AI-Powered Question Answering Chatbot using Microsoft Azure Language Studio

AI-Powered Conversational Assistant using Azure Cognitive Services for Language Understanding

Project Overview

This project demonstrates the development of an intelligent conversational agent using Microsoft Azure Cognitive Services for Language (Custom Question Answering). The solution is designed to understand natural language queries, provide accurate predefined responses, and seamlessly integrate into business communication channels. The project highlights how cloud-based NLP (Natural Language Processing) can automate customer interactions, reduce manual support overhead, and improve user experience across various industries.

Business Problem

Businesses across retail, healthcare, and travel face high volumes of repetitive customer inquiries, leading to long response times, increased operational costs, and inconsistent service quality. A custom-built Language Understanding bot can autonomously handle frequently asked questions, provide instant responses, and escalate complex issues to human agents—enabling 24/7 support while maintaining high accuracy and personalization.

Project Objectives

The objectives of this project were to:

- Select a relevant business domain (Retail, Healthcare, or Travel) for the chatbot.
- Define and structure a training dataset with at least five unique queries and their variations.
- Develop and train a custom question-answering model using Azure Language Studio.
- Outline a rigorous testing strategy to evaluate model accuracy and intent recognition.
- Deploy the trained chatbot to a live Azure platform (Web App, Teams, or Telegram).
- Demonstrate end-to-end functionality through a recorded demo video.

Solution Workflow

Phase 1: Domain Selection & Dataset Curation (Credit Task)

- Selected a use case (e.g., Retail Chatbot – product inquiries, Healthcare Assistant – symptom/visit FAQs, or Travel Booking – itinerary/recommendations).
- Defined a structured training dataset containing at least five core user questions.
- Expanded the dataset with multiple linguistic variations (paraphrases, typos, and rephrased questions) to improve model robustness.
- Created a mapping of expected responses for each intent to ensure accurate knowledge retrieval.

Phase 2: Model Training & Testing (Credit Task)

- Used Azure Cognitive Services Language Studio to import the dataset and train the custom question-answering model.
- Designed a testing approach to evaluate performance (precision, recall, and confidence thresholds).

CHIRAG ARUNKUMAR VYAS

Phase 3: Deployment & Integration (Distinction Task)

- Deployed the trained model to Azure Web App for public accessibility.
- Integrated the bot with a communication platform (e.g., Microsoft Teams, Telegram, Slack, or an embedded website widget).
- Configured response variations to make interactions more dynamic and human-like.
- Conducted live testing to validate deployment stability and response latency.

Model Testing & Evaluation

The system was evaluated using the following approach:

- Accuracy Metrics: Tested the model against unseen queries to measure Exact Match and Semantic Similarity scores.
- Confidence Thresholding: Analyzed the model's confidence scores to determine optimal thresholds for accepting vs. handing off queries.
- Out-of-Scope Handling: Evaluated the bot's ability to gracefully handle irrelevant or unsupported questions.

Sample Test Cases:

Test Case	User Query	Expe
Standard	"What are your store hours?"	Retur
Paraphrased	"When do you guys open?"	Corre
Out-of-Scope	"What is the weather today?"	Grace queri

Key Findings

Strengths

- Azure Language Studio provides an intuitive, no-code interface for rapid prototyping.
- The model handles linguistic variations effectively when trained with diverse paraphrases.
- Deployment to Web Apps and Teams is seamless with built-in Azure integrations.
- Response confidence scoring offers transparency and control over automated replies.

Limitations

- Model performance is highly dependent on the quality and size of the training dataset.
- Handling complex multi-turn conversations requires additional orchestration (e.g., Power Virtual Agents).
- Without active learning, the model may degrade over time as user phrasing evolves.
- Latency can increase slightly when deployed over cloud channels compared to local inference.

Potential Improvements

Future enhancements include:

- Implementing Active Learning to continuously retrain the model on real user queries flagged as low-confidence.
- Adding Multi-Language Support using Azure Translator to cater to a global audience.
- Integrating Azure Bot Service and Power Automate to enable transactional actions (e.g., booking appointments or placing orders).
- Utilizing Semantic Ranking (Azure Cognitive Search) to fetch dynamic answers from large document repositories rather than static Q&A pairs.
- Enhancing the UI/UX with rich cards (buttons

Table of content

Sr no.	Content	Page no.
1	Step 1 Azure Language Services – Overview (task 6C)	1
2	Step 2 Libraries installation for azure language services	2
3	Step 3 Create a language resource in your resource group (Azure Portal) *A. Creating resource group as per following click on review and create* *B. Click on and create* *C. we can see resource group is created successfully* *D. Click on newly created resource group named as CAV-2-language* *E. After clicking on resource group click to create resources* *F. click to create* *G. Azure language services provide many default services like as follows* *H. Setup azure language services as follow premium LRS also free service* *I. Verify and Click on create review it once again* *J. Deployment is in progress* *K. takes few minutes to deployment* *L. Click on overview of resource group* *M. Click on require resources.* *N. verify and view keys and endpoints* *O. storing endpoint and key*	4
4	Step 4: importing secured credentials for initializing Azure QnA	11
5	Step 5: Language Studio setup A. Creating project on language studio B. Creating project on language studio (using python sdk) C. Add a knowledge base D deploying project	11
6	Step 6: Create bot: (task 6D)	16
7	Step 7: Test bot on Azure web chat	17
8	Step 8: Integrating it with telegram	18
9	step 9: Model testing and evaluation	19
10	step 10: Delete Resources	21

- **Step 1 Azure Language Services – Overview**

Azure offers powerful Language Services as part of Azure Cognitive Services and Developer Tools, serving two major areas:

🗣️ Natural Language Processing (NLP) Services

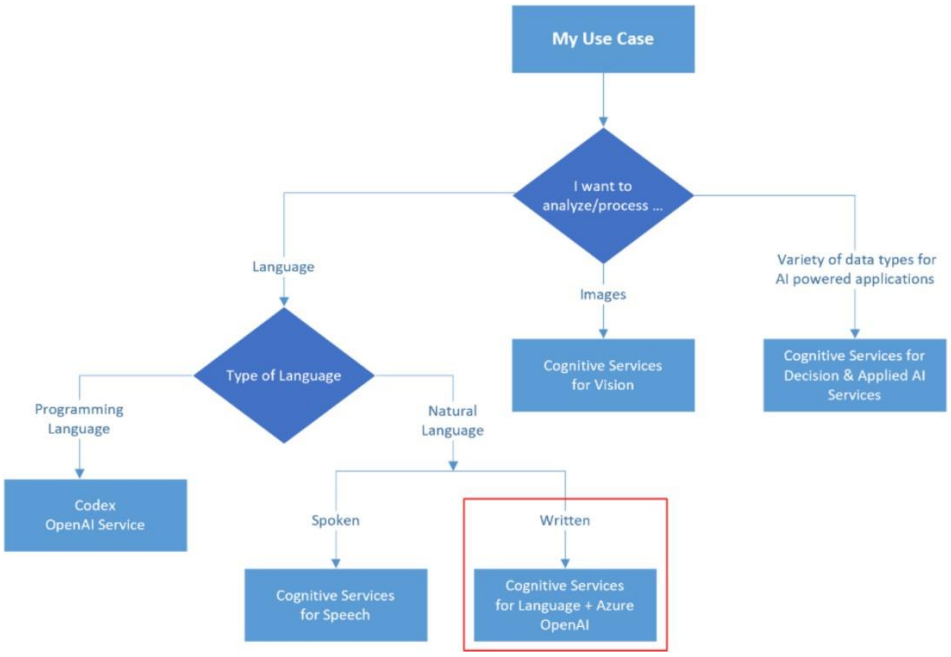
These services are designed to help applications understand and process human language.

1. **Conversational Language Understanding (CLU)**
   - Detects intents and entities from user utterances (e.g., chatbot inputs).
   - Ideal for building intelligent assistants and domain-specific NLP models.
2. **Text Analytics**
   - Performs sentiment analysis, key phrase extraction, language detection, and named entity recognition (NER).
3. **Question Answering**
   - Builds Q&A systems using knowledge bases (FAQs, documents).
4. **Text Translation**
   - Automatically translates text between languages using Microsoft Translator.
5. **Text Summarization**
   - Extracts summaries from larger documents, useful in news, legal, and business content.
6. **Language Detection**
   - Identifies the language of a given text input.
7. **Speech-to-Text and Text-to-Speech (integrated with Azure Speech Service)**
   - Converts spoken language to written text and vice versa.

💻 Programming Language–Related Services and Tools

While not part of "Azure Language Services" specifically, Azure provides tools and integrations to support code understanding, analysis, and development:

1. GitHub Copilot (Powered by Azure OpenAI Service)
   - AI pair programmer that helps write, complete, and understand code in editors like VS Code.
2. Azure OpenAI Service
   - Gives access to models like GPT-4, which can assist in code generation, debugging, documentation, and more.
   - Supports both natural language and programming language use cases.
3. Azure DevOps & Visual Studio Code Extensions
   - Tools for code collaboration, CI/CD, and AI-assisted development using extensions.



- **Step 2 Libraries installation for azure language services**

1. Installation of neccessary Library

```
[12]: # https://learn.microsoft.com/en-us/python/api/overview/azure/ai-language-questionanswering-readme?view=azure-python

[13]: # Install the azure question answering library
!pip install azure-ai-language-questionanswering

# doenv library
!pip install python-dotenv
```

```
Collecting azure-ai-language-questionanswering
  Downloading azure_ai_language_questionanswering-1.1.0-py3-none-any.whl.metadata (19 kB)
Requirement already satisfied: azure-core<2.0.0,>=1.24.0 in c:\users\chirag pc\anaconda3\lib\site-packages (from azure-ai-language-questionanswering) (1.
32.0)
Requirement already satisfied: isodate<1.0.0,>=0.6.1 in c:\users\chirag pc\anaconda3\lib\site-packages (from azure-ai-language-questionanswering) (0.7.2)
Requirement already satisfied: requests>=2.21.0 in c:\users\chirag pc\anaconda3\lib\site-packages (from azure-core<2.0.0,>=1.24.0->azure-ai-language-ques
tionanswering) (2.32.3)
Requirement already satisfied: six>=1.11.0 in c:\users\chirag pc\anaconda3\lib\site-packages (from azure-core<2.0.0,>=1.24.0->azure-ai-language-questiona
nswering) (1.16.0)
Requirement already satisfied: typing-extensions>=4.6.0 in c:\users\chirag pc\anaconda3\lib\site-packages (from azure-core<2.0.0,>=1.24.0->azure-ai-langu
age-questionanswering) (4.9.0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\chirag pc\anaconda3\lib\site-packages (from requests>=2.21.0->azure-core<2.0.0,>=1.2
4.0->azure-ai-language-questionanswering) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\chirag pc\anaconda3\lib\site-packages (from requests>=2.21.0->azure-core<2.0.0,>=1.24.0->azure-ai
-language-questionanswering) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\chirag pc\anaconda3\lib\site-packages (from requests>=2.21.0->azure-core<2.0.0,>=1.24.0->az
ure-ai-language-questionanswering) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\chirag pc\anaconda3\lib\site-packages (from requests>=2.21.0->azure-core<2.0.0,>=1.24.0->az
ure-ai-language-questionanswering) (2024.2.2)
Downloading azure_ai_language_questionanswering-1.1.0-py3-none-any.whl (113 kB)
   ---------------------------------------- 0.0/113.1 kB ? eta -:--:--
   --- ------------------------------------ 10.2/113.1 kB ? eta -:--:--
   ---------------------------------------- 113.1/113.1 kB 1.3 MB/s eta 0:00:00
Installing collected packages: azure-ai-language-questionanswering
Successfully installed azure-ai-language-questionanswering-1.1.0
Requirement already satisfied: python-dotenv in c:\users\chirag pc\anaconda3\lib\site-packages (0.21.0)
```

Installing Required Libraries for Azure Language Services (Python)

To start working with **Azure AI Language Services**, including **Question Answering**, you need to install a few key Python libraries.

