific widgets, and how to run integration tests. To perform the Integration testing in Flutter apps, these are the steps to follow:

1. Create an app to test which is the built project.
2. Add the integration_test dependency.
3. Create the test files.
4. Write the integration test.
5. Run the integration test.

1. Create an app to test

The project created, STeMS, is the application to test.

2. Add the integration_test dependency

To do so, use the **integration_test, flutter_driver, and flutter_test** packages for writing integration tests.

Add the following dependencies to the **dev_dependencies** section of the app's file **pubspec.yaml** . The location of the package should be the Flutter SDK.

```
# pubspec.yaml
dev_dependencies:
  integration_test:
    sdk: flutter
  flutter_test:
    sdk: flutter
```

3. Create the test files

Create a new directory, **integration_test**, with an empty **app_test.dart** file:

```
projectname_app/
  lib/
    main.dart
  integration_test/
    app_test.dart
```

4. Write the integration test

Now, write the integration test with the steps below:

1. Initialize **IntegrationTestWidgetsFlutterBinding**, a singleton service that runs tests on a physical device.
2. Interact and test widgets via the **WidgetTester** class.
3. Test the necessary scenarios.

```
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';
import 'package:introduction/main.dart' as app;

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();
  group('end-to-end test', () {
    testWidgets('Meaninful description of what to test',
        (WidgetTester tester) async {

      app.main();
// the rest of the code goes over here.
  });
}
```

5. Run the integration test

The process of executing integration tests depends greatly on the platform used. If the platform permits, it is plausible to test on mobile devices or web browsers.

5.1 Run on mobile

To test on a real iOS / Android device, first, connect the device and run the following command from the root of the project:

flutter test integration_test/app_test.dart

Or only specify the directory to run all integration tests:

flutter test integration_test

This command runs the app and integration tests on the target device.

5.2 Run on web

To test on a web browser (ie. Chrome), ChromeDriver can be downloaded (Download ChromeDriver). Then create a new directory named **test_driver** containing a new file named **integration_test.dart**:

```
import 'package:integration_test/integration_test_driver.dart';
Future<void> main() => integrationDriver();
```

Launch WebDriver, for example:

chromedriver --port=4444

From the root of the project, run the following command:

```
flutter drive \
  --driver=test_driver/integration_test.dart \
  --target=integration_test/app_test.dart \
  -d web-server
```

(Geek, 2022).

Features of the Application Backend

The features of the STeMS backend are made available through the Agent class. The APIs exposed include the Recording API, Form Filler Browser Extension API, ChatGPT Summary API, Food Orders Summary API, and STML Reminders API. Each API is implemented as a set of methods defined in the Agent class and exposed as Agent instance methods. The organization of the APIs in a single class provides ease of use and one time initialization of all STeMS backend service functionality.

Importing backend_services_exports.dart

For convenience, all the exposed STeMS backend services classes and methods can be imported into a client Dart file with a single import statement:

```
import 'package:backend_services/backend_services_exports.dart';
```

Agent Class and Initialization

The Agent class can be found in the backend_services package. The flutter_dotenv package is used to initialize environment variables used by backend_services. Both the backend_services package and flutter_dotenv package must be included in the client pubspec.yaml in order to use the STeMS backend services. The following snippet from the ConvoBuddy client app's pubspec.yaml is required to include the backend_services package, flutter_dotenv package, and .env file asset:

Figure 10. Snippet of pubspec.yaml referencing backend_services and flutter_dotenv

```
dependencies:
...
  # backend services package
  backend_services:
    path: ../backend_services
...
  flutter_dotenv: ^5.1.0
...
flutter:
...
  assets:
    - .env
...
```

Note the indenting and placement in indentation hierarchy, as they are both relevant for proper interpretation in YAML format.

The Agent class is designed around the client creating and initializing a singleton instance. For example, the ConvoBuddy mobile app creates an instance using the Flutter get_it package, but a global static instance will work as well. Instantiation of the Agent class requires a `userId` and `appDirectory` argument. Additional named optional arguments are part of the constructor signature but are intended for unit and integration testing only.

