FALL 2024

UNIVERSITY OF SOUTHERN CALIFORNIA
CSCI401 CAPSTONE PROJECT- GROUP 11

LAFD HELP DESK AI CHAT BOT

LAFD Help Desk AI ChatBot

FALL 2024

OVERVIEW

This project, developed during the Fall 2024 semester of the USC CSCI401 Capstone course, focuses on creating an AI-powered chatbot solution for the Los Angeles Fire Department (LAFD) Help Desk. Traditionally, incoming service calls have been handled either by on-duty agents or through a basic web-based form, resulting in bottlenecks and increased pressure during high-volume periods. To address these challenges, this application leverages advanced, cloud-based technologies—primarily within the Microsoft Azure ecosystem—to provide a more efficient and scalable approach to managing technical service requests.

At its core, the chatbot utilizes Azure's speech-to-text and text-to-speech capabilities to interact with callers in real time, translating voice input into actionable data and providing audio responses. The Azure Call Automation API and Azure Event Grid facilitate seamless call processing, routing, and event management, while Azure Communication Services enable automated emailing, calling, and various other communication functionalities necessary for ticket creation and status updates. Additionally, the system integrates with a web server backend, allowing for continuous two-way communication between the chatbot and the Azure-based services handling the calls.

Building this application required significant research and practical learning beyond the standard curriculum. The team successfully met its objectives within the project's timeframe, establishing a working prototype that can alleviate the workload on human agents and enhance the overall efficiency of the help desk. Given the scale and complexity of the LAFD call center environment, further development and refinement are recommended. Future iterations may be undertaken by subsequent student groups or through extended collaboration, with details of these recommendations outlined in a dedicated section of this documentation.

NOTE: It is recommended to read this document in **MacOS Pages.** Image galleries may not generate correctly on other platforms.

I. MICROSOFT AZURE AND SETUP

1. OVERVIEW AND RESOURCE ALLOCATION

Before running any part of this application, you must properly configure and provision the necessary Azure resources. Microsoft Azure is a leading cloud computing platform that offers a vast range of services—from basic infrastructure and storage to advanced AI and analytics. Its flexibility, global reach, and integrated security make it a go-to choice for organizations looking to scale dynamically, reduce on-premises infrastructure costs, and rapidly deploy new applications or services. In this project, Azure's suite of AI and communication services is central to enabling real-time speech processing, automated call handling, and seamless event-driven communication.

All Azure credentials and connection strings must be obtained and securely stored within the codebase or its configuration files prior to executing the application. Specifically, an Azure resource group should be allocated with the following key components:

Setting Up a Microsoft Azure Account

Visit the Azure Portal: Go to https://portal.azure.com.

Sign In or Create an Account: Use your Microsoft account to sign in. If you don't have one, follow the on-screen prompts to create a Microsoft account.

Start Free Trial or Purchase a Subscription: Azure offers a free trial with credits for new users. Click the "Start free" button and follow the instructions, which will include providing a payment method for identity verification. Existing or enterprise users can log in with their organizational Azure subscription.

Verify Identity: Provide a valid phone number and payment method as required. You will not be charged unless you upgrade or exceed the free credit amount.

Access the Azure Portal: After verification, you'll land in the Azure Portal. From here, you can create, manage, and monitor all the Azure resources needed for this application.

Create resource group: A resource group is a way for Azure to group related resources for a given project. Create a resource group to hold all your future resources. You will refer to this resource group whenever a resource is allocated.

Azure Resource Group and Required Services

All Azure credentials and connection strings must be obtained and securely stored within the codebase or its configuration files prior to executing the application. Specifically, an Azure resource group should be allocated with the following key components:

1. **Azure Communications Resource**
2. **Azure Email Communication Resource**
3. **Azure Language Resource**
4. **Azure AI Multi-service Resource**
5. **Azure Event Grid Subscription**

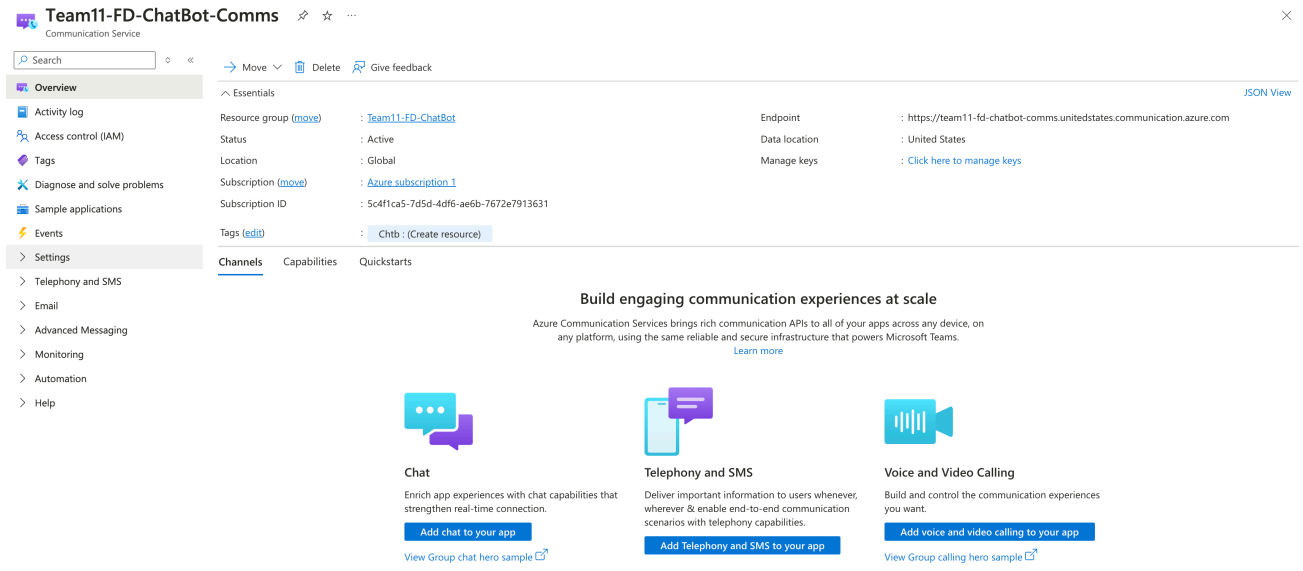
The following sections will provide step-by-step instructions for setting up each resource, maintaining secure access keys, and understanding how each service supports and enhances the chatbot's capabilities.

