

1ST EDITION

# Azure OpenAI Cookbook

Build intelligent applications with generative AI



SINA FAKHRAEE, PH.D. | DENNIS SAWYERS  
JIADONG CHEN | BALAMURUGAN BALAKRISHNAN

# Azure OpenAI Cookbook

Build intelligent applications with generative AI

**Sina Fakhraee, Ph.D.**

**Dennis Sawyers**

**Jiadong Chen**

**Balamurugan Balakreshnan**



BIRMINGHAM—MUMBAI

# Azure OpenAI Cookbook

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Publisher:** Vishal Bodwani

**Acquisition Editor:** Sathya Mohan

**Lead Development Editors:** Siddhant Jain and Oorja Mishra

**Development Editor:** Sameera Chawla

**Content Development Editor:** Saba Umme Salma

**Copy Editor:** Safis Editing

**Project Coordinator:** Yash Basil

**Proofreader:** Safis Editing

**Production Designers:** Deepak Chavan and Namkwon Cho

Production reference: 1070624

Published by Packt Publishing Ltd.

Grosvenor House, 11 St Paul's Square, Birmingham, B3 1RB, UK.

ISBN 978-1-80512-044-5

[www.packtpub.com](http://www.packtpub.com)

# Contributors

## About the authors

**Sina Fakhraee, Ph.D.**, works at Microsoft as an Enterprise Data Scientist and Senior Cloud Solution Architect. He has helped customers migrate successfully to Azure by providing best practices around data and AI architectural design and helping them implement AI and machine learning solutions on Azure. Previously, Sina worked at Ford Motor Company as a product owner for Ford's AI and machine learning platform. Sina holds a Ph.D. degree in computer science and engineering from Wayne State University and has taught undergraduate and graduate computer science courses part-time.

**Dennis Sawyers** leads data and AI at Simplify Inventions LLC. With over a decade of experience in creating innovative end-to-end solutions for companies such as Microsoft and Ford, Dennis is a recognized AI expert. He is the author of *Automated Machine Learning with Microsoft Azure*, published by Packt Publishing, and an alumnus of Carnegie Mellon University and the University of Michigan. A strong advocate for Microsoft Azure, Dennis emphasizes the importance of moving data and AI jobs to the cloud for enhanced automation, scalability, and deliverability.

**Jiadong Chen** is a distinguished Microsoft Most Valuable Professional (MVP) award winner, recognized as one of the technology industry's best and brightest for eight consecutive years. Based in Melbourne, Australia, Jiadong excels as a Cloud Architect/Senior Developer, specializing in Microsoft Azure cloud, Azure AI, DevOps, cloud security, and .NET/C#. With deep expertise in cloud architecture, he designs and implements secure cloud solutions. His impressive certifications include Microsoft Certified DevOps Engineer Expert, Cybersecurity Architect Expert, and Azure Solutions Architect Expert, among others. As a skilled communicator, he effectively conveys complex technical concepts and has spoken at major industry conferences, such as Microsoft Ignite.

**Balamurugan Balakreshnan** leads digital transformation through AI and cloud solutions, specializing in generative AI, AGI, LLM, IoT, big data, machine learning, and advanced analytics. With expertise in Microsoft technologies, open source solutions, and major cloud platforms, he excels in end-to-end cloud architecture and solution development. He integrates business, information, and technology for robust solutions, enhances decision making and automation through AI and machine learning, drives connectivity and insights with IoT and big data, and creates unified digital ecosystems. He delivers innovative, scalable solutions that align with business goals.

## About the reviewers

**Hyunsuk Shin** is a Principal Technical Architect specializing in cloud solutions for data and AI. He has a distinguished career at Microsoft, leading data science adoption and guiding architecture design sessions for enterprise clients. He excels in advanced software solutions, leveraging his expertise in cloud computing, distributed systems, and programming. Holding a bachelor's degree in engineering and pursuing Master of Science in Analytics at Georgia Tech, Hyunsuk is a certified expert in machine learning and data engineering. He is dedicated to empowering organizations to harness the power of cloud technology in today's digital landscape. Additionally, he is known for his excellent communication skills, which enable him to clearly explain complex technical concepts to diverse audiences.