Figure 11. Agent Constructor Signature

```
Agent(this.userId, Directory appDirectory,
      {RecordingSelectionActivator? recordingSelectionActivator,
      JsonStorage? storage,
      WebSocketClient? webSocketClient})
```

Here is an example of Agent instance registration using by ConvoBuddy in its main.dart file:

Figure 12. ConvoBuddy main.dart Snippet

```
  Directory directory = await getApplicationDocumentsDirectory();
  await dotenv.load();
  getIt.registerSingleton<Agent>(Agent('convobuddy-app', directory),
      dispose: (agent) => agent.shutdown());
```

The Agent instance is registered as a singleton with "convobuddy-app" as the userId and the directory passed in from the results of `getApplicationDocumentsDirectory()`, an async method from the path_provider Flutter package. The "`await dotenv.load()`" line is necessary before calling the `Agent.initialize()` method of the Agent class. This call initializes the provider of environmental variables for the STeMS backend services, loading the environment variables from the .env file.

The `Agent.initialize()` method takes a `RecordingSelectionActivator` instance. The `RecordingSelectionActivator` is an interface that specifies a `Future<void> getSelectorCallback()` method. This is explained in detail in the Form Filler Browser Extension API section.

The `Agent.shutdown()` method closes the WebSocket connection and disposes of the WebSocket client.

Recordings API

The Recordings API handles access to recorded audio files and the metadata associated with those audio files. The bulk of the Recordings API is made available through the conversationProvider property on the Agent class. As soon as an Agent class is instantiated, the conversationProvider property is populated and available with all previously stored conversations and metadata loaded to memory. All updates to conversations through the ConversationProvider are written to local storage, and thus storage and in memory data remain consistent and persist across app usage sessions. The ConversationProvider class provides the following methods:

Table 5. ConversationProvider Methods

Method Name	Description
get conversations	Readonly property to return the list of all conversations.
get selectedConversation	Readonly property to return the selected conversation.
get sortingType	Readonly property to return current sort type.

Method Name	Description
get appDirectory	Readonly property to return the directory where conversations files and metadata are stored.
get conversationJsonPath	Readonly property to return the path of the conversations json file where all the conversation metadata is stored.
addConversation()	Add a conversation instance to the list of conversations.
removeConversation()	Remove a conversation instance from the list of conversations.
updateConversationTitle()	Update a conversation's title.
updateConversationTranscript()	Update a conversation's transcript data.
updateCustomDescription()	Update a conversation's description.
updateGptDescription()	Update a conversation's ChatGPT generated summary.
updateGptFoodOrder()	Update a conversation's ChatGPT food orders summary.
updateGptTranscript()	Update a conversation's human readable transcript.
setSelectedConversation()	Set the currently selected conversation.
filterConversations()	Returns a list of conversations that match the argument searchText.
setSortingType()	Set the currently selected sort type.

The `ConversationProvider` class includes the `ChangeNotifier` mixin, allowing Flutter widgets to observe state on a `ConversationProvider` instance using a Consumer widget (Flutter, n.d.a). When `ConversationProvider` notifies any listener—for example, Consumers of the `ConversationProvider`—will be notified on change to the instance and will perform a redraw of child widgets with the update `ConversatinoProvider` data. Object instances are attached to the widget hierarch using `ChangeNotifierProvider` at a root widget level, and then any `Consumer<ConversationProvider>` in the hierarchy are notified of changes. The following code from ConvoBuddy's main.dart shows the Agent's conversationProvider instance referenced in a ChangeNotifierProvider:

```
  runApp(ChangeNotifierProvider<ConversationsProvider>(
    create: (_) => getIt<Agent>().conversationsProvider,
    child: MaterialApp(
      ...
```

In the ConversationDetailsScreen, the Consumer widget wraps the body of the widget, allowing the widget to update whenever new data is received from the backend services and applied to a conversation managed by the ConversationProvider instance.

```
    return Consumer<ConversationsProvider>(
      builder: (context, provider, child) {
        return Scaffold(
          ...
```

Transcription and Diarization API

The following method encapsulates uploading a file to the S3 bucked by the transcription lambdas to initiate generation transcription.

Table 6, Methode name throughout the user guide document.

Method Name	Description
`audioFileUpload()`	Uploads a file to an Amazon Execute API endpoint which initiates the transcription process.