2. KEY MANAGEMENT AND CRITICAL SETUP

You will need several credentials from your resources, including *connection-strings* and *keys,* and *endpoints*.

Azure Communication Resources:

1. *ACS_CONNECTION_STRING:* Navigate to your Azure Communication Resource. On the side bar, navigate to *Settings*, and click on *Keys*. Your *connection-string* will be found here. This *connection-string* will be inserted into the *main.py* file at the designated place at or around line 194. Please ensure it is formatted correctly (shown below):

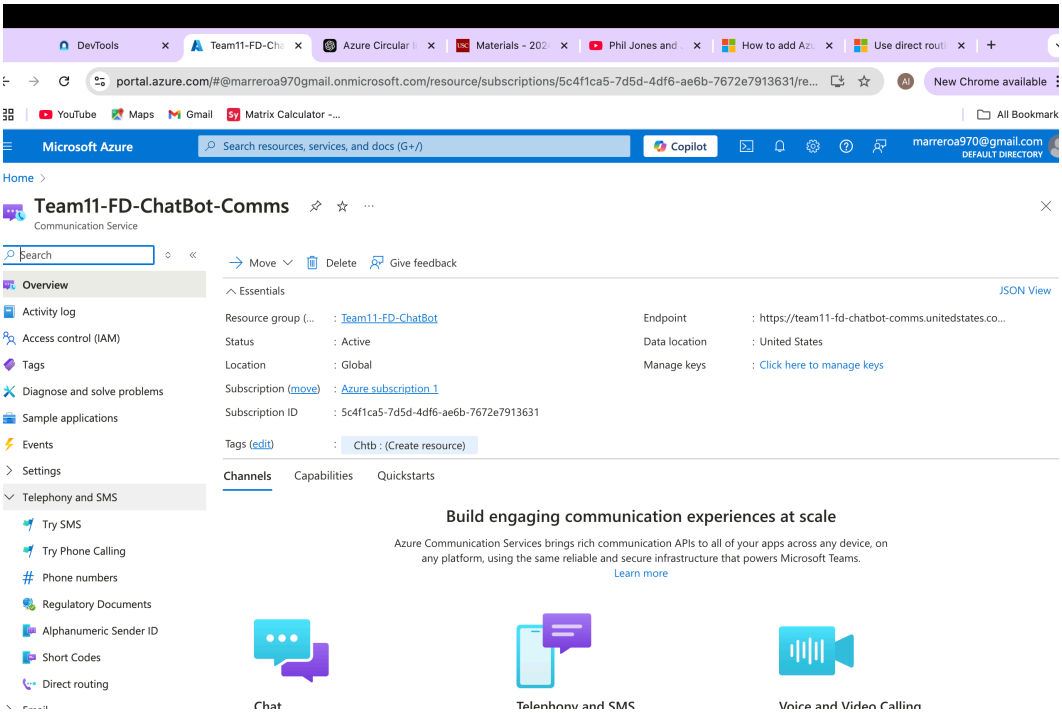


Scroll through images

2. *ACS_PHONE_NUMBER:* Navigate to your Azure Communication Resource. On the side bar, navigate to *Telephony and SMS*. Once here, click on *Phone Numbers*. At this point, you'll need to provision a toll-free number to act as the phone number that receives the service calls for the app. This project was completed using a single toll-

free number representing the Help Desk call-center. The app can certainly run this way, however the scope and scale of the full call-center phone infrastructure will no doubt require more complex telephone routing. For more advanced routing capabilities, please visit this link : *Use direct routing to connect to existing telephony service*

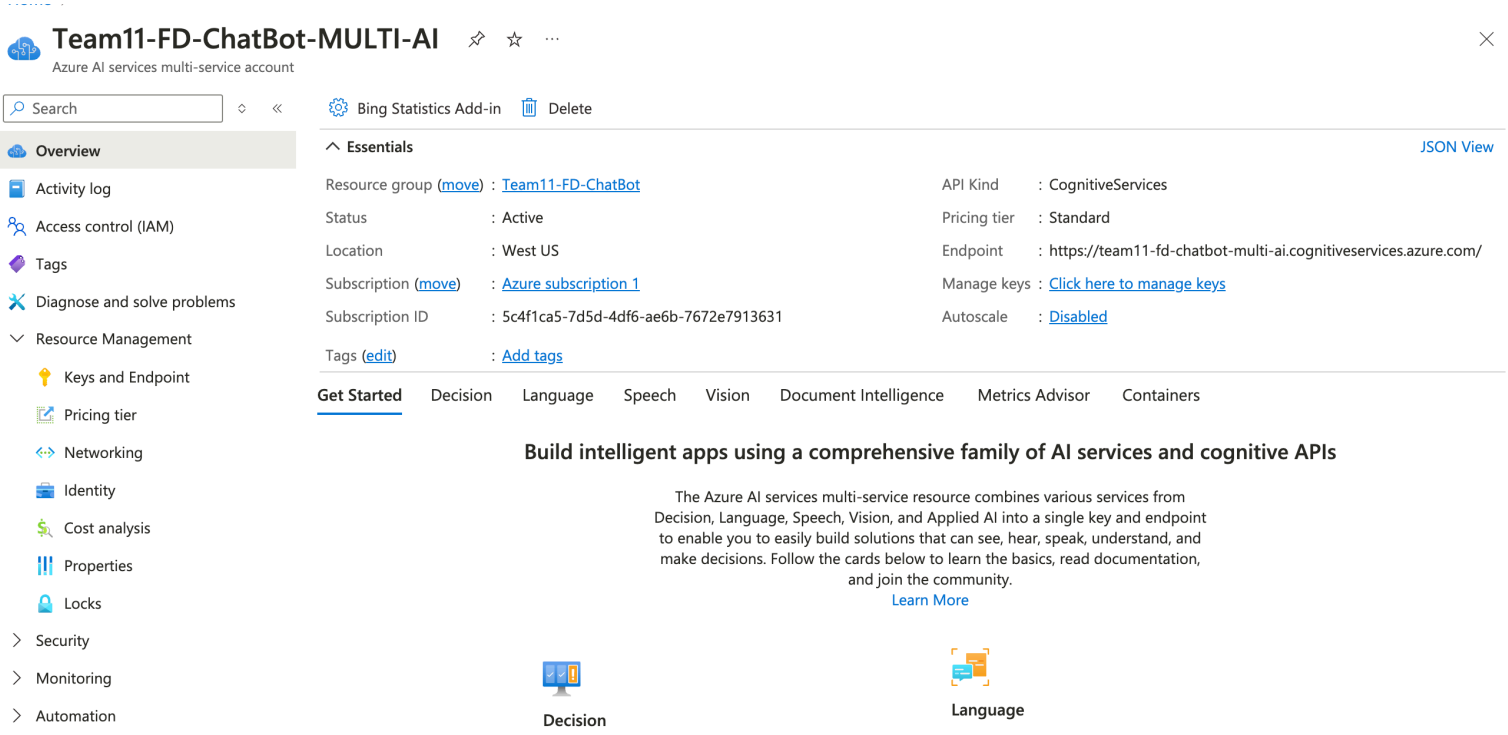
Once your telephone number is provisioned, save the phone number and insert it into the correct variable in *main.py*, at or around line 196 (shown below):