**Jetro WILS** is a Cloud and Information Security Advisor, who began providing managed IT services as a teenager. He's a certified Microsoft Cybersecurity Architect, Azure Solutions Architect, and MCT. For 18 years, he has been active in various tech companies in Belgium. From developer to business analyst to product manager to cloud specialist, he has experienced digital evolution first-hand. Jetro is the founder of BlueDragon Security ([www.bluedragonsecurity.com](http://www.bluedragonsecurity.com)), where he helps organizations operate safely in the cloud.

# Table of Contents

## Preface

---

## 1

### Introducing Azure OpenAI and Building Your First Chatbot

---

Technical requirements	2	Deploying models to Azure OpenAI	14
Introducing generative AI	2	Getting ready	14
The many models behind generative AI	3	How to do it...	15
Proprietary AI models and the companies behind them	4	How it works...	18
Other important players behind the AI revolution	4	There's more...	18
		See also	18
Understanding AI use cases	5	Using models in the Playground	19
Understanding AI use cases for text	5	Getting ready	19
Understanding AI use cases for image generation	7	How to do it...	19
Understanding AI use cases for speech	8	How it works...	21
		There's more...	21
		See also	22
Getting started with Azure OpenAI Service	9	Building your first chatbot	22
Getting ready	9	Getting ready	22
How to do it...	10	How to do it...	22
How it works...	12	How it works...	25
There's more...	13	There's more...	26
See also	14	See also	26

## 2

### Engineering Useful Prompts

---

<b>Technical requirements</b>	<b>28</b>	<b>Improving output with chain-of-thought prompts</b>	<b>43</b>
<b>Using system messages</b>	<b>28</b>	Getting ready	44
Getting ready	29	How to do it...	44
How to do it...	29	How it works...	46
How it works...	30	There's more...	46
There's more...	31	See also	47
<b>Using zero-shot prompts</b>	<b>32</b>	<b>Adding clear syntactical markers</b>	<b>47</b>
Getting ready	32	Getting ready	47
How to do it...	33	How to do it...	48
How it works...	35	How it works...	48
There's more...	35	There's more...	49
See also	36	See also	49
<b>Avoiding hallucinations</b>	<b>36</b>	<b>Creating machine-readable output</b>	<b>50</b>
Getting ready	36	Getting ready	50
How to do it...	36	How to do it...	50
How it works...	38	How it works...	51
There's more...	38	There's more...	51
See also	38	See also	52
<b>Grounding prompts with data</b>	<b>39</b>	<b>Crafting non-chat prompts</b>	<b>52</b>
Getting ready	39	Getting ready	52
How to do it...	39	How to do it...	53
How it works...	40	How it works...	54
There's more...	41	There's more...	54
See also	41	See also	55
<b>Improving prompts with few-shot learning</b>	<b>41</b>	<b>Altering advanced parameters</b>	<b>55</b>
Getting ready	41	Getting ready	55
How to do it...	42	How to do it...	55
How it works...	42	How it works...	56
There's more...	43	There's more...	56
See also	43	See also	57

---

**3****Coding Solutions with the OpenAI Python SDK**

---

Technical requirements	59	How to do it...	65
Connecting Azure OpenAI Service to your Python environment	59	How it works...	67
Getting ready	59	There's more...	68
See also	59	See also	70
How to do it...	60	Using function calls with Azure OpenAI Service	71
How it works...	63	Getting ready	71
There's more...	63	How to do it...	72
See also	64	How it works...	75
Creating a Python application with Azure OpenAI Service	65	See also	76
Getting ready	65		

**4****Chatting with Your Own Data Using RAG**

---

Technical requirements	78	How to do it...	94
Getting started with Azure AI Search	78	How it works...	98
Getting ready	78	There's more...	98
How to do it...	78	Creating vector embeddings with the Azure OpenAI SDK	98
How it works...	82	Getting ready	98
Creating vector embeddings with Azure OpenAI Studio	83	How to do it...	98
Getting ready	83	How it works...	101
How to do it...	83	Chatting with your own data using the Azure OpenAI SDK	102
How it works...	93	Getting ready	102
Chatting with your own data using Azure OpenAI Studio	94	How to do it...	102
Getting ready	94	How it works...	105