Ultimately, the transcription lambdas produce a JSON payload that is sent back to the client Flutter app through the BESie WebSocket server. The following example outlines the format:

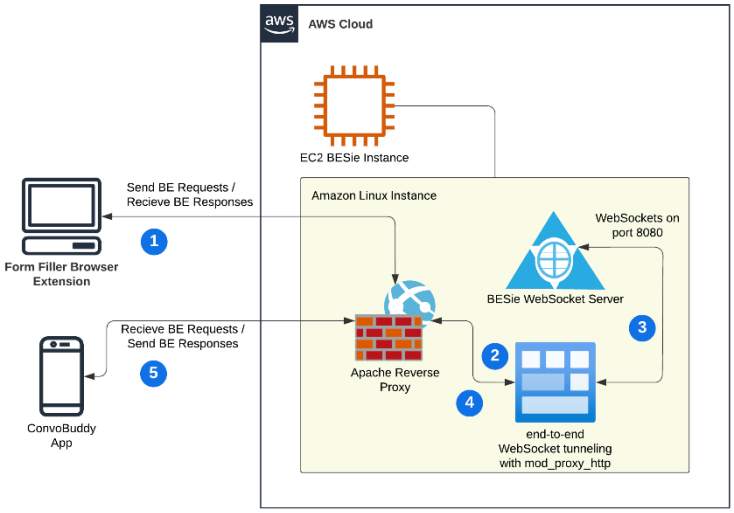
Figure 13. Transcription Result Payload

```
{
        jobName: "<recordingUUID followed by extra identifiers>",
        results: {
            transcripts: [{ transcript: "a human readable text transcript"],
            speaker_labels: {
            channel_label: "ch_0",
            speakers: 1,
            // Segments represent a sequence of items, which represent individual sounds and
            // includes a speaker label
            segments: [
                {
                start_time: "0.109",
                speaker_label: "spk_0",
                end_time: "4.389",
                items: [
                    {
                    start_time: "1.039",
                    speaker_label: "spk_0",
                    end_time: "1.659",
                    },
                ],
                },
            ],
            },
            // A list of items or sounds, content type and content (actual word or
            // punctuation) in the alternatives array and marked by a confidence rating
            items: [
            {
                start_time: "1.039",
                speaker_label: "spk_0",
                end_time: "1.659",
                alternatives: [
                {
                    confidence: "0.996",
                    content: "Testing",
                },
                ],
                type: "pronunciation",
            },
            ],
        },
        status: "COMPLETED",
    }
```

Form Filler Browser Extension API

The Form Filler Browser Extension API relies on a WebSocket server running on AWS. Instructions for setup and configuration can be found in the Deployment and Ops Guide. The relevant points in this guide are the event driven nature of the Browser Extension API methods, and the always connected interface between the web browser extension, the WebSocket server, and the Flutter mobile app.

Figure 14. BESie service infrastructure



The Form Filler Browser Extension API is divided into 3 parts. Browser Extension API lifecycle, managing the app instance code, and extracting form values.

Browser Extension API Lifecycle

Table7, Lifecycle Methods

Method Name	Description
Agent.initialize()	Opens a WebSocket connection to BESie and starts listening for form value extraction requests.
Agent.shutdown()	Closes and disposes WebSocket connection to BESie.

The Form Filler Browser Extension API provides a means for the Form Filler Browser Extension to interact with the Flutter app. The Flutter app must initialize the backend_services package before this can happen.

An app instance code is used to identify what instance of the Flutter app to pair with when using the browser extension. The app instance code is displayed in the Flutter app, and then the user can type this code into the browser extension to make a connection. Once a request to fill in a form is initiated from the browser extension, all configured Flutter apps are notified, but only the one with a matching app instance code responds.

The target Flutter app first responds by initiating a recording selection UI for the user to select a conversation to pull information from, and then once selected, to execute a ChatGPT query given the form fields from the browser extension and the transcript from the user selected recording. The results are sent back to the browser extension and the browser extension parses out the field values and fills in the field values on the displayed web page.