Scroll through images

Azure AI Multi-Service Resource:

3. *COGNITIVE_SERVICES_ENDPOINT*: Navigate to your Azure AI Multi-Service Resource. On the sidebar, navigate to *Key Management*. Click on *Keys and Endpoints*. Your *cognitive services endpoint* will be shown here. Please save this value, and insert it into *main.py* at its proper place, at or around *line 200* (shown below):

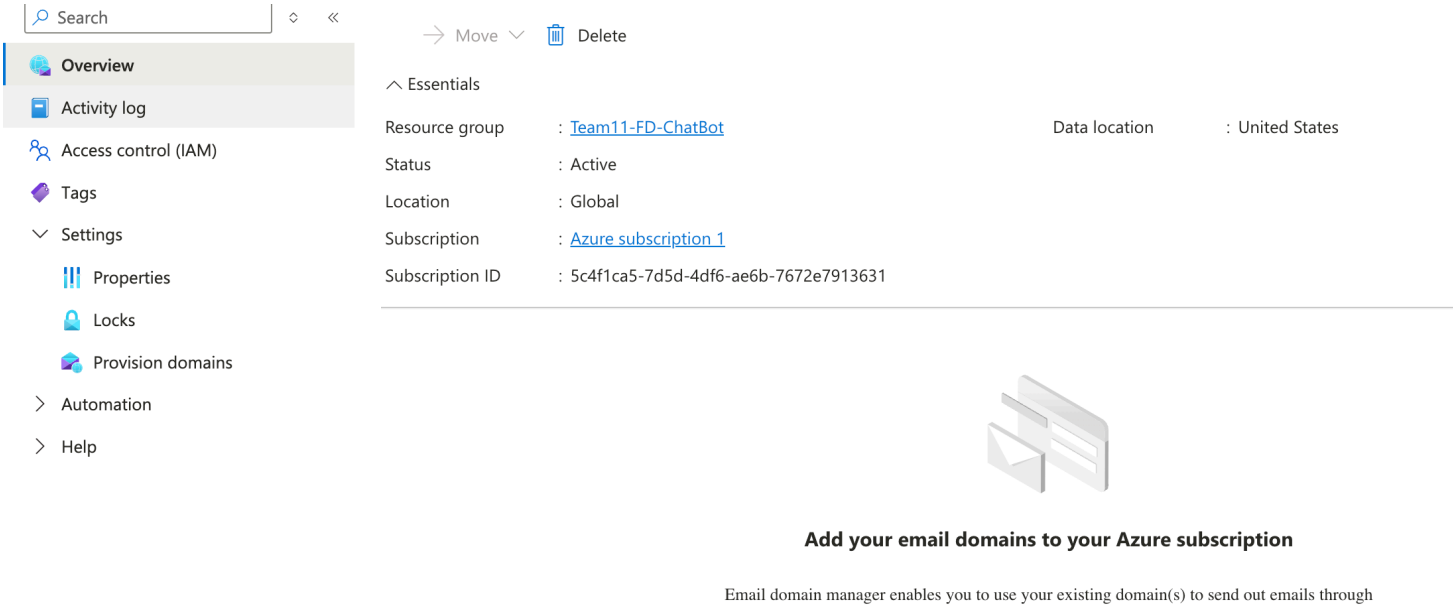


Scroll through images

Azure Email-Service Resource:

4. *SENDER_ADDRESS*: After creating your Azure Email Communication Resource, provision a domain to obtain your Azure email address. This will be the address that sends the completed ticket to the Microsoft Footprints email. This project provisioned a simple Azure domain, however there are different options for this. If you would like to explore more diverse options for email domains, please visit this link: *How to add Azure Managed Domains to Email Communication Service*.

After provisioning your domain, navigate to your resource, and click on *Settings* in the left hand toolbar. Click on *Provision Domains,* and select your domain. Now, on this page, navigate to *Email Services* on the left hand toolbar, and click on *MailFrom Address*. You will see your provisioned email address displayed. Save this email address, and insert it into the *SENDER_ADDRESS* variable in *main.py* at the correct spot, at or around line 201 (shown below):



Scroll through images

IMPORTANT! **:** Once this email resource is provisioned, you need to navigate back to your Azure Communication Resource (where you provisioned your telephone number), and connect your newly created email resource. Without doing this, the app will not function properly. Once at your Azure Communication Resource page, navigate to *Email* in the left hand toolbar. Click on *Domains,* and connect your newly provisioned email resource.