1. Azure Question Answering Library

This library allows you to integrate Azure's **Question Answering** capability into your Python applications.

! pip install azure-ai-language-questionanswering

This provides access to prebuilt and custom question answering features from Azure's Language Service.

---

2. dotenv Library

The python-dotenv library helps you manage **environment variables** (like your Azure keys and endpoint) securely using a .env file.

! pip install python-dotenv

Ideal for keeping sensitive credentials out of your main codebase.

2. Import the required library

```
[ ]: # This section loads your Azure endpoint and subscription key from a .env file and prepares for secure API calls.

[14]: # Read secret keys and environment variables securely
import os
from dotenv import load_dotenv

# Load variables from .env file
load_dotenv()

# Azure authentication helper
from azure.core.credentials import AzureKeyCredential

[15]: # The AuthoringClient helps build and manage the Q&A project, while the QuestionAnsweringClient is used for querying that project.

[16]: # Authoring client - used to create and manage knowledge base projects
from azure.ai.language.questionanswering.authoring import AuthoringClient

# Runtime client - used to query your knowledge base at runtime
from azure.ai.language.questionanswering import QuestionAnsweringClient

# Import QnA models for structured interactions (e.g., QueryOptions, Answers)
from azure.ai.language.questionanswering import models as qna
```

1. Reading Secret Keys for Authentication

python

```
import os
from dotenv import load_dotenv
from azure.core.credentials import AzureKeyCredential
```

What It Does:

- os: Lets you interact with environment variables in your system.
- dotenv.load_dotenv(): Loads environment variables (like your Azure key and endpoint) from a .env file.
- AzureKeyCredential: A secure way to pass your Azure key when authenticating with Azure services.

This setup ensures you don't hard-code sensitive info like API keys directly in your script.

2. Creating the Authoring Client

python

from azure.ai.language.questionanswering.authoring import AuthoringClient

What It Does:

- The AuthoringClient allows you to:
  - Create and configure a **Q&A project**
  - Upload documents or URLs as knowledge sources
  - Train and manage your knowledge base

You use this **only during setup and configuration** of your Q&A system.

---

3. Creating the Runtime (Question Answering) Client

python

from azure.ai.language.questionanswering import QuestionAnsweringClient

What It Does:

- This client lets you **send questions** to your deployed knowledge base and **receive answers**.
- It's used when your app is running and interacting with users.

Think of this as the "live" Q&A interface that users talk to.

---

4. Importing Models (Optional but Useful)

python

from azure.ai.language.questionanswering import models as qna

What It Does:

- Gives you access to data models and classes like:
  - qna.AnswerResult
  - qna.QueryKnowledgeBaseOptions
- Useful for **advanced customization** of questions, filters, and interpreting results.

• Step 3 Create a language resource in your resource group (Azure Portal)

Creating a Language Service Resource in Azure Portal

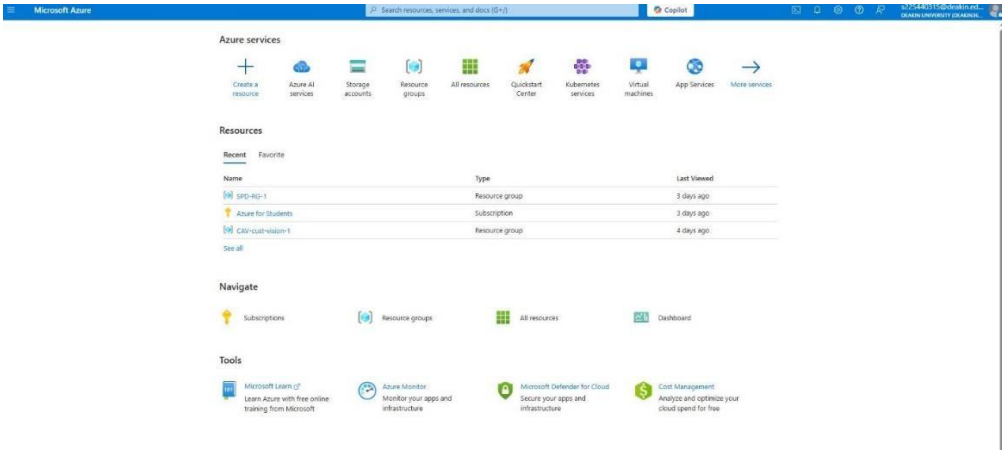
Follow these steps to create a **Language Service resource** in Azure for working with **Custom Question Answering** and **Conversational Language Understanding (CLU)**:

Steps:

1. **Go to Azure Portal:**
   https://portal.azure.com
2. **Navigate to Your Resource Group:**
   Select the appropriate resource group where you want to deploy the service.
3. **Create a New Language Resource:**
   - Search for **"Language"** in the search bar.
   - Select **"Language"** from the services list and click **"Create."**
4. **Choose Features:**
   - In the configuration page, under **Features**, select:
     - **Custom Question Answering**
     - **Custom Text Classification**, **Conversational Language Understanding**, etc.

Important Notes:

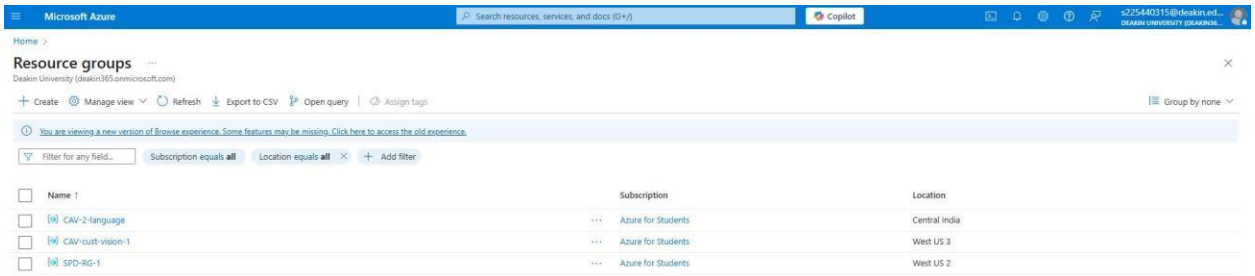
- You **only need Custom Question Answering** to use the QnA maker capabilities.
- However, we're selecting **additional options** like **CLU** because this case study will continue into **intent and entity detection** later.
- **Free Tier Limitation:**
  Azure allows only **one Free Tier Language Resource per subscription**, regardless of selected features.
  - If you've already created one, you must **delete it** or choose a **paid tier** to create another.



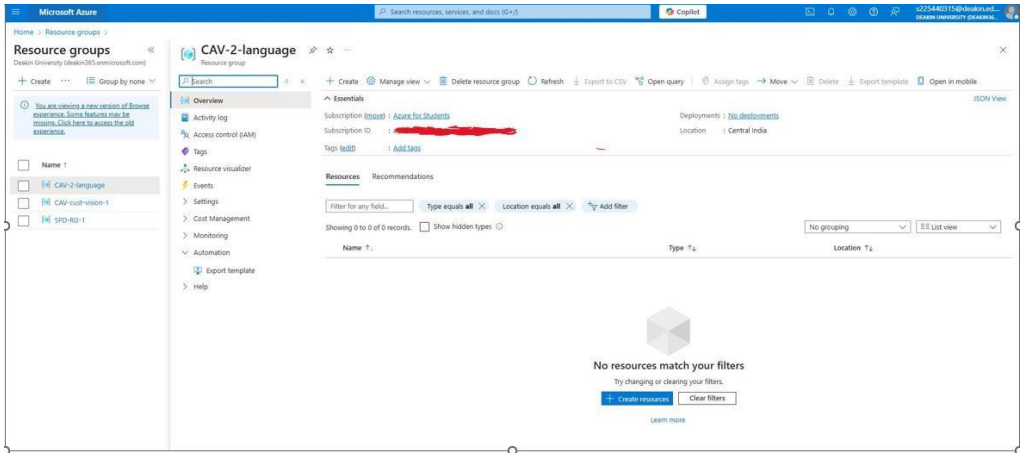
Step 3A. Creating resource group as per following click on review and create