To initialize the Form Filler Browser Extension API, two calls must be made. First, `Agent.initialize()` must be called with an object implementing RecordingSelectionActivator. In the call to the `Agent.initialize()` method, the Agent class creates a WebSocket connection to BESie and begins listening on the "/topic/form-model" topic for form fill requests. Second, the `Agent.generateInstanceCodeIfNone()` must be called. The app instance code is a random 4-digit number used by the form filler browser extension to connect to the user's Flutter application instance. This call generates an app instance code if one has not been initialized on a previous run of the application. Once an app instance code is generated, it is stored in the app documents directory so that it can be loaded again on the next run and reused. This provides a better user experience, as the app will maintain the same app instance code for as long as it is installed.

The front-end application must provide a callback to the backend to trigger a conversation selection screen for the form filler browser extension. On selection of a conversation on this screen, the front-end can call `Agent.extractFormValues()` with the selected conversation Universal Unique Identifier (UUID) to kick off completion of the form fill request.

Expected flow is:

1. The form filler browser extension API receives a request over BESie on the "/topic/form-model" topic.
2. The API checks the app instance code and calls the previously provided `RecordingSelectionActivator` instance.
3. The activator is responsible for displaying the UI for the user to select a recording and calling `Agent.extractFormValues()` with the selected recording UUID.
4. The browser extension API will query ChatGPT and then send the results back to BESie on the "/app/filled-form" topic.

BESie receives messages on "/app/filled-form" topic and then broadcasts them to all listeners on the "/topic/filled-form" topic. When the form filler browser extension is opened, it starts listening on the "/topic/filled-form" topic for responses from the backend services.

This snippet from the ConvoBuddy ConversationSelectionScreen.dart shows a call to `extractFormValues()` when the user taps on a conversation item:

Figure 15. ConversationSelectionScreen.dart Snippet

```
            return ConversationListItem(
                conversation: conversation,
                onTap: () {
                  getIt<Agent>().extractFormValues(conversation.id);
                  Navigator.pop(context);
                });
```

The following is an example of a RecordingSelectionActivator implementation class.

Figure 16. ConversationSelectionActivator.dart class

```
import 'package:backend_services/interfaces/recording_selection_activator.dart';
import 'package:flutter/material.dart';

class ConversationSelectionActivator implements RecordingSelectionActivator {
  ConversationSelectionActivator(this.rootContext);

  BuildContext rootContext;

  @override
  RecordingSelectionActivatorCallback getSelectorCallback() {
    return () async {
      if (!rootContext.mounted) return;
      Navigator.pushNamed(rootContext, '/conversationSelection');
    };
  }
}
```

An instance of this class is passed to the `Agent.initialize()` method, and then the callback is executed when the backed services receive a message on the "/topic/form-model" topic from the form filler browser extension. The `context.mounted` check verifies that the context is still mounted. This is important when a UI method is executed asynchronously, or in this case, executed as a callback in response to an external event. The mounted property will indicate "whether the context is still valid" (Flutter, n.d.b). Interacting with an unmounted `BuildContext` could cause crashes that are difficult to debug (Dart, n.d.).

App Instance Codes

Table 8. App Instance Code Methods

Method Name	Description
Agent.generateInstanceCodeIfNone()	Generates a new app instance code if one does not exist and saves it to local storage.
Agent.getInstanceCode()	Gets the instance code saved to local storage.

Browser Extension Requests

Form Fill Browser Extension API requests are sent to the Flutter app through BESie. The BEService class is responsible for parsing a BERequest object from the WebSocket frame and

storing it in memory for when the user has selected a conversation as the source from which to extract information.

Figure 17. BERequest class

```
class BERequest {
  String pin; // required
  dynamic form; // optional, either form or formHtml should be defined
  String? formHtml; // optional, as above
  bool shouldExtractFromHtml; // optional, defaults to false
}
```

The shouldExtractFromHtml boolean allows selection of either a JSON list of fields or a HTML form snippet as the source for the list of fields to extract. If the boolean is left off, the API defaults to field list JSON mode.