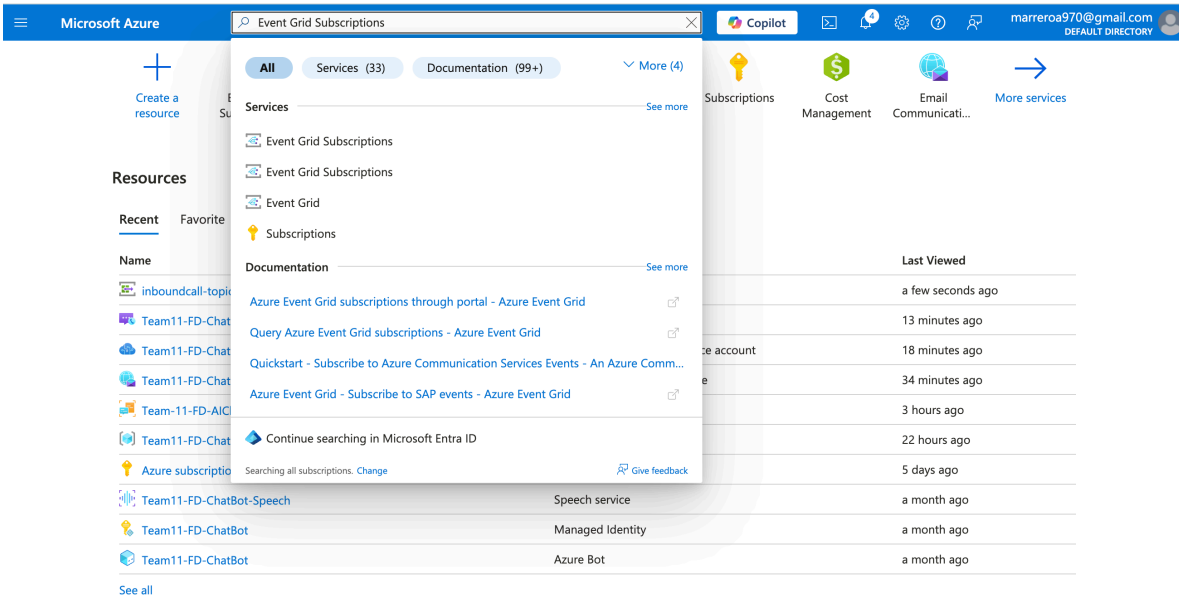
Azure Event Grid Subscription Resource

5. The Azure Event Grid maintains dynamic events related to your Azure Resources. In this case, it is used to keep track of incoming calls to your Azure telephone number so it can redirect the call to your app. This is a crucial aspect to the entire setup, so take your time and do this part correctly.

   Note: Your web server needs to provisioned and up and running with the app for this section to be completed. **You may want to go to the *Web Server* section of this documentation, and return to this section after your server is configured.**

   At the Azure Home Page, enter *"Event Grid Subscriptions"* into the search bar. Click on *Event Grid Subscriptions,* and click on the "+ *Event Subscriptions*" icon to add a new event subscription. Name your event "*IncomingCall",* and keep the Event Schema as is. For *Topic Types*, choose *"Azure Communication Services"* from the given list. Choose your subscription and resource group. For *Event Types,* type/choose "*Incoming Call"* from the list. For *Endpoint Type,* choose "*Web Hook"*. Click *Configure Endpoint*. Here, add your URL for your web-app server, followed by "/inboundCall", as shown in the images below.

   Important: *The app needs to be up and running for this to work*. Once your web server URL is configured and you've entered the correctly formatted endpoint, click *Confirm Selection*. Here, your resource will attempt to create a connection with your app. The app needs to be running for this to be successful. Once the connection is made and validated (done automatically by the program), you will see a confirmation on the Azure page. The app cannot receive phone calls until this is completed successfully.



Scroll through images

3. WEBSERVER SETUP AND APPLICATION STARTUP

You will need several credentials from your resources, including *connection-strings* and *keys*, and *endpoints*.

For development and testing purposes, this project uses a local web server environment facilitated by Microsoft's DevTunnels. A web server is essential because it enables the application to send and receive HTTP requests, communicate with Azure services, and handle event-driven interactions. While Azure services can host full-scale production web servers capable of handling enterprise-level traffic, this project's current scope is limited. As a result, we rely on DevTunnels for a simple, local testing environment that can still interact with Azure's cloud-based resources. Future teams may choose to deploy the application onto Azure's production-grade hosting services to fully support the Los Angeles Fire Department's call center needs at scale.

Why a Web Server is Required

Event Handling: The application relies on Azure Event Grid subscriptions and other Azure services that send events to the app's endpoint. A web server is needed to accept these incoming requests.

Two-Way Communication: Azure Call Automation and other APIs respond to events and prompts sent by the web server. Without a web server, these cloud services have nowhere to deliver their responses.

Scalability Options: While a simple DevTunnels server suffices for development use, full-scale deployment would typically use an Azure App Service or similar resource to handle high volumes of concurrent requests, improve security, logging, and scaling.

What Are DevTunnels?

DevTunnels allow you to quickly expose your local development environment to the internet. This is useful when testing webhooks, callbacks from cloud services like Azure Event Grid, or any external integrations that need a publicly accessible endpoint.

Installing DevTools (Including DevTunnels)

1. *Prerequisites: Ensure you have the .NET SDK or the Azure Developer CLI installed. DevTunnels is commonly part of these development kits.*

2. *Installing DevTools: Since this project was completed on MacOS, the instructions to install DevTools may be different for Windows systems. Please follow the instructions given here:* Create and host a dev tunnel
   All command line entries should be entered into a terminal in the same directory as main.py.

