# **AI-Powered Airline Call Center Optimization**

## **Overview**

In an airline call center, agents handle a wide range of customer inquiries—from flight cancellations and reschedules to refunds and complaints. These conversations produce valuable data that can be leveraged through Generative AI. By integrating services like AWS Bedrock or OpenAI or Together AI models, the system can:

* **Transcribe** audio recordings
* **Categorize** issues (e.g., Flight Cancellation vs. Refund)
* **Compute KPIs** to optimize call center operations

This approach should enhance customer satisfaction, reduce call handling times, and streamline agent performance.

## **Problem 1: Two-Agent System with Function Calling & Structured Output**

### **Evaluation Criteria:**

* **Multi-Agent Coordination** – Handling two AI agents working together
* **Function Calling** – One agent invoking another’s functions
* **Structured JSON Output** – Ensuring responses follow a defined schema
* **Prompt Engineering** – Effectively guiding AI agents

### **Problem Statement**

You are building a system where two AI agents collaborate to answer user queries about airline flights:

#### **1. Info Agent**

* Has access to a function get\_flight\_info(flight\_number) that returns structured flight data (e.g., destination, departure time, status).
* Responds **only in JSON format**, with no additional text.

#### **2. QA Agent**

* Receives user queries (e.g., *“What time does Flight 123 depart?”*).
* Calls the Info Agent to fetch relevant flight data.
* Processes the result and returns a structured JSON response in a user-friendly format.

### **Functions to Implement**

#### **1. get\_flight\_info(flight\_number: str) -> dict**

* Simulates or mocks flight data retrieval.

Returns a Python dictionary with keys like:  
  
{

"flight\_number": "AI123",

"departure\_time": "08:00 AM",

"destination": "Delhi",

"status": "Delayed"

}

#### **2. info\_agent\_request(flight\_number: str) -> str**

* Calls get\_flight\_info and returns the data **as a JSON string**.
* **No extra text**—only valid JSON.

#### **3. qa\_agent\_respond(user\_query: str) -> str**

* Extracts the flight number from the query (e.g., *“Flight 123”*).
* Calls info\_agent\_request to get the flight’s JSON data.

Returns a structured JSON response, for example:  
  
{

"answer": "Flight AI123 departs at 08:00 AM to Delhi. Current status: Delayed."

}

* The output **must strictly follow JSON format**—no plain text or extra commentary.

### **Test Cases:**

|  |  |
| --- | --- |
| **Function Call** | **Expected Output** |
| get\_flight\_info("AI123") | {"flight\_number": "AI123", "departure\_time": "08:00 AM", "destination": "Delhi", "status": "Delayed"} |
| info\_agent\_request("AI123") | JSON string of the above dictionary (without extra text) |
| qa\_agent\_respond("When does Flight AI123 depart?") | {"answer": "Flight AI123 departs at 08:00 AM to Delhi. Current status: Delayed."} |
| qa\_agent\_respond("What is the status of Flight AI999?") | {"answer": "Flight AI999 not found in database."} (if flight doesn’t exist) |

## **Problem 2: Binary Classification for Customer Sentiment**

### **Evaluation Criteria:**

* **Data Preprocessing** – Tokenization, handling missing values, etc.
* **Model Training** – Logistic Regression, Naive Bayes, or similar
* **Prediction & Evaluation** – Accuracy, confusion matrix, etc.
* **Use of Provided Training Data**

### **Problem Statement**

You have a small dataset of customer feedback from an airline. Each row contains a **text snippet** and a **binary label**: *positive* or *negative*.

#### **Your task:**

1. Train a model to classify feedback as **positive** or **negative**.
2. Predict the sentiment for new input text.

### **Sample Training Data**

|  |  |
| --- | --- |
| **Text** | **Label** |
| "The flight was on time, and the staff was friendly." | positive |
| "I had to wait 3 hours due to a delay. Terrible!" | negative |
| "Great legroom and comfortable seats." | positive |
| "Lost my luggage, extremely upset about this." | negative |
| "Check-in was smooth, no issues at all." | positive |

*(Full training dataset will be provided.)*

*Link to download:* [*2026\_ML\_test\_dataset*](https://neuralworksai-my.sharepoint.com/:f:/g/personal/akash_c_goml_io/EqWRzEwiRlxHjHkbZ0USytAB_SjQ2_8vv_56g330uxGmMA?e=slLujI)

### **Functions to Implement**

#### **1. train\_sentiment\_model(training\_data: List[Tuple[str, str]]) -> Any**

* Accepts a list of (text, label) pairs where label ∈ { "positive", "negative" }.
* Preprocesses the text (tokenization, lowercasing, etc.).
* Trains a simple model (e.g., LogisticRegression from sklearn).
* Returns the trained model object.

#### **2. predict\_sentiment(model: Any, new\_text: str) -> str**

* Accepts the trained model and a text string.
* Applies the same preprocessing used during training.
* Returns "positive" or "negative" based on the model’s prediction.