Figure 18. BERequest Field List JSON Example

```
{
    pin: "8347",
    form: [
        { "firstName": "text" },
        { "lastName": "text" }
    ]
}
```

Figure 19. BERequest Field List HTML Example

```
{
    pin: "8347",
    formHtml: "<form><input type="text" name="firstName"><input type="text"
name="lastName"></form>",
    shouldExtractFromHtml: true
}
```

In both examples, the user would be prompted to select a conversation through the `RecordingSelectionActivator` callback, and then once selected the client would call `extractFormValues` with selected recording UUID. The Browser Extension API would then send a request to ChatGPT with a prompt constructed from the list of field values or the HTML snippet, and the transcript from the selected recording. Once the Browser Extension API receives a response from ChatGPT, in the requested JSON format, it sets the response to the payload property of the `BEResponse` object and sends it to BESie on the "`/app/filled-form`" topic. From there, the Form Filler Browser Extension can parse it and apply it to the web form targeted for filling.

ChatGPT Summary API

The ChatGPT Summary API is simple in concept as it focuses on distilling an entire conversation into a succinct, easy to understand pair of sentences. Within the GptCalls.dart file,

the `getOpenAiSummary()` method only requires a transcript and the user profile information for the ChatGPT account owned by the user to begin.

After referencing the appropriate API key through _openAIApiKey, the method compiles the query to be sent to ChatGPT by combining a static prompt, descriptionPrompt, which requests a two-sentence high-level summary of the transcript, the user profile information and the full transcript before sending the request off to ChatGPT. The response from ChatGPT is then stored in a list containing the user and the full query sent to ChatGPT.

Finally, the query is sent to ChatGPT to open a chat with GPT 3.5 Turbo and the full contents of the message containing the prompt and transcript with the response being saved to record.

Food Orders Summary API

The Food Orders Summary API takes recordings of interactions between a server and their customer or customers. Within the agent.dart file, the `getOpenAiFoodOrder()` method is responsible for identifying the appropriate recording needed to be summarized.

The `getOpenAiFoodOrder()` method calls the `getRestaurantOrder()` method from the GptCalls.dart file to query ChatGPT using a common format of a static prompt for ChatGPT to understand the context of the request and then sends the transcript of the conversation for parsing and summary. The `getRestaurantOrder()` method utilizes a _logger to keep a record of which prompt and transcript combination has been queried by ChatGPT. There is also a running list of all restaurant order prompts that are sent and records the user who initiated the request as well as the entire prompt and transcript as the context of that particular request. The method then creates a connection with ChatGPT using the free tier offering, ChatGPT 3.5 Turbo, and sends the message that includes the contextual prompt along with the transcript for analysis which should return the list of choices made by the restaurant customers as shown below:

Figure 20. Snippet of getOpenAiFoodOrder method.

```
final completion = await OpenAI.instance.chat
    .create(model: "gpt-3.5-turbo", messages: messages);

return completion.choices[0].message.content;
```

If a server runs into a situation where after an order has been recorded, they decide to change their order, the `updateGptFoodOrder()` method from the conversation_provider.dart file will take as arguments the ID of the original order, as long as the conversation ID is valid. Then, using the `gptFoodOrder()` method from the conversations.dart file that establishes the conversations class, which reads the information from the Json containing a string for the food order itself. Once the new food order has been saved to the object, all dependent listeners are notified, and the conversation is then written to an updated Json file as seen below:

Figure 21. UpdateGptFoodOrder method from the conversation_provider.dart file.

```
void updateGptFoodOrder(String id, String newGptFoodOrder) {
  Conversation? conversation =
      conversations.firstWhereOrNull((convo) => convo.id == id);
  if (conversation == null) {
    return;
  }
  conversation.gptFoodOrder = newGptFoodOrder;
  notifyListeners();
  writeConversationsToJsonFile();
}
```

STML Reminders API

The STML reminders API creates and compiles a list of reminders to be generated by ChatGPT. The STML reminders endpoint validates the transcript. From the transcription, ChatGPT queries the provided audio file and generates a reminders prompt. This prompt is then sent to OpenAI for inferencing. As a result, formatted data is returned which contains a list of reminders with times carrying over from the query.