3. *Commands to Start the Web App Locally with DevTunnels*

    1. *Create a New DevTunnel: Enter this into the terminal CLI:*

        devtunnel create --allow-anonymous

        1. *What it Does: This creates a secure, public URL that maps to your local environment, allowing external services (like Azure Event Grid) to reach your machine.*

        2. *Why: Azure Event Grid and Call Automation need a publicly reachable endpoint to deliver events. The --allow-anonymous flag lets anyone with the URL hit your endpoint, suitable for testing. For production, secure authentication methods are recommended.*

    2. *Assign Port for Application: Enter this into the terminal CLI:*

        devtunnel port create -p 8080

        1. *What it Does: Assigns a public tunnel port for your application's local port 8080 (or whichever port your app runs on).*

        2. *Why: Your Flask or web application is likely running on localhost:8080. This maps that local port to a public URL, making it accessible over the internet.*

    3. *Host the DevTunnel:: Enter this into the terminal CLI:*

        devtunnel host

        1. *What it Does: Starts hosting the tunnel you just created, effectively bringing your local web server online and accessible via the provided public URL.*

        2. *Why: Without this command, the tunnel doesn't actively forward requests from the internet to your local machine.*

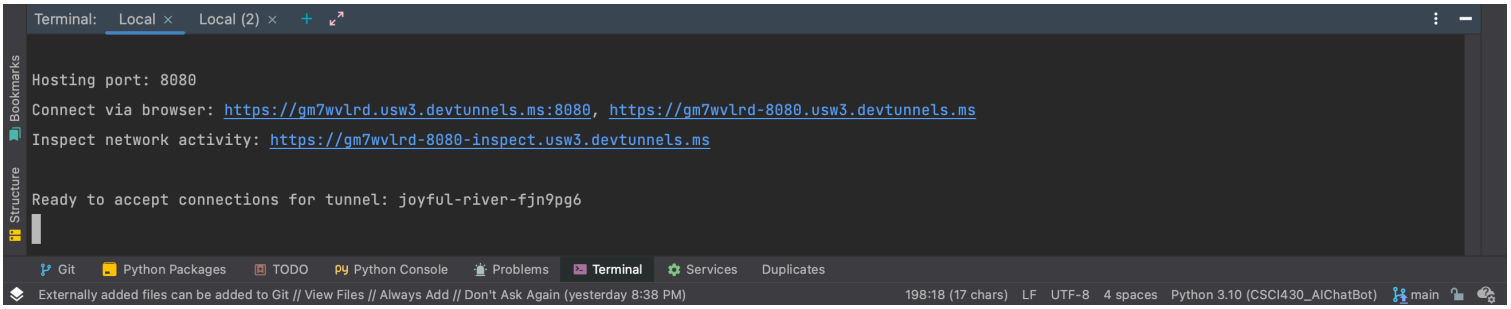
Starting the Web App via CLI

IMPORTANT: Do not start up the application until ALL setup is completed. Continue through the documentation and return to this section when setup is complete.

Once you have your tunnel set up and hosted, open another terminal window and start your app by entering the following command into the CLI:

python3 main.py

CRITICAL: *Now that the webserver is up and running, the URL of the web server can now be retrieved. After entering the above commands in the terminal, you well see a confirmation message. The URL can be found here (picture shown below). This URL is to be added into the CALLBACK_URI_HOST variable in* main.py *at the correct place, at or around line 198. ADDITIONALLY, this URL is to be configured as your Event Grid Subscription endpoint (refer to above documentation). The provided URL is provisioned for 30 days, and needs to be reset after this time frame. After 30 days, the previous URL will no longer be valid, and all credentials end endpoints concerning the URL need to be changed to the new URL after restarting the DevTools server.*



DevTools

In the above picture, the URL can be found by taking the tunnel name, in this case "joyful-river-rjn9pg6" and formatting it the following way:

"https://<TUNNEL NAME>.usw3.devtunnels.ms:8080"

*(URL for above example would be "https://joyful-river-fjn9pg6.usw3.devtunnels.ms:8080", and the endpoint for the Event Grid Subscription would be "*https://joyful-river-fjn9pg6.usw3.devtunnels.ms:8080/inboundCall*")*

Future Considerations

In a production environment, you wouldn't rely on DevTunnels. Instead, you might deploy the application to Azure App Service or Azure Functions, both of which natively provide robust, scalable web hosting. This would remove the need for a local tunnel and ensure that the LAFD Help Desk call center could operate at scale. For now, DevTunnels serves as a flexible, lightweight solution for development, testing, and small scale application deployment.

LAFD Help Desk AI ChatBot

FALL 2024

II. PYTHON SPECIFIC INSTRUCTIONS

This project, developed during the Fall 2024 semester of the USC CSCI401 Capstone course, focuses on creating an AI-powered chatbot solution for the Los Angeles Fire Department (LAFD) Help Desk. Traditionally, incoming service calls have been handled either by on-duty agents or through a basic web-based form, resulting in bottlenecks and increased pressure during high-volume periods. To address these challenges, this application leverages advanced, cloud-based technologies—primarily within the Microsoft Azure ecosystem—to provide a more efficient and scalable approach to managing technical service requests.

I. PYTHON SETUP

Python Requirements and Setup

This application's source code is hosted in a GitHub repository. To run the app locally, users must clone the repository, ensure a proper Python environment is set up, and install all required dependencies.

Prerequisites:

Python 3.10+: Ensure you have Python 3.10 or a newer stable release installed. You can verify this by entering this on CLI : python3 --version

Integrated Development Environment (IDE):

Any IDE or text editor that supports Python will work (e.g., Visual Studio Code, PyCharm, or even a terminal-based environment).

Cloning the Repository:

Navigate to the Desired Directory: Open a terminal or command prompt and navigate to the directory where you want to store the project:

Clone the Repository: Enter the following into the terminal CLI: (Replace <REPOSITORY_URL> with the actual GitHub repository URL provided for this project.)

```
git clone <REPOSITORY_URL>
```

Open the Project in Your IDE and setup environment:

Open the newly cloned repository in your IDE of choice. It's best practice to use a virtual environment to avoid dependency conflicts with other projects on your machine.

Create a Virtual Environment:

From within the repository directory, enter this into the CLI:

```
python3 -m venv venv
```

followed by

```
venv\Scripts\activate
```

This creates and activates `venv` directory containing a clean environment. Once activated, you should see `(venv)` before your terminal prompt, indicating that all subsequent Python commands will use this isolated environment.

Install the Requirements:

The project includes a requirements.txt file that lists all necessary Python packages. To install them, enter this into the CLI:
pip install -r requirements.txt

This command ensures all dependencies are properly installed in your virtual environment. Common libraries might include Flask, Azure SDK libraries, and other tools the app relies on.

Running the Application:

With dependencies installed, you can run the main Python script. The exact command may vary, but commonly via CLI: python main.py

Testing the Setup:

Local Access: Click the link that appears in the terminal, and the template front-end page for the web-app should appear (this page is used to test outbound calling, which is outside of the scope of this application. DO NOT USE IT)

DevTunnels/External Services: If using DevTunnels or other external tools, ensure those are running as described in the previous documentation sections. Confirm that Azure services can connect to your locally running code through the public tunnel URL.