Step 3B. Click on and create

Step 3C. we can see resource group is created successfully

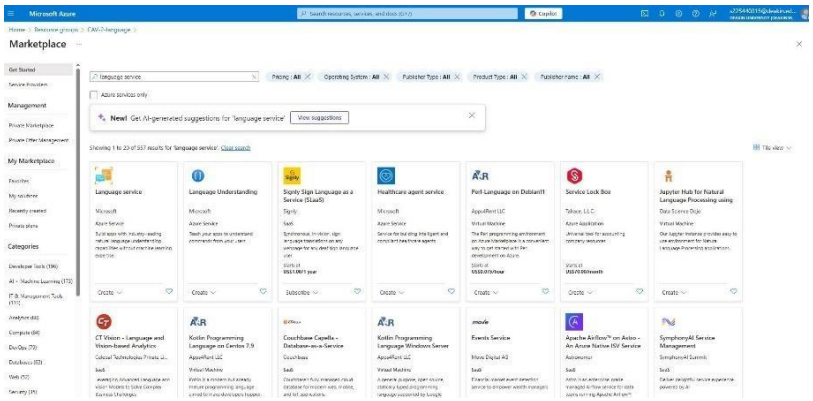


Step 3D. Click on newly created resource group named as CAV-2-language

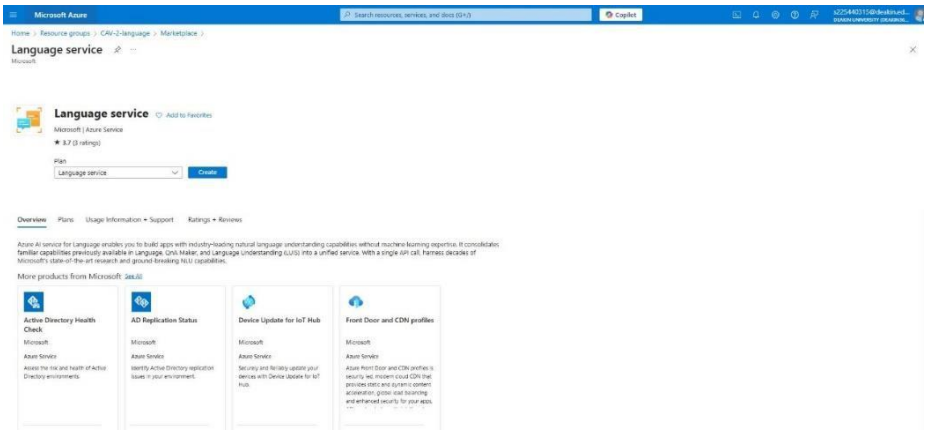


Step 4E. After clicking on resource group click to create resources

- Click on Language service



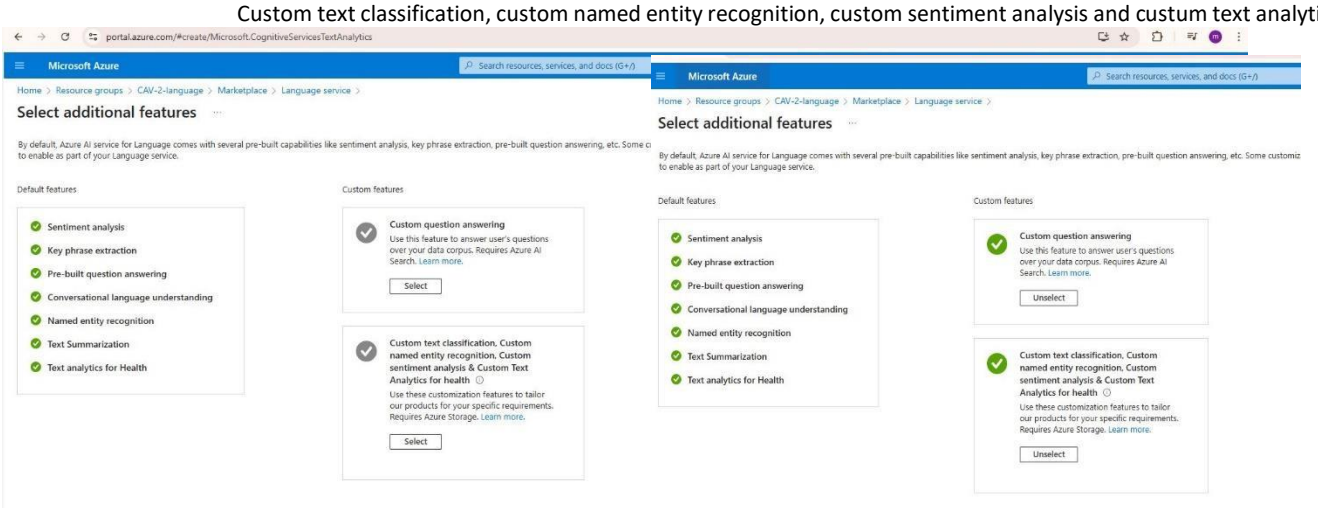
Step 3F. click to create



Step 3G.Azure language services provides many default services like as follows

- Sentiment Analysis
- Key phrase extraction
- Pre-built question answering
- Conversational language understanding
- Named entity recognition
- Text summarization
- Text analysis for health

We select some other services also as per problem statement we select both custom question answering
Custom text classification, custom named entity recognition, custom sentiment analysis and custum text analytics for health



Step 3H. Setup azure language services as follow premium LRS also free service

---

Azure Language Service Pricing Tiers Explained
When creating a **Language resource** in Azure, you'll need to choose a **pricing tier**. These determine:

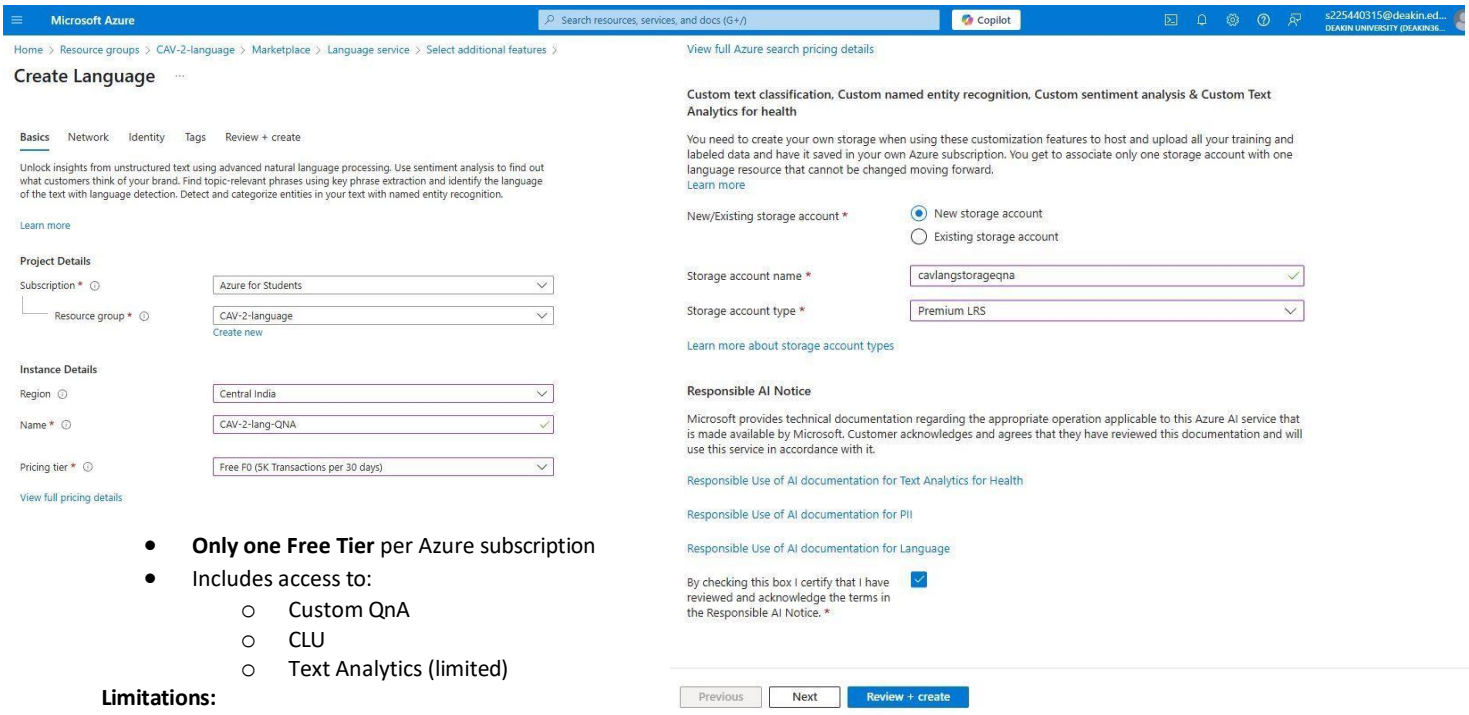
- **Cost**
- **Performance**
- **Features & Capacity**

Free Tier (F0)
Good For:

- Learning
- Prototyping
- Personal/small demo projects

Key Details:

- **Limited usage quota** (e.g., a few thousand transactions/month)



- **Only one Free Tier** per Azure subscription
- Includes access to:
  - Custom QnA
  - CLU
  - Text Analytics (limited)

Limitations:

- Not suitable for production use
- Lower performance
- No SLA (Service Level Agreement)
- Limited scalability

Standard/Paid Tier (S or S0)
Good For:

- Production apps
- Moderate to large workloads

Key Details:

- Pay-as-you-go model based on usage
- Includes:
  - All language features (Custom QnA, CLU, Sentiment Analysis, etc.)
- Scalable and suitable for high traffic
- SLA-backed service

Premium Tier (S0 with LRS – Locally Redundant Storage)
Good For:

- Enterprises
- Mission-critical applications

Key Details:

- Includes **Locally Redundant Storage (LRS)**:
  - Ensures your data is **stored redundantly** across multiple physical locations in a single region
  - Improves **data durability and reliability**
- Better data compliance and reliability
- Often required for regulatory environments

Why Choose Premium (LRS)?

- If you're building a **real-world, business-critical** solution (like a healthcare assistant, retail bot, or government system)
- You need **high availability**, **data reliability**, and **long-term support**
- You're integrating with other **enterprise-grade Azure services**

✅ Summary Comparison:

Tier	Cost	Use Case	Limits	Redundancy	SLA
Free (F0)	$0	Learning & Testing	One per sub, low quota	❌	❌
Standard	Pay-as-you-go	Production & Scaling	Higher quotas	❌	✅
Premium (LRS)	Higher cost	Mission-Critical Apps	Highest quotas	✅ LRS	✅ (High SLA)

Storage Account in Microsoft Azure is a foundational step for many cloud-based solutions — including **Language Services** like **Custom Question Answering**, **Conversational Language Understanding (CLU)**, and more. Here's a clear explanation of **why** you need to create one:

What is an Azure Storage Account?

An **Azure Storage Account** provides **durable, secure, scalable** cloud storage for various types of data:

- Files
- Blobs (binary large objects)
- Tables
- Queues
- Disks

It's like a **container** that holds all your structured and unstructured data.

Why Create a Storage Account for Azure Language Services?

When using services like **Custom Question Answering (QnA Maker)** or **CLU**, the storage account is used for:

1. Storing Project Artifacts

- Documents, FAQs, and datasets you upload for QnA
- Logs, trained models, and language project definitions

2. Auto-saving Your Resources

- Uploaded files (PDF, DOCX, TXT) used to build a knowledge base
- Configuration files used for managing intents/entities

3. Enabling Versioning & Recovery

- Azure automatically stores snapshots and versions of your models and resources
- Useful for rollbacks or audits

4. Separation of Concerns

- Keeps data storage **independent from compute**
- Makes scaling, migrating, or managing resources easier

5. Integration with Other Services

- Can be accessed by other Azure services like Web Apps, Cognitive Search, Logic Apps, etc.
- Enables automation and orchestration

Benefits of Using a Dedicated Storage Account

Feature	Benefit
Secure Access	Encryption at rest, identity-based access control (RBAC, SAS tokens)
Redundancy Options	Choose from LRS, GRS, ZRS, etc., to match durability needs
Scalability	Can handle massive datasets and traffic, grows as your project scales
Backup & Restore	Easier data recovery and fault tolerance

Example Use Case in Custom QnA

If you're building a retail Q&A bot:

- You upload 10 PDF documents with product FAQs.
- These are stored in the **Storage Account**.
- Azure Language Service reads those files to generate a **knowledge base**.
- When users ask questions, answers are fetched based on content from storage.

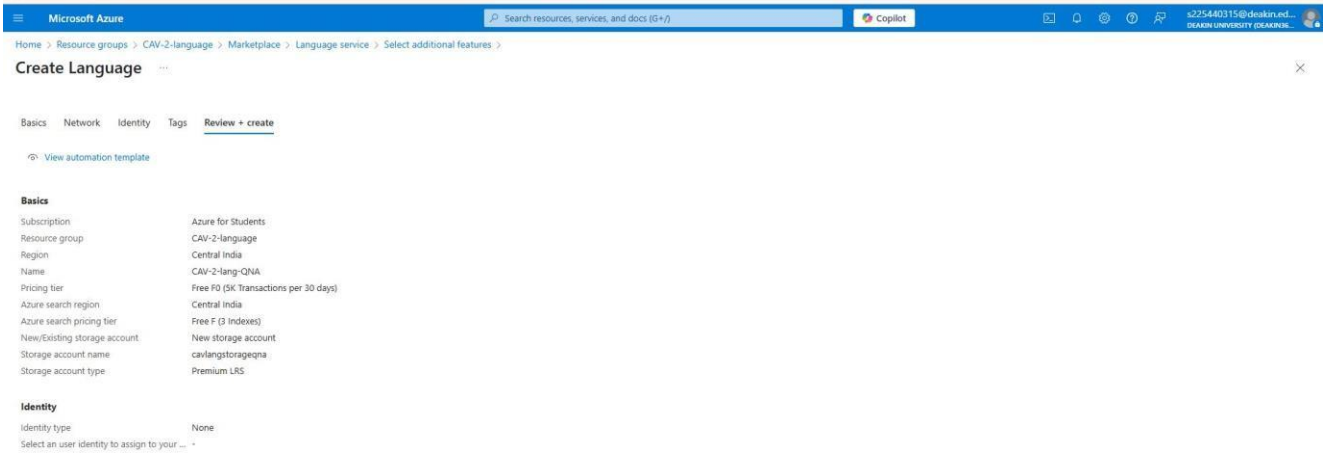
Summary

A **Storage Account** in Azure acts as the **backbone for storing files, training data, models, and configurations** used by various Azure services. It ensures **persistence, scalability, and security** of your data — making it a necessary step in setting up services like QnA and CLU.