Figure 22. getOpenAiReminders method from agent.dart file.

```
final recordingTranscript = getRecordingTranscript(recordingGuid);
final recordedDate = getRecordingDate(recordingGuid);

final gpt = GptCalls(EnvironmentVars.openAIApiKey);
final completion = await gpt.getReminders(recordingTranscript, '',
    recordedDate);
```

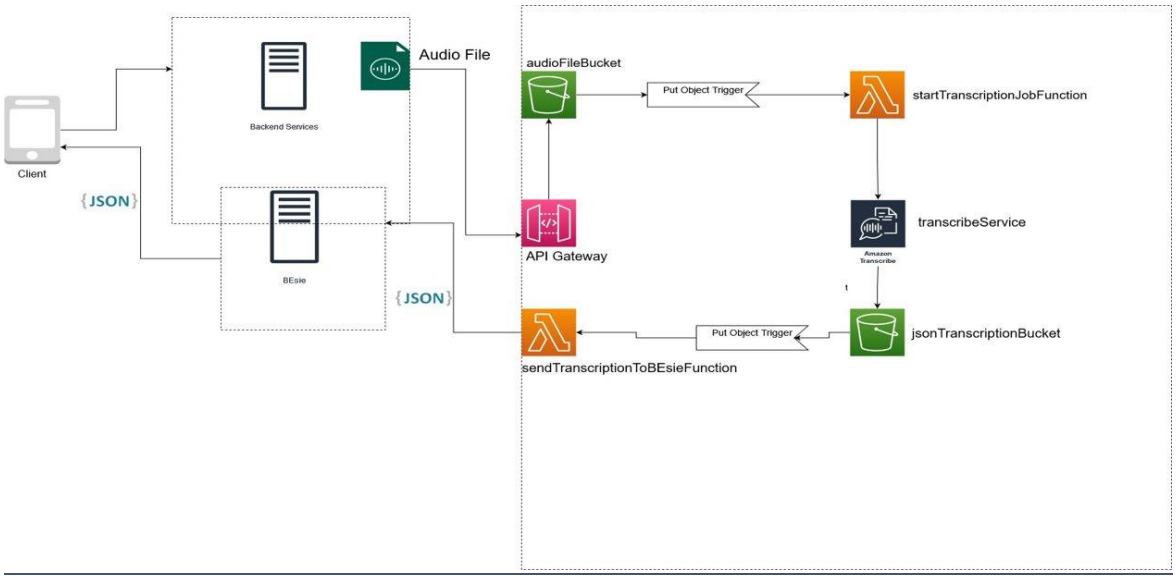
In agent.dart of the backend_services folder, the method getOpenAiReminders() method is defined. First, it gets the transcription by calling the getRecordingTranscript() method. Then, a call to ChatGPT is prepared by retrieving the OpenAI key which should be set in the .env file. The list of reminders is returned, and the reminders are converted into JSON as part of the conversation for the frontend in the reminders transmog section.

AWS Transcription Services

The AWS Transcription Services (ATS) is used to create transcripts and send the audio file data from the Talker App and generate a transcript of a recording. The Talker App calls the audioFileUpload function from the backend services library. This function takes the File path of the audio file, and the sends the audio bytes via an HTTP request to the AWS hosted API endpoint, or API Gateway (APIG). APIG then takes the filename and associated data and creates an object in an S3 bucket. This then triggers the convoBuddyStartTranscript lambda function that

starts the transcription service using AWS Transcribe (AWST). AWST then creates the JSON object of the transcribed audio file and places the transcript JSON file in a separate s3 bucket. This put action then triggers the convoBuddySendTranscript lambda function that reads the JSON file contents, parses the data, and sends it via a websocket request to BESie.

Figure 23. AWS Transcribe Services Architecture



Troubleshooting

This section provides instructions on how to work around common issues or debug unexpected behavior when working with the STeMS Backend Services.