Troubleshooting:

If dependencies fail to install, double-check that you're using Python 3.10+ and that you're in the correct directory and virtual environment.

If the application won't start, ensure the code is up to date with the repository's latest version and that all environment variables or configuration files are properly set.

By following these steps—cloning the repository, setting up a virtual environment, installing dependencies from requirements.txt, and running the main application—you establish a functional Python environment for local development and testing. This approach provides a clean, controlled environment that can be easily replicated and shared with other team members or future developers.

II. EMPLOYEE CSV FILE

Employee Information Data Formatting

A critical requirement for this application is the proper formatting and placement of the employee information data file. The application expects a CSV file named *Employee_Information.csv* located in a folder named EmployeeInfo at the root of the project directory. The file's structure and formatting are non-negotiable; any deviation will result in the application failing to process employee records correctly.

Required Format:

Filename: Employee_Information.csv

Directory: EmployeeInfo folder, which should be placed alongside the project files.

CSV Header and Example Record:

```
ID,FirstName,LastName,DisplayName,TelephoneNumber,EmailAddress
e693344,Brian,Barragan,Brian Barragan,,brian.Barragan@fire.lacounty.gov
[Rest of employees]
```

NOTE: A correctly formatted file of employee information is included in this project already. These instructions need to be followed if the file needs to be changed.

Formatting Rules:

Identical Headers: The column headers must appear exactly as shown. No additional columns, renamed headers, or altered casing is allowed.

Consistent Data Fields: Every row must follow the exact sequence and data type as the example. If a field is empty, it must still be represented by consecutive commas to maintain positional integrity.

No Alterations: Do not remove columns, rearrange their order, or introduce new data fields. The application's logic depends on this strict format for accurately parsing and mapping employee information.

User Responsibility: It is entirely up to the end-user or the team deploying this application to ensure that `Employee_Information.csv` is correctly formatted. If users find it challenging to conform to these requirements, it may be advisable for future development teams to implement a data-cleaning or validation script. For now, if the file is not named exactly `Employee_Information.csv` and placed in the `EmployeeInfo` directory, or if the data inside is not formatted as instructed, the application will not function as intended.

LAFD Help Desk AI ChatBot

FALL 2024

III. CONTINUED DEVELOPMENT

As the current solution serves as a proof-of-concept and partial implementation of the LAFD Help Desk AI chatbot, there are several key areas where future development efforts can significantly enhance functionality, scalability, and user experience. The following subsections outline recommended improvements and expansions.

1. DATA CLEANING AND VALIDATION FOR EMPLOYEE INFORMATION

Objective: Ensure that the Employee_Information.csv file is always correctly formatted, reducing the likelihood of application errors.

Details: Implementing a data cleaning script or functions would automatically validate CSV structure, field presence, and data types. Such scripts could provide diagnostic outputs or corrections for formatting issues, making it easier for non-technical staff to maintain accurate employee records.

Impact: This enhancement would minimize manual preparation, streamline the data intake process, and improve reliability when new employee data sets are introduced.

2. DIRECT ROUTING FOR TELEPHONE CONFIGURATION

Objective: Integrate direct routing capabilities with the organization's telephony systems.

Details: Currently, the project conceptually handles inbound and outbound call logic through Azure services. However, integrating with on-premises telephony infrastructure or a direct routing solution may require in-person configuration and coordination with the LAFD's IT and networking teams. This step ensures the AI chatbot is seamlessly integrated into the department's call center flow.

Impact: Achieving direct routing integration would allow the chatbot to handle live calls more efficiently and reliably, further reducing agent workload and improving call response times.

3. FRONT-END INTERFACE AND FULL-SCALE WEB SERVER

Objective: Provide a user-friendly front-end for administrators and possibly callers, as well as a production-grade web server.

Details: While the current system relies on DevTunnels for testing and a lightweight backend, future teams can build a dedicated web interface for viewing logs, configuring system settings, and monitoring call activity. Moving from a simple development server to an Azure App Service or another robust hosting solution ensures scalability, better security, and more reliable uptime.

Impact: A polished front-end and production-level hosting environment would make the system more approachable for non-technical stakeholders and support high-volume, real-world deployments.

4. LOGGING AND CALL RECORDING

Objective: Implement persistent logging and optional call recording for quality assurance, debugging, and audit purposes.

Details: Future enhancements might store detailed logs of interactions, including timestamps, recognized phrases, and system responses. Additionally, integrating call recording features—subject to LAFD policies and regulations—would allow administrators to review past calls for training, compliance, and improvement of the AI's responses.

Impact: Comprehensive logging and recording improve transparency, help diagnose issues, and provide a valuable resource for continuous refinement of the chatbot's conversational models and handling logic.

5. MULTILINGUAL CAPABILITIES

Objective: Expand the system to support multiple languages, making the help desk accessible to a broader range of employees.

Details: Leveraging Azure's Language and AI services, the system could automatically detect a caller's preferred language or allow the caller to select one. Future enhancements include training language models for different languages, integrating text-to-speech and speech-to-text services for those languages, and adapting menus and prompts accordingly.

Impact: Multilingual support greatly increases the utility of the system, ensuring inclusivity and improved user satisfaction across a diverse workforce.

6. DIRECT ROUTING TO RADIO-CONTROLLED UNIT

Another key area for future enhancement is the integration of direct routing to the Radio-Controlled Unit (RCU) of the LAFD Help Desk. While the current setup can transfer callers between Azure-supported telephone endpoints, extending this capability to seamlessly connect external callers to the radio-controlled unit's existing telephony system requires additional infrastructure work.

Why This Feature Remains Incomplete:
Currently, Azure supports call transfers between numbers and services within the Azure communication environment. However, the Help Desk's radio-controlled unit is tied into the LAFD's on-premises or existing telephony infrastructure. To enable a direct transfer

from an external caller (a non-Azure phone) to the RCU (also a non-Azure number), Azure direct routing configurations must be established. This involves integrating the LAFD's numbers and telephony systems directly with Azure's telephony services so that call routing flows seamlessly between them.