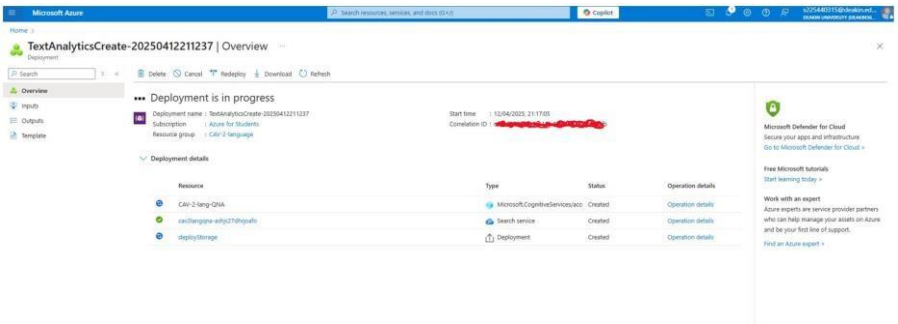
Step 3I. Verify and Click on create review it once again

Key Observations & Conclusion

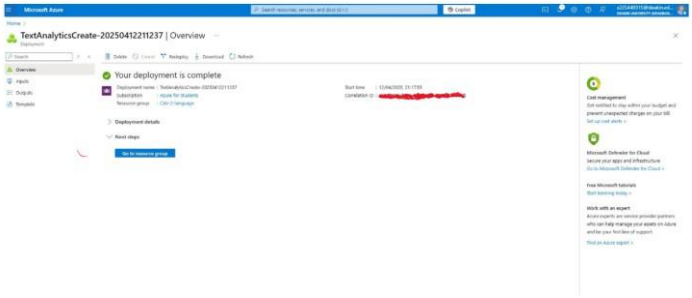
1. **Free Tier (F0)** was selected – suitable for student projects and learning, but it has usage limitations (e.g., limited transactions per month, one resource per subscription).
2. **Region** is Central India – this can affect latency and availability if your user base is in or near that region. Always ensure your clients and services are in proximity for better performance.
3. **Premium LRS Storage** was chosen – even though you are using a **Free Tier for the Language Service**, you've opted for **Premium, Locally Redundant Storage**:
   - Ensures **higher durability**, **redundancy**, and **enterprise-grade reliability** for your files and data.
   - Useful if you plan to store **important documents**, **large KBs**, or plan to **scale to production** later.
4. **Good combination for growth**:
   - Free Tier helps during early dev/testing.
   - Premium Storage ensures your **data is secure and production-ready**, even if your service tier is basic for now.



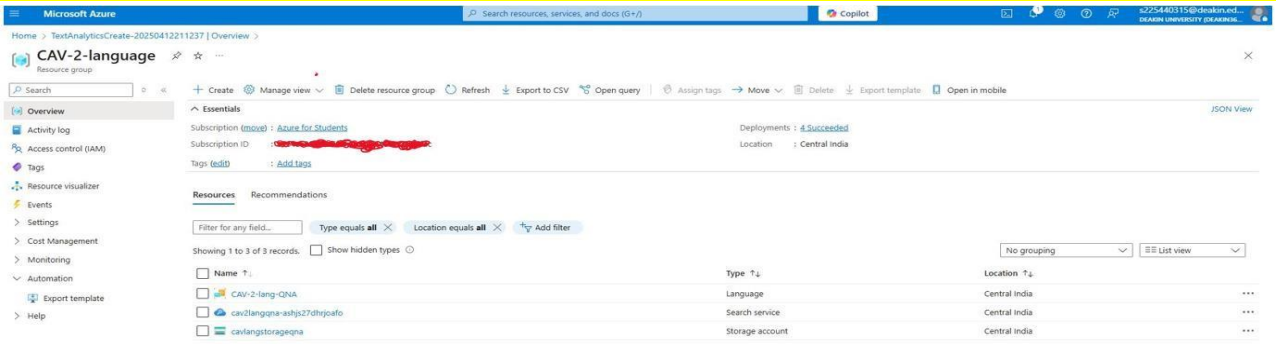
Step 3J. Deployment is in progress



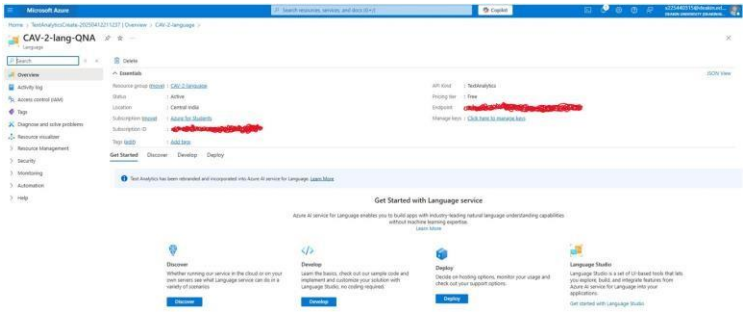
Step 3K. takes few minutes to deployment



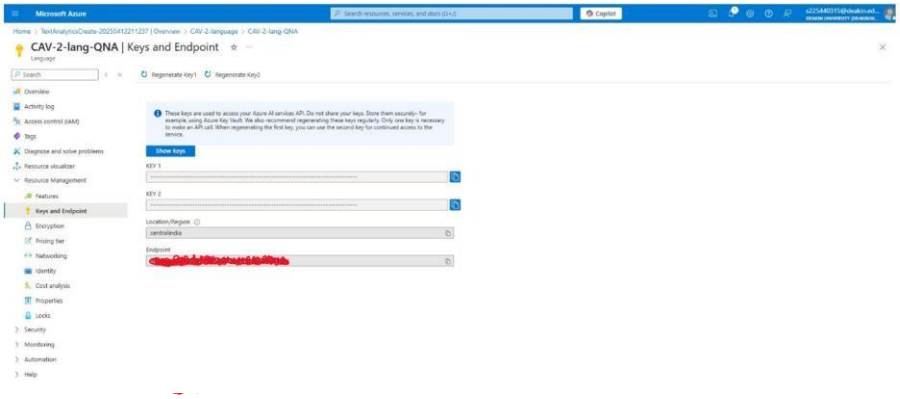
Step 3L. Click on overview of resource group



Step 3M. Click on require resources.

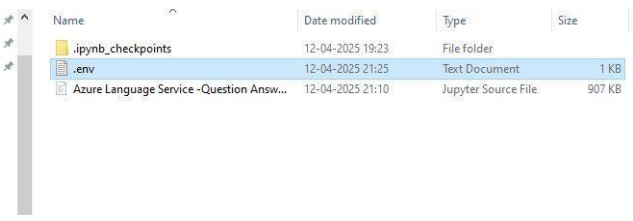


Step 3N. verify and view keys and endpoints



Step 3O. storing endpoint and key

- Open notepad
- Create proper name for endpoint and subscription key
- These keys will be required for using model of azure for language services



Step 4: importing secured credentials for initializing Azure QnA

Step 4 Initializing Azure QnA and Authoring Clients with Secure Credentials ¶

```
# get service secrets
load_dotenv(r"E:\2. MDS DEAKIN UNIVERSITY\TASK\Task6\Hands On Ipynb\.env.txt")
endpoint = os.environ.get("endpoint")
key = os.environ.get("subscription_key")

# Please note, we will create two clients, one for creating the project (Authoring) and one for querying it (Question Answering)

authoring_client = AuthoringClient(endpoint, AzureKeyCredential(key))
qna_client = QuestionAnsweringClient(endpoint, AzureKeyCredential(key))
```

Code Explanation

python

Load environment variables from the .env file

load_dotenv(r"E:\2. MDS DEAKIN UNIVERSITY\TASK\Task6\Hands On Ipynb\.env.txt")
endpoint = os.environ.get("endpoint")
key = os.environ.get("subscription_key")

This block loads sensitive information (like the endpoint and subscription key) from a .env.txt file. Using environment variables helps maintain the security of authentication credentials and keeps them separate from the source code.

---

python
 # Initialize the Authoring and Question Answering clients
authoring_client = AuthoringClient(endpoint, AzureKeyCredential(key))
qna_client = QuestionAnsweringClient(endpoint, AzureKeyCredential(key))

Two clients are created using the provided endpoint and key:

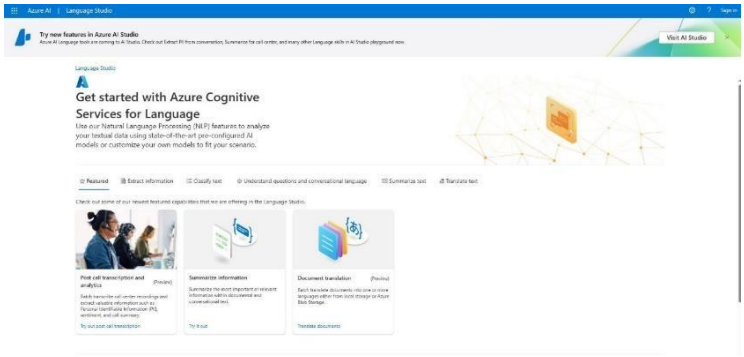
- **AuthoringClient**: Used to define and manage Question Answering projects, including creating, editing, and publishing knowledge bases.
- **QuestionAnsweringClient**: Used to interact with the published project during runtime, enabling users to send queries and receive relevant answers.

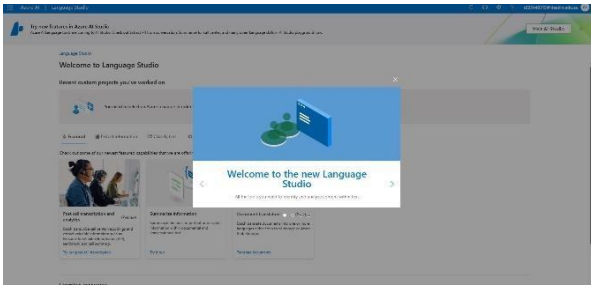
📑 Conclusion

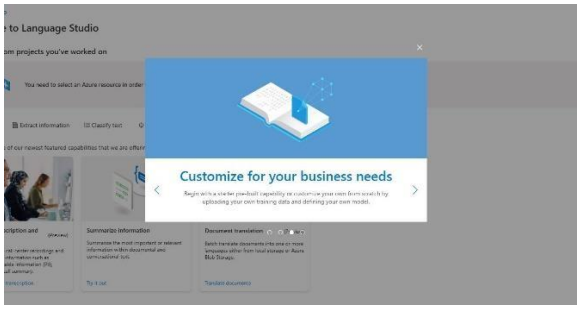
This setup ensures secure and modular interaction with Azure Language Services by separating configuration data from core logic and establishing dedicated clients for both authoring and runtime functionalities.