### **Test Cases**

|  |  |
| --- | --- |
| **Function Call** | **Expected Output** |
| train\_sentiment\_model([("I love this airline", "positive"), ("Worst experience ever", "negative")]) | Trained model object (e.g., LogisticRegression) |
| predict\_sentiment(model, "The seats were comfortable and service was great!") | Likely "positive" (depending on training) |
| predict\_sentiment(model, "They lost my baggage and were very unhelpful!") | Likely "negative" |
| predict\_sentiment(model, "Nothing special, just an average flight.") | Could be "positive" or "negative"—depends on the model. |

## **Implementation & Submission Guidelines**

### **Prerequisites**

Before you begin, ensure you have:

* **Python 3.8+** installed
* A virtual environment setup (venv or conda recommended)
* API keys for OpenAI, AWS Bedrock, or Together AI (depending on your choice of LLM provider)

Installed required libraries:  
  
 pip install openai boto3 together ai

## **LLM Setup and API Key Configuration**

### **1. Using OpenAI**

#### **Getting API Key**

1. Sign up or log in at [OpenAI](https://platform.openai.com/signup/).
2. Navigate to **API Keys** under your **Account Settings**.
3. Generate a new API key and copy it.

#### **Setting Up API Key in Python**

import openai

import os

os.environ["OPENAI\_API\_KEY"] = "your\_openai\_api\_key"

response = openai.ChatCompletion.create(

model="gpt-4",

messages=[{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": "What is AI?"}]

)

print(response["choices"][0]["message"]["content"])

### **2. Using AWS Bedrock**

#### **Getting API Key**

1. Sign in to [AWS Console](https://aws.amazon.com/console/).
2. Navigate to **AWS IAM** → Create a user with Bedrock access.
3. Attach the **AmazonBedrockFullAccess** policy.
4. Generate and download the **Access Key ID** and **Secret Access Key**.

#### **Setting Up API Key in Python**

```

import boto3

bedrock = boto3.client(

service\_name="bedrock-runtime",

region\_name="us-east-1", # Change region if needed

aws\_access\_key\_id="your\_aws\_access\_key",

aws\_secret\_access\_key="your\_aws\_secret\_key"

)

response = bedrock.invoke\_model(

body='{"prompt": "What is AI?", "max\_tokens": 100}',

modelId="anthropic.claude-v2"

)

print(response["body"].read().decode("utf-8"))

```

### **3. Using Together AI**

#### **Getting API Key**

1. Sign up at [Together AI](https://together.ai/).
2. Go to **API Keys** and generate a new key.
3. Copy and store it securely.

#### **Setting Up API Key in Python**

import together

import os

os.environ["TOGETHER\_API\_KEY"] = "your\_together\_api\_key"

response = together.ChatCompletion.create(

model="together/gpt-neoxt-20b",

messages=[{"role": "user", "content": "What is AI?"}]

)

print(response["choices"][0]["message"]["content"])

### **2. Language & Libraries**

* Use **Python**.
* For **Problem 1**, simulate/mock multi-agent interaction.
* For **Problem 2**, libraries like sklearn, numpy, pandas are permitted.

### **3. Project Structure**

**Suggested file structure:**  
  
├── agent\_system.py # For Problem 1

├── ml\_classifier.py # For Problem 2

├── README.md # Setup & usage instructions

### **4. Testing**

* Ensure functions pass **all test cases**.
* Handle edge cases (e.g., missing flight number, empty text).

### **5. Documentation**

* **Docstrings** for every function.
* **README** should explain:  
    
  + Installation
  + Running the code
  + Approach for multi-agent function calling (optional).

### **6. Time Constraints**

* **Recommended:** 60 minutes (or as allocated).
* Extra time? Consider:  
    
  + Robust error handling
  + Hyperparameter tuning

### **7. Submission Instructions**

1. **Project Structure**  
     
   1. Keep each problem in a separate folder.
   2. Each folder must contain a README.md with instructions to run the code.
   3. Include all necessary files but **DO NOT** include \_\_pycache\_\_, venv, or unnecessary logs.
2. **API Keys**  
     
   1. Store API keys in an api\_keys.env file inside each folder.
   2. **Do not** hardcode API keys in the scripts.
3. **Dependencies**  
     
   1. Each folder must have a requirements.txt specifying the necessary packages.
4. **Packaging for Submission**  
     
   1. Zip the project folder while excluding virtual environments and cache files.
   2. The final ZIP file should contain all folders with their respective README.md, requirements.txt, and scripts.
   3. Add the Zip file to your drive and give access to whoever can access the link.
   4. Submit the link to google form given
   5. Google Form for Submission: https://docs.google.com/forms/d/e/1FAIpQLSc8E8Sh32CeKFDrN82DouvMh1DLimzWgiTW\_VtmJAmlziophw/viewform?usp=header

**Example Structure:**

submission.zip/

│── problem1/

│ ├── main.py

│ ├── api\_keys.env

│ ├── requirements.txt

│ ├── README.md

│── problem2/

│ ├── main.py

│ ├── api\_keys.env

│ ├── requirements.txt

│ ├── README.md

All the very best