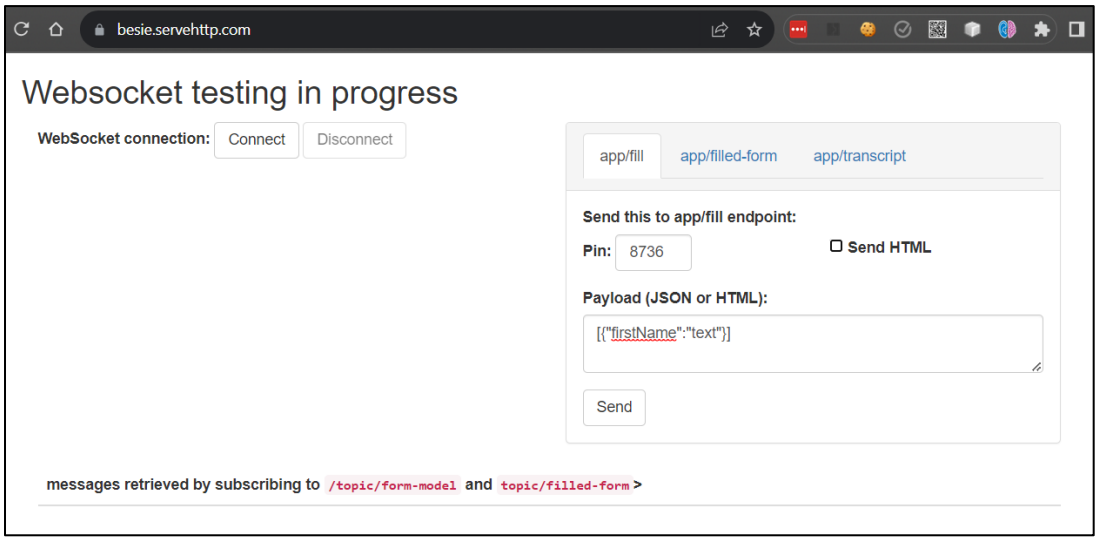
AWS Transcript not Received

Due to limitations in the WebSocket server, longer transcripts will sometimes fail to be sent back from AWS transcription lambda functions. In testing, this limit can be hit when transcribing a recording anywhere from 30 seconds to 1 minute in length. In order to ensure a transcript is generated, keep recordings short and make multiple recordings if necessary.

Debugging BESie with the Test Page

The Browser Extension service (BESie) developed by Team A includes a test page where all messages sent through a WebSocket connection can be viewed. Also, an interface to generate messages for the 3 broadcast topics is located at the top right of the page (or beneath the header and connect/disconnect buttons on mobile or narrow browser windows).

Figure 24. BESie Test Page



There is a tab for each of the three topics, mapping to 3 send endpoints, which are each mapped to a broadcast topic:

Table 9. Test Page Tab to Topic Mappings

Tab Name	Send Topic	Broadcast Topic
app/fill	/app/fill	/topic/form-model
app/filled-form	/app/filled-form	/topic/filled-form
app/transcript	/app/transcript	/topic/transcript

Messages sent from the app/fill tab will be received by listening Flutter apps and will initiate a recording selection callback, and if a recording is selected a ChatGPT prompt and response

which is fed to the “/topic/filled-form” topic. A message from the app/filled-form tab will send a message to the browser extension via the “/topic/filled-form” topic. Finally, a message from the app/transcript tab will be sent to Flutter apps listening on “/topic/transcript” and will update the transcript for the recording indicated by the provided recording UUID. Any message sent or received externally will be shown below the “messages retrieved by subscribing…” line (including topic/transcript though not indicated).

References

Android Studio (April 12, 2023). Snapshots. Retrieved August 5, 2023, from https://developer.android.com/studio/run/emulator-snapshots

Apple. (n.d.). Set the source for audio input and the device for audio output. Retrieved August 4, 2023, from https://help.apple.com/simulator/mac/current/#/deve63797392

Dart. (n.d.). use_build_context_sychronously. Retrieved July 30, 2023, from https://dart.dev/tools/linter-rules/use_build_context_synchronously

Flutter. (2023.a). An introduction to unit testing. Retrieved July 30, 2023, from https://docs.flutter.dev/cookbook/testing/unit/introduction

Flutter. (2023.b). Integration testing. Retrieved July 30, 2023, from https://docs.flutter.dev/testing/integration-tests

Flutter. (n.d.a). Simple state management. Retrieved August 1, 2023, from https://docs.flutter.dev/data-and-backend/state-mgmt/simple

Flutter. (n.d.b). Mounted property. Retrieved July 30, 2023, from https://api.flutter.dev/flutter/widgets/BuildContext/mounted.html

Geek, N. (February 1, 2022). How to run integration tests on Flutter apps. Retrieved July 31, 2023, from https://www.browserstack.com/guide/integration-tests-on-flutter-apps

GitHub. (August 2022). Git Clone. Retrieved July 30, 2023, from https://github.com/git-guides/git-clone