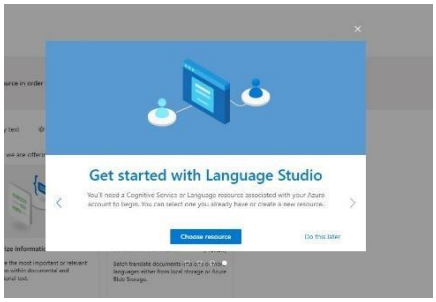
Step 5: Language Studio Setup

Step 5A: Creating project on language studio





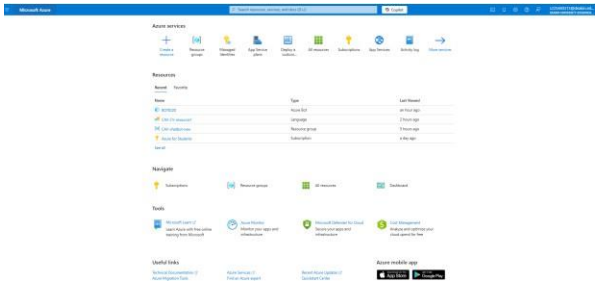




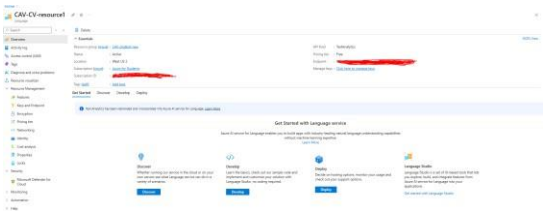
Sign in to portal of language studio for custom question and answering and to create project and knowledge base
You can create project manually on azure portal
2) through python sdk in our project we creating with python

Steps for creating manually as follow

1) Login Azure portal and open resource group

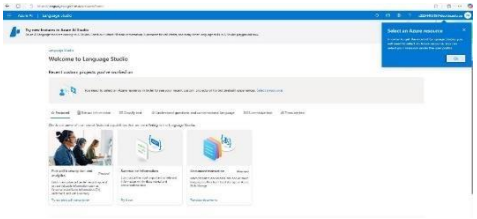


2) Click on language resource you have created

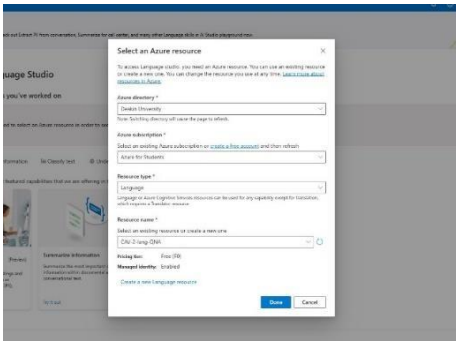


3) Click on language studio

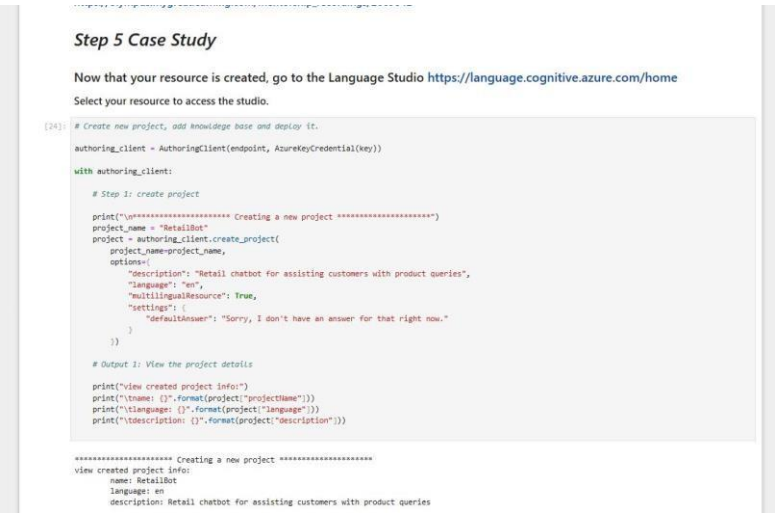
4) Sign in language studio



4) Fill up credentials add resources information given name to project



Step 5B: Creating project on language studio (using python sdk)



Step 5 Case Study

Now that your resource is created, go to the Language Studio https://language.cognitive.azure.com/home

Select your resource to access the studio.

```
# Create new project, add knowledge base and deploy it.

authoring_client = AuthoringClient(endpoint, AzureKeyCredential(key))

with authoring_client:

    # Step 1: create project

    print("\n********** Creating a new project **********")
    project_name = "RetailBot"
    project = authoring_client.create_project(
        project_name=project_name,
        options={
            "description": "Retail chatbot for assisting customers with product queries",
            "language": "en",
            "multilingualResource": True,
            "settings": {
                "defaultAnswer": "Sorry, I don't have an answer for that right now."
            }
        })

    # Output 1: View the project details

    print("view created project info:")
    print("\tname: {}".format(project["projectName"]))
    print("\tlanguage: {}".format(project["language"]))
    print("\tdescription: {}".format(project["description"]))
```

```
********** Creating a new project **********
view created project info:
        name: RetailBot
        language: en
        description: Retail chatbot for assisting customers with product queries
```

This Python code demonstrates how to programmatically create a **Custom Question Answering (CQA)** project in **Azure Language Studio** using the AuthoringClient. It starts by initializing the client with the required **endpoint** and **subscription key**. Inside a context manager (with authoring_client), the script performs the first major step: **creating a new project** named "RetailBot". The project includes basic configuration such as a description ("Retail chatbot for assisting customers with product queries"), the language ("en"), and an optional multilingual setting. It also sets a **default fallback answer** for when no good match is found. Once the project is created, it prints out key project details like the name, language, and description to confirm successful setup.

Why is Language Studio Required?

Language Studio is an **essential tool** for working with Azure Cognitive Services – Language, especially when you're creating and managing projects like **Custom Question Answering (CQA)**.

It provides a **centralized, user-friendly platform** that allows users to work with powerful AI models without needing to write code. It's ideal for building, training, and deploying language understanding solutions.

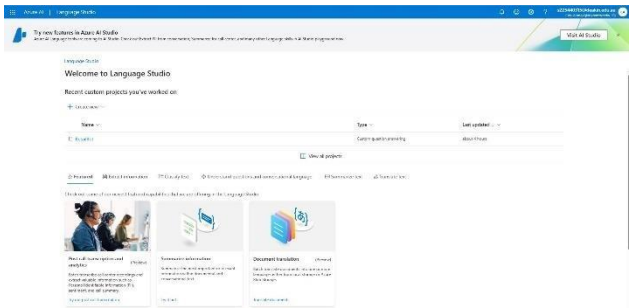
Purpose of Language Studio

Purpose	What You Can Do
Project Creation	Create Custom Question Answering (CQA), Conversation Language Understanding (CLU), or other language projects.
Data Management	Upload, edit, or manually create **QnA pairs**, documents, and alternate phrasings.
Training the Model	Train your knowledge base with updated content for better language understanding.
Deployment	Deploy your trained model to an endpoint for API consumption or chatbot integration.
Testing & Evaluation	Run test questions directly in the portal and view confidence scores, ranking, etc.
Integration Readiness	Copy **Prediction URL** and **Deployment names** required for API calls or bot setup.

Who Should Use It?

- **Non-technical users**: Can build QnA projects without code
- **Content creators**: Manage FAQs and answers easily
- **Developers**: Get trained models and integrate them into apps, bots, or APIs

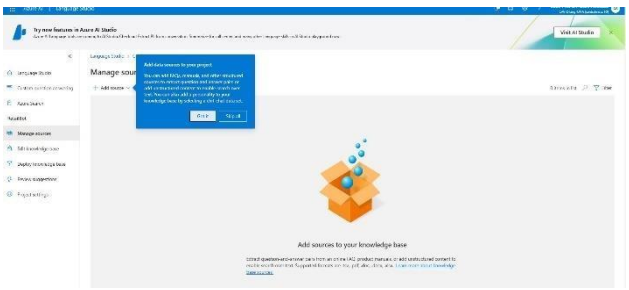
Verify from the azure language studio your app is created there with name as RetailBot



View the project on the Language Studio:

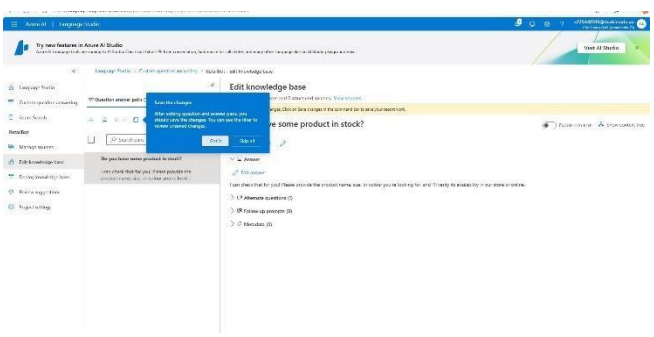
1. Navigate to https://language.cognitive.azure.com/home
2. Before executing the above steps, we can see that the Language Studio had no projects. Now, you can see your project created here.
3. Click on the project title to view the details like source name, url, and whether it is structured or unstructured.
4. Clicking on the source on this page shows you the questions and answers.

Step 5.C: Add a knowledge base



- We are adding two knowledge bases here. A minimum of one knowledge base is required.
- Please note, the knowledge base cannot have more than 3 sources (urls, files) in Free Tier.
- In other words, you can add upto three sources/indices in one project, under the free tier.

1. Using Azure Language Studio UI



- Manually but visual
- Add sources by uploading files (TSV, PDFs, URLs)
- Good for quick edits and tests

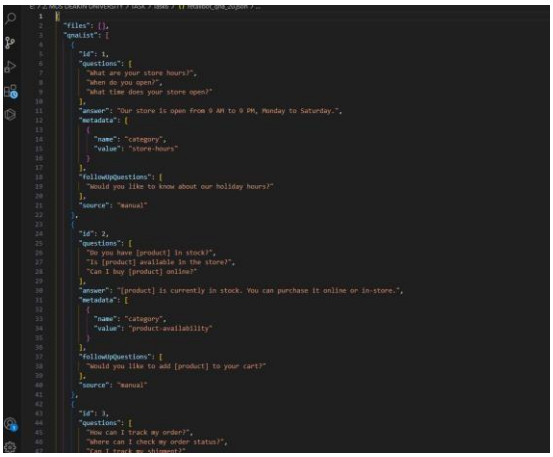
→ **YES, this is correct and official.**

After creating a project in Language Studio, you can click on “Edit Knowledge Base” to manually enter question-and-answer pairs that will form the core of your Custom Question Answering model. In this editor, you can provide a primary question along with its corresponding answer. Additionally, you have the option to enhance the model’s understanding by adding alternate phrasings of the question to improve recognition of user queries. You can also include follow-up prompts, which help guide users through a conversational flow by suggesting related questions. Furthermore, metadata fields can be used to categorize, tag, or filter content within the knowledge base for advanced query handling. This manual approach gives content creators full control over the knowledge base without needing to write any code.

---

2. Using Python SDK (Code) (URL)

- You generate .tsv files programmatically



- Then upload via portal OR host them and connect via code using sourceUri

→ YES, this is also fully supported and great for automation.