## 5

### Fine-Tuning LLMs for Your Use Case

---

<b>Technical requirements</b>	<b>107</b>	Getting ready	120
<b>Preparing your dataset</b>	<b>107</b>	How to do it...	121
Getting ready	107	How it works...	123
How to do it...	108	<b>Testing your fine-tuned model with</b>	
How it works...	109	<b>Azure OpenAI Studio</b>	<b>124</b>
There's more...	109	Getting ready	124
How to do it...		How to do it...	124
How it works...		How it works...	127
There's more...		There's more...	127
<b>Fine-tuning your model with Azure</b>			
<b>OpenAI Studio</b>	<b>109</b>		
Getting ready	109	<b>Testing your fine-tuned model with</b>	
How to do it...	110	<b>the Azure SDK</b>	<b>128</b>
How it works...	120	Getting ready	128
There's more...	120	How to do it...	128
<b>Fine-tuning your model with the</b>			
<b>Azure SDK</b>	<b>120</b>	How it works...	129

## 6

### Building AI Solutions with Prompt Flow

---

<b>Technical requirements</b>	<b>130</b>	How to do it...	135
<b>Connecting Azure AI Studio to</b>		How it works...	139
<b>Azure OpenAI Service</b>	<b>131</b>	There's more...	141
Getting ready	131	See also	143
How to do it...	131	<b>Building generative AI applications</b>	<b>144</b>
How it works...	133	Getting ready	144
There's more...	133	How to do it...	144
See also	134	How it works...	149
<b>Using sample flows to build an AI</b>		There's more...	151
<b>application</b>	<b>134</b>	See also	154
Getting ready	134		

### Index

---

### Other Books You May Enjoy

---

# Preface

Azure OpenAI is a cloud-based platform for building and deploying AI models. It is the result of a collaboration between Microsoft Azure and OpenAI. This book is a practical guide for building intelligent applications using the Azure OpenAI platform. Written by specialists in the field, this book provides step-by-step instructions, real-world examples, and expert tips and tricks for leveraging the power of Azure OpenAI to build intelligent systems that can solve complex business problems.

From the basics of Azure OpenAI to building **generative AI (GenAI)** applications in a streamlined and low-code manner, this book has everything you need to develop AI applications that can transform your business. You'll learn about key concepts, such as deep learning, the different types of GenAI models, and AI use cases. You will also gain expert guidance in data preparation and prompt engineering as well as model deployment, fine-tuning, and testing.

## Who this book is for

This book is for developers, data scientists, and IT professionals who are looking to advance their practical knowledge of GenAI and Azure OpenAI. This book follows a recipe-based approach to building AI applications with Azure OpenAI Service, Azure OpenAI Python SDK, Azure AI Search, and Azure AI Studio. Working knowledge of Python is required for some of the recipes in this book. Familiarity with the basics of Azure is preferred but not necessary.

## What this book covers

*Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*, provides an introduction to GenAI and Azure OpenAI Service. You'll explore the fundamentals of GenAI models and their different types and use cases. You will become fully acquainted with the basics of Azure OpenAI Service and use it to build basic chatbots.

*Chapter 2, Engineering Useful Prompts*, gives an overview of crafting expert-level prompts. You will be introduced to increasingly complex prompt engineering patterns and will be able to develop prompts that consistently produce quality content.

*Chapter 3, Coding Solutions with the OpenAI Python SDK*, shows how to integrate Azure OpenAI Service into your Python code and build powerful real-world AI applications with the Azure OpenAI SDK. You will learn about function calls and be able to process LLM output into actionable insights.

*Chapter 4, Chatting with Your Own Data Using RAG*, demonstrates how to chat with your own data using **retrieval-augmented generation (RAG)**. You'll learn how to generate meaningful responses from your data by utilizing Azure OpenAI Service and Azure AI Search.

*Chapter 5, Fine-Tuning LLMs for Your Use Case*, delves into tailoring LLMs for your specific use case. You will identify suitable data for your model and use Azure OpenAI Studio and the Azure SDK for fine-tuning and testing your model to make it more efficient, reliable, and tailored to your specific requirements.

*Chapter 6, Building AI Solutions with Prompt Flow*, focuses on using prompt flow within Azure AI Studio to build LLM-based applications from scratch. You'll learn how to connect Azure OpenAI Service to Azure AI Studio, use prompt flow to adapt sample flows to your own projects, and evaluate your application using a variety of metrics.

## To get the most out of this book

You will require a working knowledge of Python for some recipes in this book. Familiarity with the basics of Azure will prove useful but is not necessary.