What Future Teams Must Do:

Establish Direct Routing: Set up Azure direct routing capabilities to link the LAFD's telephone infrastructure with Azure Communication Services.

Coordinate with LAFD IT and Network Teams: Achieving direct routing likely requires on-site configuration, discussions with the LAFD's telecommunications vendors, and ensuring compliance with organizational policies and security measures.

Update Code to Leverage New Routing: Once direct routing is in place, the code can be adjusted to treat the RCU as a recognized, routable endpoint, allowing external callers to be transferred just as easily as they are currently transferred between Azure-controlled numbers.

This feature—enabling seamless transfers to the radio-controlled unit—cannot be fully implemented until the underlying telephony integration is complete. Subsequent groups working on this project will need to collaborate with the LAFD's IT department and follow Azure direct routing best practices to achieve this goal. Once in place, this enhancement would further streamline call handling and bring the solution closer to full operational readiness within the LAFD Help Desk environment.

IMPORTANT: Currently, if the caller chooses the connect to a radio controlled unit from the main menu selections, a prompt is played that explains the ongoing development of this feature.

7. RECOMMENDATION FOR FULL INTEGRATION TESTING

While basic error checking and handling have been included in this prototype, ensuring the application can handle all possible conversation branches and scenarios will require a

thorough, systematic testing approach. This includes testing speech recognition accuracy, event handling reliability, network resilience, and overall logic integrity. A robust testing strategy will help identify logic flaws, unforeseen input combinations, and error conditions that may only arise during real-world usage.

Why Integration Testing is Crucial:

Complex Interactions: The application depends on multiple Azure services (Call Automation, Event Grid, Communication Services), a local web server, and external data sources. Testing each piece in isolation isn't enough—verifying they work together smoothly is essential.

Conversation Branches: The call logic includes numerous branching points based on user input. Exhaustive tests ensure the code handles unexpected phrases, invalid DTMF entries, or incomplete information without crashing or producing incorrect results.

Scalability and Reliability: Before future teams deploy the system at scale, rigorous integration tests will help validate that the solution can handle higher volumes of calls and more complex call flows.

Recommended Testing Tools and Approaches:

Pytest for Unit and Integration Tests:

Pytest is a popular Python testing framework known for its simplicity, readability, and flexibility. Using Pytest, you can write integration tests that:

Simulate incoming Azure Event Grid events.

Mock Azure Call Automation responses.

Validate that ticket creation and email sending logic triggers correctly. Its extensive plugin ecosystem and fixtures allow you to create realistic test

environments, inject mock data, and simulate different call states and user inputs.

Mocking and Stubbing External Services:
Use libraries like unittest.mock or Pytest's built-in mocking capabilities to simulate Azure service responses, network timeouts, or invalid data. This approach ensures that tests are not dependent on live endpoints and can be run repeatedly and consistently during development or CI/CD pipelines.

Behavior-Driven Development (BDD) Tools:
For more conversational or scenario-based testing, consider tools like Behave. BDD allows you to write tests in plain English, mapping conversation steps (e.g., "User says 'File ticket'" or "User presses 1") to code that verifies the system's responses. This can be particularly helpful for ensuring conversational logic covers all expected paths.

Continuous Integration (CI) and Continuous Deployment (CD):
Integrating these tests into an Azure DevOps pipeline or GitHub Actions workflow ensures that every change to the codebase triggers a full integration test suite. Automatically running tests on every commit or pull request reduces the risk of introducing new bugs and ensures the code remains stable over time.

Performance and Load Testing:
While not strictly integration testing, using tools such as Locust for load testing can help identify bottlenecks or performance issues. By simulating multiple simultaneous calls, you can verify that logic, event handling, and email sending remain reliable under stress.

The application depends on multiple Azure services (Call Automation, Event Grid, Communication Services), a local web server, and external data sources. Testing each piece in isolation isn't enough—verifying they work together smoothly is essential.

For a solution as multifaceted as this AI-driven call handling application, robust integration testing is a must. Pytest, combined with mocking frameworks and, if desired, a BDD tool like Behave, can cover a wide range of test scenarios. Future development teams should establish a comprehensive test suite to continuously validate every aspect of the system—conversation logic, external service interactions, error handling—ensuring a stable, reliable help desk solution for the LAFD.

In Summary:
These recommended development areas offer clear pathways to making the AI chatbot more robust, user-friendly, integrated, and feature-rich. Subsequent student groups will find this documentation helpful as a springboard to completing a full-production scale application. By addressing data quality, telephony integration, front-end usability, comprehensive logging, and multilingual support, future teams can transform this proof-of-concept prototype into a fully realized solution that serves the LAFD Help Desk's evolving needs.

LAFD Help Desk AI ChatBot

FALL 2024

IV. CODE EXPLANATION AND CALL FLOW LOGIC

This section provides a high-level walkthrough of the application's code structure and logic flow, illustrating how a call moves from initial connection to the creation and emailing of a service ticket. The goal is to give a reader—particularly developers and future maintainers—a clear understanding of how the code is organized and how the logic progresses from start to finish.

1. CODE STRUCTURE OVERVIEW

Imports and Dependencies:

At the top of the code, you will find imports for Python standard libraries (e.g., csv, time) as well as Azure SDK modules and Flask. These libraries power the application's ability to handle HTTP requests, connect to Azure services, process speech, and handle telephony events.

Class Definitions (Employee, Ticket, RetryObject):

Employee: Represents staff members of the LAFD, including basic identifying information and contact details.

Ticket: Encapsulates all the data needed for a help desk ticket (e.g., employee name, ID, contact info, work location, urgency, and issue description).

RetryObject: Tracks attempts to get valid input from the caller, such as speech or DTMF responses, allowing the logic to prompt the user multiple times if needed.

Employee Data Loading:

The application loads Employee_Information.csv from the EmployeeInfo folder into a dictionary. Each row in the CSV corresponds to an Employee object. This ensures that when a caller provides an employee ID, the system can look up their details automatically.

Global Variables and Configuration:

Several global variables store runtime state, such as current_employee, current_ticket, and retry_object. Additionally, constants and connection strings for Azure services (Communication Services, Event Grid, etc.) are defined. These values must be set correctly before starting the application.