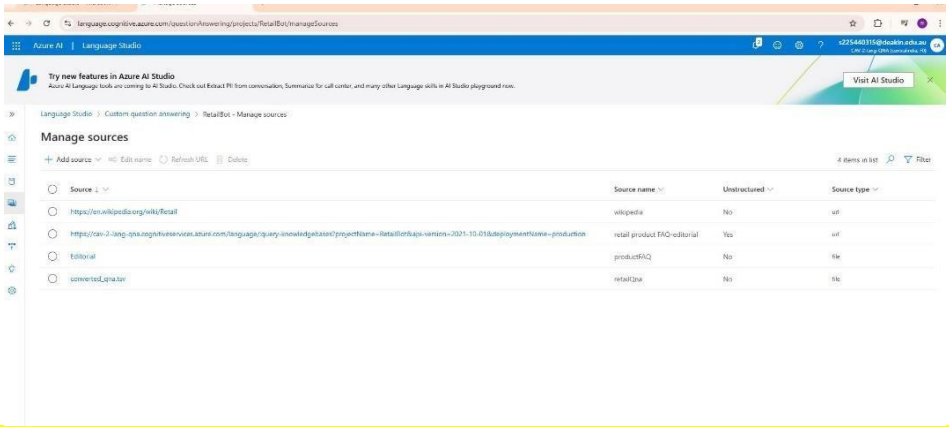
In this step, the code uses the AuthoringClient to add a knowledge base source to the previously created project. Inside the with authoring_client block, it calls the begin_update_sources() method to add a new source, which can be a URL, file, or Azure Blob. In this example, the source is a URL-based FAQ document for retail products, labeled as "retail product FAQ-editorial". It is important to note that under the Free Tier, a project can have a maximum of three sources in total. After initiating the source update, the script waits for the operation to complete using .result(). Once the source is successfully added, it lists all knowledge base sources linked to the project using list_sources(), and prints out details such as the name, type of source (e.g., URL), and the source URI. This step is essential for feeding content into the model for training and deployment.

Creating knowledge base of external file generated in json format assuming we are importing it from database of app

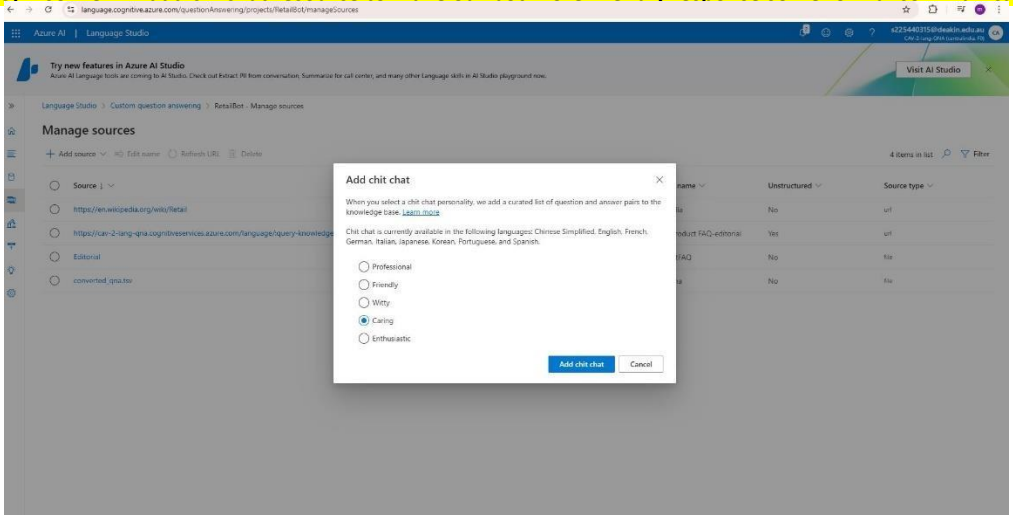
Converting External QnA JSON Data to TSV Format for Integration and Processing (files)

This script is designed to convert a QnA JSON file—retrieved from an external application or website database—into a structured TSV (Tab-Separated Values) format for further processing or integration. The JSON file is assumed to contain a structured list of question-answer entries (qnaList), each with fields such as questions, answers, metadata (e.g., category), follow-up questions, and the original data source. To avoid duplicate entries, the script uses a set to track unique combinations of these fields. It then constructs a clean table where each question is mapped to its corresponding answer and metadata, and organizes the results into a pandas DataFrame. Finally, it exports the data into a .tsv file—ideal for use in data analysis, bulk editing, or uploading to systems like Azure Language Studio or knowledge base editors.

This .tsv file we will add manually to language studio



3) Also we will add chi chat resource to make our bot more friendly response some formal communication (chitchat)



Once again we run above code to add all knowledge base to out project for deployment and we got this output
We also included external url from knowledge base, like wiki-pedia, zolando but it did not solved purpose we removed it

```
********** Adding a knowledge base **********

list project sources
Knowledge base name: retailQna
        Source: converted_qna.tsv
        Source kind: file
        Source Uri: https://qsincame.blob.core.windows.net/portal-files/356a6632-ed32-a48b-e290-5f388afa60a2-converted_qna.tsv
Knowledge base name:
        Source: qna_chitchat_Caring
        Source kind: file
        Source Uri: N/A
Knowledge base name: zalando
        Source: https://www.zalando.co.uk/faq/Returns-and-Refunds
        Source kind: url
        Source Uri: https://www.zalando.co.uk/faq/Returns-and-Refunds
```

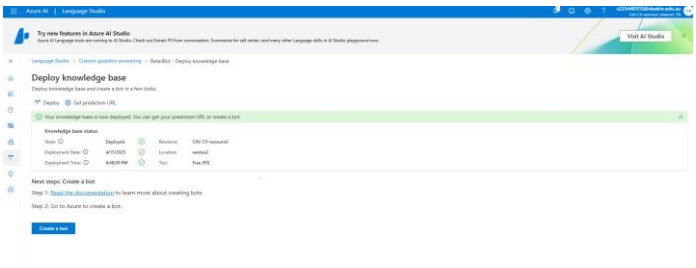
Step 5D deployment of project

Step 5.3 deploy the project

```
authoring_client = AuthoringClient(endpoint, AzureKeyCredential(key))
with authoring_client:
    # Step 3: deploy the project

    print("\n********** Deploying the project **********")

    deployment_poller = authoring_client.begin_deploy_project(
        project_name=project_name,
        deployment_name="production"
    )
    deployment_poller.result()

    # list all deployments
    deployments = authoring_client.list_deployments(
        project_name=project_name
    )

    print("view project deployments")
    for d in deployments:
        print(d)
```

```
********** Deploying the project **********
view project deployments
{'deploymentName': 'production', 'lastDeployedDateTime': '2025-04-15T13:10:27Z'}
```

This code demonstrates the process of deploying a project in Azure Language Studio using the AuthoringClient. After initializing the client with the provided endpoint and API key, it moves to the deployment phase. The begin_deploy_project () method is called to **deploy the project** named RetailBot under the deployment name "production". This method initiates the deployment process, and the .result() function ensures that the script waits for the deployment to finish before proceeding. After the deployment is completed, the script retrieves a list of all deployments associated with the project by calling list_deployments(). The deployment details are then printed to the console for review, allowing you to verify the deployment status and any additional metadata associated with each deployment.



After deploying through python sdk the model will generate prediction Url we can save it for model evaluation and testing
Also om language studio there are other information like date time of deployment, location, resource name etc.

Note: date and time and information or screen shot may vary due to multiple trial and testing of resources

Another way to create project Using REST API (Direct HTTP Calls) (not for this project just for information)

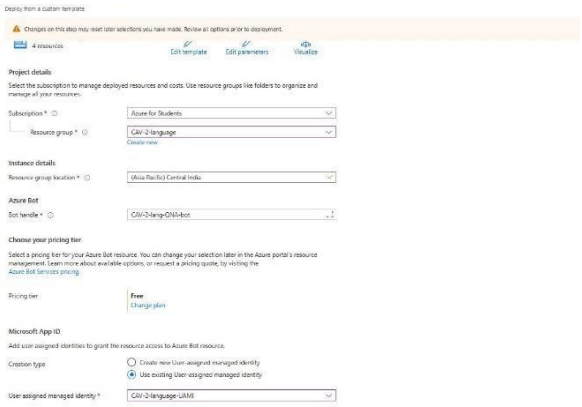
This is the **third official method**:

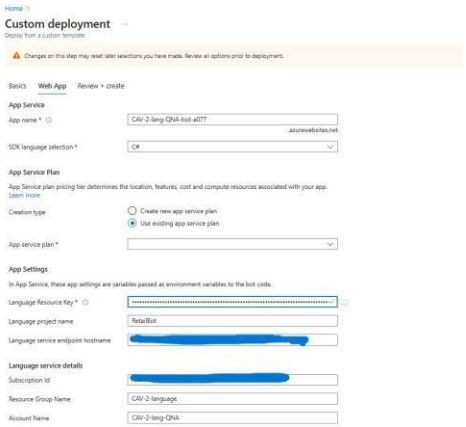
- You call the Azure Language Service REST API to create projects, add/update sources, and deploy — *without using Python SDK or the portal*.

Great for use in systems where Python isn't available (e.g., Node.js, bash scripts, Postman, custom backend, etc.)

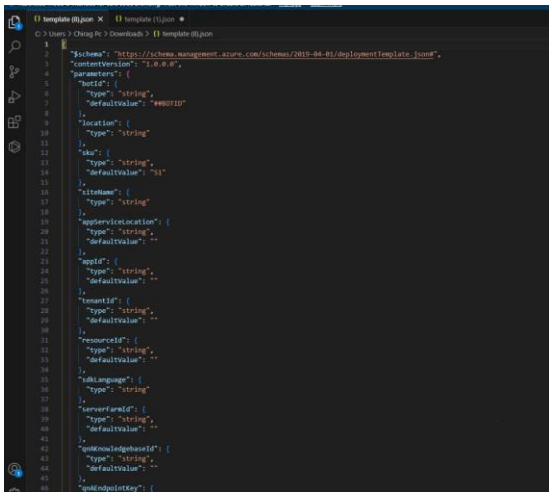
Step 6: Create Azure bot

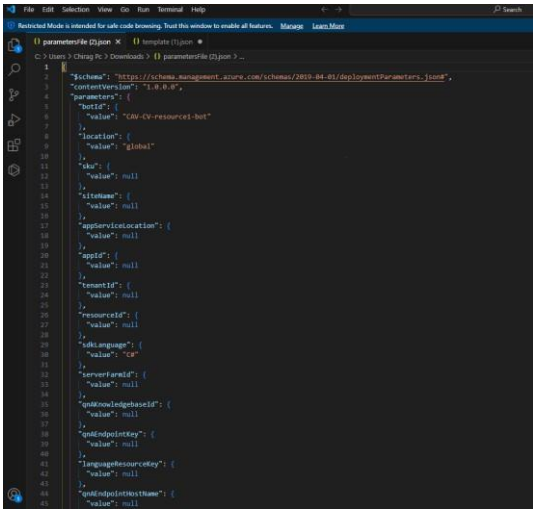
After the project is successfully deployed in Language Studio, the next step is to **create a bot that uses the deployed model**. To automate and streamline this process, we use an **Azure Resource Manager (ARM) custom deployment template**. This template allows us to define all the resources needed for the bot — such as the Web App Bot, App Service Plan, and Language Studio project integration — in a structured JSON file. By deploying the ARM template, we can provision all necessary Azure resources in one go and link the deployed Language Studio project to the bot seamlessly. This approach ensures consistency, reduces manual configuration errors, and enables easy redeployment in different environments (like development, testing, or production).