Software/hardware covered in the book	Operating system requirements
Jupyter Notebook	Windows, macOS, or Linux
Microsoft Azure	Windows, macOS, or Linux
Python 3.8 +	Windows, macOS, or Linux
Visual Studio Code	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

## Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “It’s crucial to note that the `deployment` variable, which you will use to initialize the client, should be set to the name of your model deployment.”

A block of code is set as follows:

```
from openai import AzureOpenAI
client = AzureOpenAI(
    azure_endpoint = openai_api_url,
    api_key= openai_api_key,
    api_version="2024-02-01"
)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
az group create --name YourResourceGroupName --location
YourRegion
```

Any command-line input or output is written as follows:

```
pip3 install openai

pip3 install pandas
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: “Select **Python 3** under **Notebook** to create and open a Python notebook.”

**Tips or important notes**

Appear like this.

## Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

## Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

## How to do it...

This section contains the steps required to follow the recipe.

## How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

## There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

## See also

This section provides helpful links to other useful information for the recipe.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, email us at [customercare@packtpub.com](mailto:customercare@packtpub.com) and mention the book title in the subject of your message.

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata) and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

# 1

## Introducing Azure OpenAI and Building Your First Chatbot

While there have been many breakthroughs in AI, from self-driving cars to social media recommendation algorithms and automated machine learning, the biggest breakthrough yet has been commercially viable generative AI. **Generative AI (GenAI)** refers to machine learning models that produce text, images, videos, code, sound, or other similar types of content.

Copywriters are using GenAI to write blogs, marketing copy, and social media posts. Engineers are using it to write, comment, fix, and understand code. Business leaders are using it to source new ideas, get constructive feedback, and identify new opportunities for investment and expansion. AI is quickly creeping into every aspect of the modern workplace. Yet it's not without its perils.

Perhaps the key problem with GenAI is the difference between perception and reality. Over the years, science fiction and media have influenced public perception of AI: *it's smarter than people, its responses are perfect, and it never lies.* Unfortunately, that is not GenAI. Real AI is error-prone, presents lies as facts, and requires a careful understanding of how to interact with it, and therein lies the key value of this book.

This chapter introduces GenAI. Following that, you will use the **Microsoft Azure portal** to create an **Azure OpenAI Service** resource. This will allow you to deploy and use AI models within the service. Finally, you'll end the chapter by building and deploying your first chatbot to a web app.

### Note

The Azure documentation refers to Azure OpenAI Service as both *Azure OpenAI* and *Azure OpenAI Service* interchangeably. In the Azure portal, however, the resource is called *Azure OpenAI*.

By the end of this chapter, you will be fully acquainted with the basics of Azure OpenAI Service and use it to build basic chatbots. You will also have built a foundation for working on the recipes covered later in this book.

This chapter will cover the following topics and recipes:

- Introducing generative AI
- Understanding AI use cases
- Getting started with Azure OpenAI Service
- Deploying models to Azure OpenAI
- Using models in the Playground
- Building your first chatbot

## Technical requirements

To follow along with the recipes in this chapter, you will need to have the following:

- A Microsoft Azure account.
- Permission to create resources within your Azure subscription.
- Permission to create an Azure OpenAI Service resource from Microsoft.

To obtain permission from Microsoft, fill out this customer request form:  
[https://customervoice.microsoft.com/Pages/ResponsePage.aspx?id=v4j5cvGGr0GRqy180BhbR7en2Ais5pxKtso\\_Pz4b1\\_xUNTZBNzRKN1VQSFhZMU9aV09EVzYxWFdORCQlQCN0PWcu](https://customervoice.microsoft.com/Pages/ResponsePage.aspx?id=v4j5cvGGr0GRqy180BhbR7en2Ais5pxKtso_Pz4b1_xUNTZBNzRKN1VQSFhZMU9aV09EVzYxWFdORCQlQCN0PWcu).

While filling out this form, you will have to provide your name, Azure subscription ID, and company details, such as the company name, email address, physical address, website, and phone number. Make sure you have these ready before you apply.

- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter1>. The Chapter 1 folder contains all the notebooks for this chapter.

## Introducing generative AI

GenAI is an umbrella term that encompasses many disparate topics and use cases. It's easy to become confused about what counts as AI and what does not. Thus, it's helpful to get familiar with the different model types, proprietary AI models, and companies driving GenAI.