Helper Functions for Menu Choices and Interactions:

Functions like get_menu_choices(), get_confirm_choices(), and others define the sets of recognition choices (speech phrases and DTMF options) the caller can select from. These helper functions standardize the logic for prompting users, confirming their responses, and branching the conversation flow.

Azure Call Automation and Event Grid Integration:

The code uses CallAutomationClient to manage phone calls. Azure Event Grid events are sent to the application's callback endpoints whenever something significant happens during a call (e.g., the call connects, speech is recognized, DTMF is entered).

Flask Routes:

@app.route('/inboundCall', methods=['POST']): Handles incoming calls and Event Grid subscription validation. When a new call arrives, the code answers it using Azure Call Automation.

@app.route('/api/callbacks', methods=['POST']): Receives ongoing event notifications from Event Grid during the call lifecycle. Each event triggers logic within the application to guide the caller through the next steps.

2. CALL LOGIC FLOW

Inbound Call and Initialization:
When a call is first placed to the system's phone number, Azure sends an IncomingCall event to /inboundCall. The code answers the call via call_automation_client.answer_call() and logs that a new call connection is established. At this point, a fresh current_ticket is created, ensuring each call session starts with a blank ticket template.

Main Menu Prompt:
After connecting the call, the callback_events_handler receives a CallConnected event. In response, the code initiates media recognition with get_menu_choices(). The caller hears a greeting and is asked whether they'd like to file a ticket or speak to a radio-controlled unit. The caller can respond via speech or by pressing a key on their phone keypad.

Gathering Employee and Issue Details:
If the caller chooses to file a ticket, the logic flow guides them through a series of prompts:

> **Employee ID:** The system asks for and confirms the caller's employee ID. Once verified, the code retrieves the Employee details from the loaded CSV data.
>
> **Contact Information:** The caller is prompted to confirm or provide their phone number and email address.
>
> **Work Mode and Address:** The caller states whether they are in the office or teleworking, and provides the physical address where the issue is occurring.
>
> **Urgency and Issue Description:** Finally, the caller classifies the urgency of their problem (low, medium, high) and describes the issue in their own words.

Throughout this process, RecognizeCompleted events trigger transitions between prompts. If the caller's input is unclear, RetryObject logic may prompt them again.

Confirming the Collected Information:
After all necessary information is collected, the current_ticket now holds all data required

to create a service ticket. The code can easily reference the employee's name, contact method, issue description, and other details stored in the Ticket object.

Sending the Ticket to Microsoft Footprints via Email:
Once the caller confirms that they are done and wants to create the ticket, the code calls send_email() with the completed Ticket object. This function uses Azure Communication Services EmailClient to send a formatted email to a designated Microsoft Footprints email address. Microsoft Footprints, the ticketing system, receives this email and automatically generates a ticket on the help desk's internal platform. The email includes all the details the caller provided, ensuring the request is logged, tracked, and can be acted upon by the help desk team.

Call Termination:
After sending the ticket, the system thanks the caller and ends the call. The hang_up() method is eventually called on the call_connection_client, and the session completes. Logs and callbacks can be reviewed for troubleshooting and auditing.

4. SUMMARY OF CALL LIFECYCLE

Call Arrives → Answered by Azure Call Automation.

Main Menu Prompt → Caller chooses ticket or MCU transfer.

Ticket Filing → System prompts for employee ID, phone/email, work location, urgency, and issue details.

Data Collection Completed → send_email() sends a formatted ticket to Footprints.

Call Ends → User hears a goodbye message and the call is terminated.

By following this flow, the code ensures a structured, step-by-step conversation with the caller, leveraging Azure's communication, speech, and event services to deliver a streamlined and efficient help desk experience.

Full Python documentation is available in the project directory under the name **main.html** in the **html** directory. This can be opened in the browser by right clicking and navigating to the correct menu choice.

LAFD Help Desk AI ChatBot

FALL 2024

ADDITIONAL HELP

If you encounter difficulties or need more information beyond this documentation, there are many resources available to help you configure, maintain, and extend this application's capabilities. Whether you're troubleshooting issues, exploring advanced features, or planning major enhancements, the following websites and guides can assist in deepening your understanding of Microsoft Azure services and related technologies:

1. MICROSOFT AZURE DOCUMENTATION:

The official Azure documentation provides comprehensive guides, tutorials, and reference material for all Azure services.

https://docs.microsoft.com/azure

2. AZURE COMMUNICATION SERVICES:

For voice, video, and telephony integrations, start with the Azure Communication Services documentation. It covers setup, integration, and best practices for Call Automation and related features.

https://docs.microsoft.com/azure/communication-services/

Call Automation Specific Docs:

https://docs.microsoft.com/azure/communication-services/concepts/voice-video-calling/call-automation

3. AZURE EVENT GRID:

If you need to handle more complex event routing, monitoring, or scaling scenarios, the Event Grid documentation will guide you through setting up event subscriptions, handling custom events, and integrating with other Azure services.

https://docs.microsoft.com/azure/event-grid/

4. AZURE LANGUAGE AND AI SERVICES:

For implementing speech-to-text, text-to-speech, or multilingual capabilities, Azure Cognitive Services and Language resources are your go-to references.

https://docs.microsoft.com/azure/cognitive-services/

https://docs.microsoft.com/azure/cognitive-services/language-service/

5. AZURE APP SERVICE AND HOSTING OPTIONS:

If future teams decide to deploy this application to production, the Azure App Service documentation can guide you through setup, scaling, security, and cost management.

https://docs.microsoft.com/azure/app-service/

6. COMMUNITY FORUMS AND SUPPORT:

If official documentation doesn't resolve your issue, consider reaching out to the community or Microsoft support:

Stack Overflow: Use the azure tag to get community help.
https://stackoverflow.com/questions/tagged/azure

Microsoft Q&A: Official forum for Azure-related questions.
https://docs.microsoft.com/answers/topics/azure.html

Azure Support Plans: For paid tiers of assistance.
https://azure.microsoft.com/support/

By leveraging these resources, you can troubleshoot challenges, stay updated with the latest features, and continually improve and extend the solution for the Los Angeles Fire Department's evolving help desk needs.