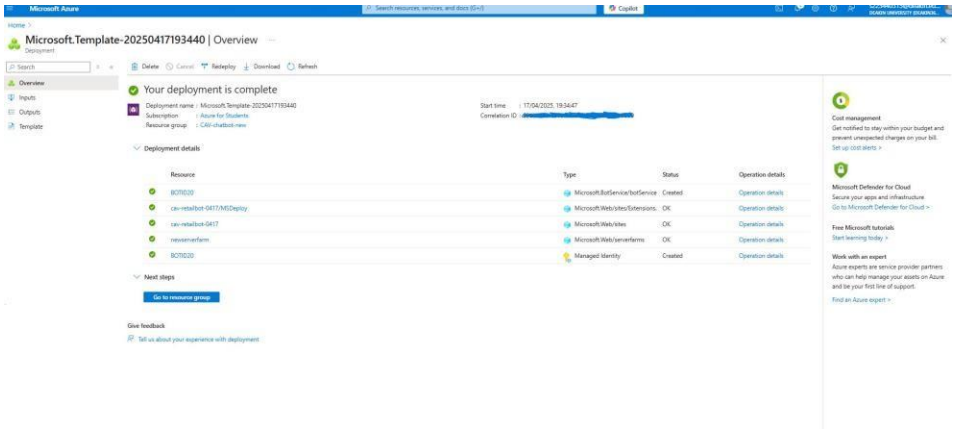
Once the Language Studio project is successfully deployed, the next step is to integrate it into a chatbot by creating and deploying an Azure Bot using a custom ARM (Azure Resource Manager) template. This template automates the process of setting up all the necessary Azure resources required for the bot. The process begins by downloading the deployment template and parameters file in JSON format from the Azure portal. These files define the infrastructure and configuration needed for deployment, such as the bot service, web app, and integration with the deployed Language Studio project.
After downloading, the parameters file is manually edited to reflect the desired configuration. This includes updating fields like the bot's name, region (location), endpoint URL, and description. These changes make the deployment specific to your bot's requirements and hosting setup. Once editing is complete, both the template and parameter files are uploaded back into the Azure portal using the "Deploy a custom template" option. From there, the deployment is reviewed and initiated.
This method provides a repeatable and customizable way to deploy bots at scale and is especially useful in production environments where consistency and automation are important.
Downloading that format in of template and parameters manually editing and uploading it again the format in Jason is as follows



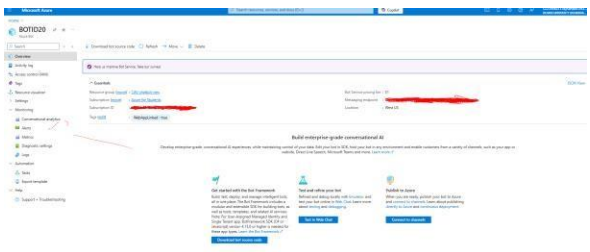


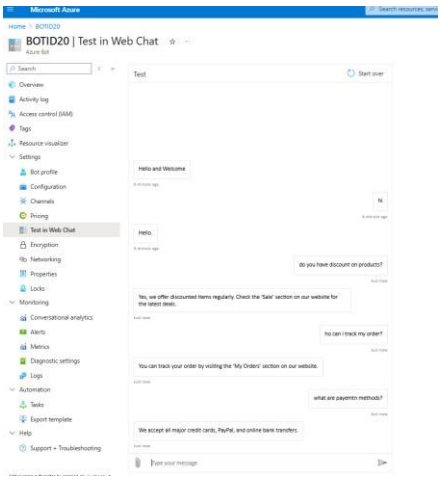
Here is the step-by-step process

1. Please Prefill the data like RG, Language key, bot name, creation type etc as you mentioned in the screenshot above.
2. Then go back to 1st step and download the template by clicking on "Edit Template" (ignore UI changes prompt and say yes) and Download
3. Then go to "Edit parameter"((ignore UI changes prompt and say yes) as mentioned in below screenshot
   (Go to Custom Template link and upload the files of template and parameter, Put RG name in prefilled template and deploy)
4. Go to Custom template link and click on "Build you own template in the editor"
5. Upload the Template file from step-2 in load file option for "Edit Template" and save.
6. Then upload the parameter file from step-2 in "load file" option from "Edit parameters" and save
7. You will see the details like language key, creation type etc prefilled now
8. Put details of Resource group again and deploy.
9. Once deployed, you can locate the bot and test it from webchat.



Step 7 Test bot on Azure web chat service

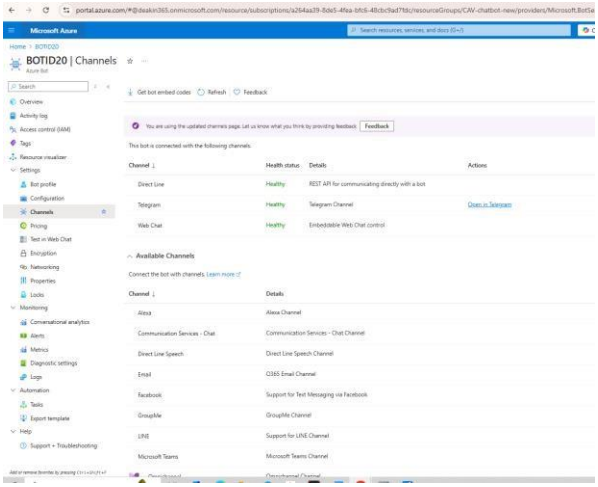


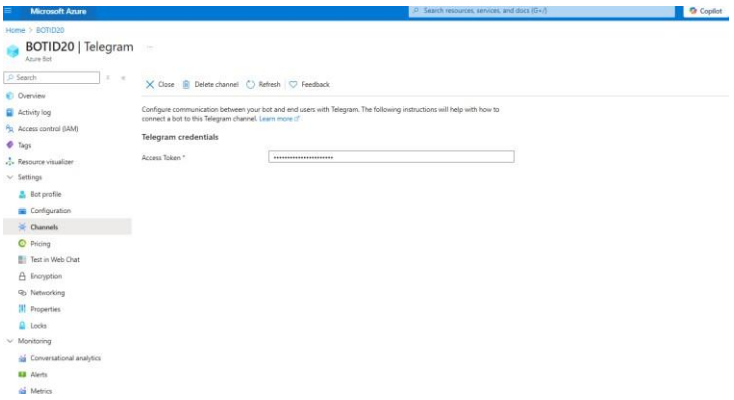


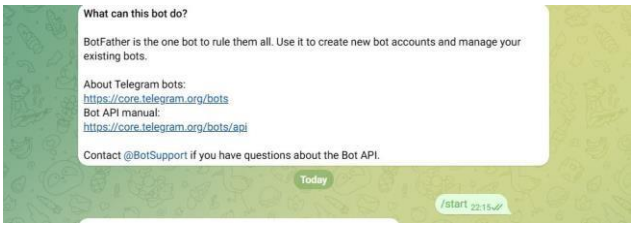
We have successfully performed Web Chat testing on the Azure Portal, and the bot is responding satisfactorily. The next step is to deploy the bot and integrate it with social media platforms such as Telegram for broader accessibility and real-world interaction.

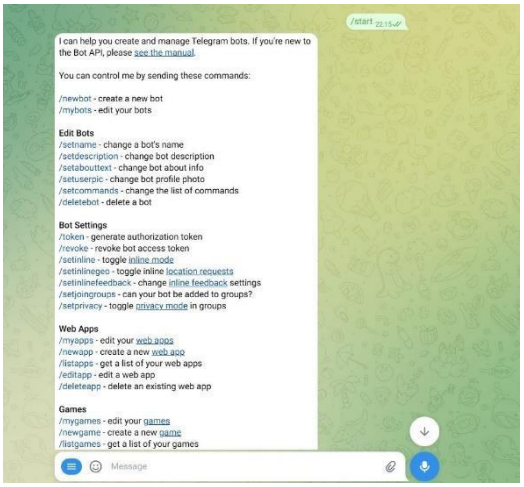
Step 8: integrating it with telegram

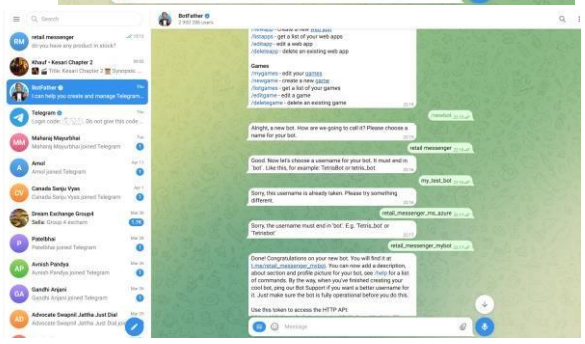
1) We click on telegram available channel and enter access token from telegam

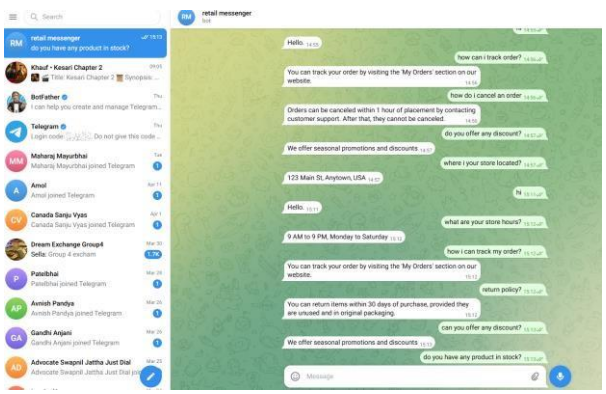












Connecting Telegram to Azure Bot

1. **Create a Bot on Telegram**
   - Open the Telegram app and search for **BotFather**.
   - Start a chat with BotFather and send the command /newbot.
   - Follow the instructions: choose a name and a unique username for your bot (ending with *bot*, e.g., myretailbot).
   - Once created, BotFather will give you a **Bot Token**. Save this token securely; it's needed to connect with Azure.
2. **Go to Azure Portal**
   - Navigate to your **Bot Channels Registration** or **Web App Bot** in the Azure portal.
   - In the left-hand menu, select **Channels**.
3. **Add Telegram Channel**
   - Click on **Telegram** from the list of available channels.
   - In the configuration window:
     - Paste the **Bot Token** you received from BotFather.
     - Optionally, set up a webhook or additional security parameters.
   - Click **Save** to add the Telegram channel.
4. **Verify Connection**
   - After a successful connection, you will see the **Telegram icon** under the "Configured channels".
   - You can now message your bot on Telegram using its username.
5. **Test the Integration**
   - Open Telegram and search for your bot by the username.
   - Start a chat and type a test message to verify that your bot responds as expected.

Note: I have created steup and username and bot name as shown in image.

Finally tested out bot and began chatting successfully.

step 9: model testing and evaluation

```
[9]: import requests
from fuzzywuzzy import fuzz
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
# ----------------------
# CONFIGURATION
# ----------------------
PREDICTION_URL=PREDICTION_URLs
SUBSCRIPTION_KEY = key
SIMILARITY_THRESHOLD = 70 # Adjust as needed

HEADERS = {
    "Ocp-Apim-Subscription-Key": SUBSCRIPTION_KEY,
    "Content-Type": "application/json"
}
# ----------------------
# TEST QUESTIONS & ANSWERS
# ----------------------
test_data = [
    {"question": "What is your return policy?", "answer": "You can return the product within 30 days."},
    {"question": "can you track my order?", "answer": "You can track your order using the tracking ID sent to your email."},
    {"question": "How do I cancel an order?", "answer": "Orders can be canceled within 1 hour of placement by contacting customer support. After that, th
    {"question": "Tell me about shipping charges", "answer": "Shipping is free for orders above $50."},
    {"question": "What payment methods do you accept?", "answer": "any netbanking , gpay, Paypal, neft, credit, debit/credit card."}
]
# ----------------------
# EVALUATION
# ----------------------
y_true_binary = []
y_pred_binary = []
for item in test_data:
    payload = {"question": item["question"], "top": 1}
    response = requests.post(PREDICTION_URL, headers=HEADERS, json=payload)
    result = response.json()

    predicted_answer = result.get("answers", [{}])[0].get("answer", "No Answer")
    true_answer = item["answer"]

    similarity = fuzz.token_set_ratio(predicted_answer, true_answer)
    is_correct = similarity >= SIMILARITY_THRESHOLD
```

```
    is_correct = similarity >= SIMILARITY_THRESHOLD

    # Assign labels based on whether the prediction is correct or not
    y_true_binary.append(1)  # True answer is considered correct (True)
    y_pred_binary.append(1 if is_correct else 0)  # Predicted 1 if similar, else 0

    print(f"Q: {item['question']}")
    print(f"Answer: {true_answer}")
    print(f"Predicted: {predicted_answer}")
    print(f"Similarity: {similarity} → {'✅ Match' if is_correct else '❌ Mismatch'}")
    print("-" * 60)
# ----------------------
# METRICS
# ----------------------
print("\n🔍 Evaluation Metrics:")
precision = precision_score(y_true_binary, y_pred_binary)
recall = recall_score(y_true_binary, y_pred_binary)
f1 = f1_score(y_true_binary, y_pred_binary)
print(f"Precision: {precision:.2f}")
print(f"Recall:    {recall:.2f}")
print(f"F1 Score:  {f1:.2f}")
# ----------------------
# CONFUSION MATRIX
# ----------------------
conf_matrix = confusion_matrix(y_true_binary, y_pred_binary)
plt.figure(figsize=(6, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=["Incorrect", "Correct"], yticklabels=["Incorrect", "Correct"])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

Code Breakdown

1. Import Libraries

```
python
import requests
from fuzzywuzzy import fuzz
from sklearn.metrics import precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

- These are your imports:
  - requests: for making HTTP API calls to your Azure QnA endpoint.
  - fuzz (from fuzzywuzzy): for computing similarity between expected and predicted answers.
  - precision_score, recall_score, f1_score, confusion_matrix: metrics from sklearn to evaluate the model.
  - matplotlib and seaborn: for plotting the confusion matrix.

2. Configuration

```
PREDICTION_URL = PREDICTION_URLs  # Replace this with actual endpoint string
SUBSCRIPTION_KEY = key            # Replace this with actual subscription key
SIMILARITY_THRESHOLD = 70         # Minimum similarity score to consider the answer a correct match
```

- These values connect your script to Azure and define how strict your answer matching is.
- SIMILARITY_THRESHOLD decides whether the predicted answer is similar enough to be considered correct.

3. HTTP Headers

```
HEADERS = {
   "Ocp-Apim-Subscription-Key": SUBSCRIPTION_KEY,
   "Content-Type": "application/json"
}
```

- Standard Azure header with your subscription key for authentication.

4. Test Data

```
test_data = [
   {"question": "...", "answer": "..."},
   ...
]
```

- This is your manually prepared test dataset with:
  - A question to ask the bot.
  - The expected answer to compare with the prediction.

5. Evaluation Loop

```
y_true_binary = []
y_pred_binary = []

for item in test_data:
   payload = {"question": item["question"], "top": 1}
   response = requests.post(PREDICTION_URL, headers=HEADERS, json=payload)
   result = response.json()
```

- You send each question to the Azure QnA bot.
- top: 1 tells Azure to return the most relevant answer.

6. Prediction vs Ground Truth

```
predicted_answer = result.get("answers", [{}])[0].get("answer", "No Answer")
true_answer = item["answer"]
```

- Extract the predicted answer from the JSON response.
- Get the expected answer from the test data.

7. Fuzzy Matching

```
similarity = fuzz.token_set_ratio(predicted_answer, true_answer)
is_correct = similarity >= SIMILARITY_THRESHOLD
```

- fuzz.token_set_ratio compares the strings in a way that ignores word order and partial matches.
- If the similarity score is higher than your threshold (e.g., 70), it's considered a correct prediction.

---

8. Build Ground Truth and Prediction Labels

```
y_true_binary.append(1)
y_pred_binary.append(1 if is_correct else 0)
```

- You assume all test questions have a correct answer (1 in y_true_binary).
- Your model's prediction is 1 (correct) or 0 (incorrect), depending on whether similarity met the threshold.

---

9. Print Summary

python

```
print(f"Q: {item['question']}")
```

- This logs each question, expected answer, predicted answer, similarity score, and match status.

---

10. Metrics

python

```
precision = precision_score(y_true_binary, y_pred_binary)
recall = recall_score(y_true_binary, y_pred_binary)
f1 = f1_score(y_true_binary, y_pred_binary)
```

- Precision: Of the answers your bot said were correct, how many actually were?
- Recall: Of all actual correct answers, how many did the bot identify?
- F1 Score: Balance between precision and recall.

---

11. Confusion Matrix

python

```
conf_matrix = confusion_matrix(y_true_binary, y_pred_binary)
...
sns.heatmap(...)
```

- Shows the classification performance visually.
  - True Positive (TP) → Predicted correct, and it is correct.
  - False Positive (FP) → Predicted correct, but it's wrong.
  - True Negative (TN) → Predicted wrong, and it is wrong (you don't have these in your current setup).
  - False Negative (FN) → Predicted wrong, but it was actually correct.

Since you're only testing questions with known correct answers, y_true_binary only contains 1s — this means:

- You won't see True Negatives (bottom-left cell).
- Any mismatch will appear as False Negative (top-left: True=1, Predicted=0).

Conclusion / Suggestions

- Your setup is correct and well-structured.
- If you want more realistic evaluation (including False Positives and True Negatives), include some irrelevant questions too that your KB shouldn't be able to answer.
- This will help test how good your bot is at saying "I don't know**."**

Key Changes:

1. **Confusion Matrix**:
   - Plotted using seaborn and matplotlib for better visualization. This will give a clear view of how well the bot is performing in predicting correct vs incorrect answers.
2. **Conclusion**:
   - Based on the evaluation metrics (Precision, Recall, F1), the script prints some possible improvements:
     - If **Precision** is low, the model might be providing more incorrect answers.
     - If **Recall** is low, the model might be missing correct answers.
     - If **F1** is low, it indicates an imbalance between Precision and Recall.
     - Based on which metric is higher or lower, recommendations are provided for improvement.

Interpret the Output:

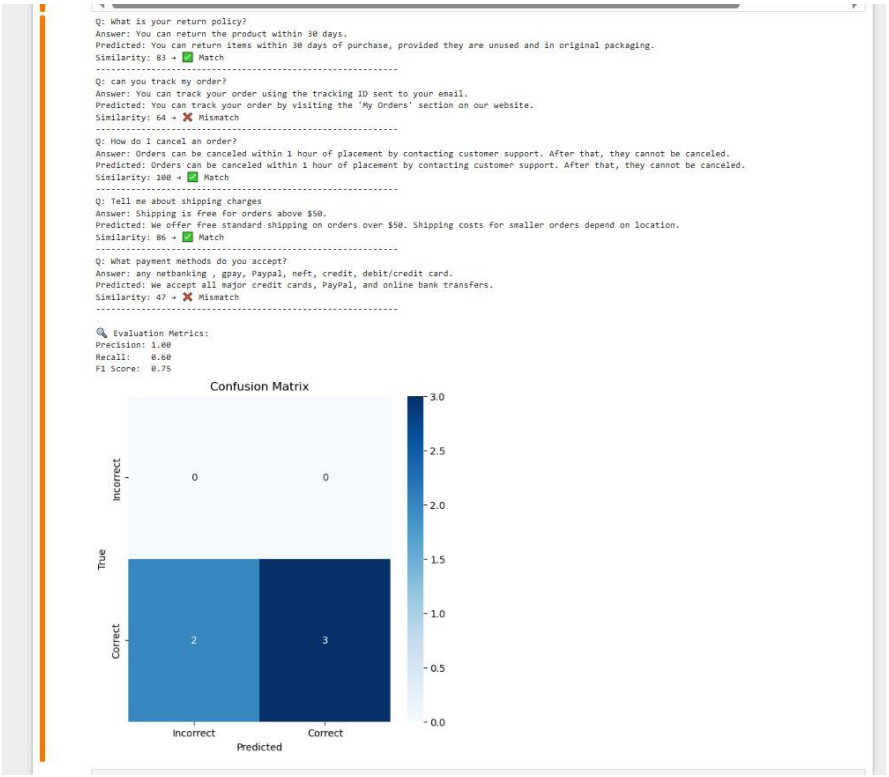
- **Confusion Matrix**: Shows how many true positives (correct predictions) and false positives/negatives the model has. It helps identify if the model is biased towards predicting certain types of answers.
- **Conclusion**: Based on the metrics (precision, recall, F1 score), we can adjust:
  - **Threshold tuning**: You may want to adjust the **similarity threshold** (e.g., 80%) for better precision and recall balance.
  - **Model improvement**: Adding more varied examples to the knowledge base can enhance recall and precision.

Conclusion on Next Steps:

If you get a low F1 score, precision, or recall, here are some ways to improve:

- **Increase the diversity** of questions and answers in the knowledge base.

- **Tune the similarity threshold** to reduce false positives or false negatives.
- **Expand the knowledge base** with more specific and varied content to ensure the model can handle a wider range of user queries.



Step 10 : Deleting project Recourses

Recommended Practice

After completing a project in **Microsoft Azure**, it is essential to review and delete all unnecessary or unused resources. These typically include:

- Resource Groups
- Virtual Machines (VMs)
- App Services and App Service Plans
- Azure SQL Databases or Cosmos DB
- Azure Storage Accounts
- Azure Cognitive Services (e.g., Language Studio, QnA Maker, CLU projects)
- Networking components (e.g., Public IPs, Load Balancers, Virtual Networks)

Importance of Resource Cleanup in Azure

1. **Cost Optimization:**
   - Azure charges for most services on a pay-as-you-go basis.
   - Even idle services (like VMs or reserved IP addresses) can accumulate costs.
   - Deleting unused resources helps avoid **unnecessary billing**.
2. **Security Best Practices:**
   - Unused services and public endpoints can be vulnerable to attacks.
   - Removing them **minimizes the attack surface** of your Azure environment.
3. **Better Resource Management:**
   - Helps maintain a **clean and organized Azure subscription**.
   - Simplifies navigation, auditing, and future planning.
4. **Quota and Subscription Limits:**
   - Azure enforces **subscription quotas** for various services.
   - Deleting unused resources **frees up limits** for future deployments.
5. **Prevent Unintentional Dependencies:**
   - Leftover resources may **interfere with future projects**, especially if names, ports, or IPs are reused.

Best Practices

- Use **Resource Tags** during project deployment for easy identification and cleanup.
- Utilize **Azure Cost Management** tools to analyse usage and locate idle resources.
- Consider implementing **automation (e.g., Azure Policy or Azure Automation)** for scheduled cleanup.
- Always **double-check dependencies** before deleting resource groups or shared components.

========== End Of Documentation ==========