This section will introduce the different types of GenAI models. Once you have a good understanding of these broad categories, you'll learn about the most popular models within each type and which companies produce them. You'll also get to know about the other important players in the AI revolution.

## The many models behind generative AI

You can simplify GenAI for yourself by mapping each type to a specific use case. Broadly speaking, GenAI is used to create text, code, images, audio, video, music, and speech. It can even be used in niche use cases, such as discovering molecules.

All GenAI models share one thing in common: they are based on deep learning. **Deep learning** is a subset of machine learning that implements neural networks with multiple layers (hence the term *deep*) to learn from data. **Neural networks** are a type of machine learning inspired by the structure of the human brain. They are composed of a series of nodes, or *neurons*, arranged in layers: the input layer, hidden layers, and the output layer.

The input layer is where the network receives its information in the form of data vectors, similar to pixels for images or word embeddings for language processing. After data is fed into the input layer, it is then processed in hidden layers by neurons that apply various transformations to the information. Each neuron in a layer receives input from every neuron in the previous layer, performs a computation, and passes the result to each neuron in the next layer. This happens successively from the input layer to the output layer. Through this method, GenAI models learn to produce text, images, and speech.

Here are the different types of models that have gained prominence in the last few years:

- **Large language models (LLMs)** generate text, including code, emails, stories, poems, book chapters, and so on. In response to a **prompt**, which is input text provided by a user, LLMs predict which **token** comes next and outputs that token. A token is equal to roughly three-quarters of a word. This process happens one token at a time until a complete response is formed and is based entirely on probability. LLMs are called *large* because they are trained on unimaginably vast datasets, often containing hundreds of billions of parameters. **Parameters** is a term for the weights and biases within the LLM's neural network that determine how neurons interact with each other. The more parameters, the more capable the model is in handling complex tasks.
- **Generative adversarial networks (GANs)** are used to generate images, including faces, paintings, and photorealistic images. GANs are called adversarial because they employ two neural networks: a *generator* that tries to produce convincing data, and a *discriminator* that tries to distinguish real data from fake data. Through this process, GANs can produce increasingly realistic images capable of tricking even human beings.
- **Diffusion models** start with random noise and gradually refine it through a process that mimics diffusion. They are most commonly used to generate images and audio clips, including music and speech. The process begins with what you might consider television static—just random noise with no discernible structure. Then, a series of steps begins, and in each step, the model makes slight modifications to this noise until it resembles a real image.
- **Acoustic models** are used primarily in automatic speech recognition. These models are designed to convert fundamental speech units into written text by recognizing patterns in speech and sounds. They accomplish this by analyzing the spectral features of speech, including pitch, tone, and articulation.

Now that you're familiar with some of the model types, let's dive into the models themselves.

## Proprietary AI models and the companies behind them

Perhaps the most famous GenAI company is **OpenAI**. OpenAI was founded in 2015 by an array of company scientists with Sam Altman and Elon Musk as board members. They're responsible for making GPT, DALL-E, and Whisper, and made headlines by releasing ChatGPT on November 30, 2022.

**Generative pre-trained transformers (GPTs)** are a type of AI model that uses deep learning techniques to generate human-like text. GPTs employ transformer architecture, which processes data using self-attention mechanisms to assess the significance of each part of an input or output sequence in context. GPTs are the world's most widely used type of LLM. OpenAI has made multiple well-known proprietary GPT models, with GPT-3.5 Turbo being the fastest and GPT-4 being the most powerful. DALL-E 2 and DALL-E 3 are models that generate images, while Whisper is an acoustic model used for speech transcription.

Another important player in the AI space is **Meta**, formerly Facebook. Meta publicly released three versions of their own LLM, Llama 2. Llama 2 is available in three sizes: 7 billion, 13 billion, and 70 billion parameters. Parameters refer to the adjustable elements of the neural network that the model learns during training to determine the output. Models with more parameters can follow more nuanced instructions and generate more complex text. While Llama 2 with 70 billion parameters is the most powerful Llama model, it also requires powerful **graphics processing units (GPUs)** to run.

**Anthropic** is an AI safety and research company that has introduced Claude 3. Claude 3 is famous for being able to take in very large text documents as input—up to around 150,000 words—as well as for refusing to engage in conversations about controversial subject matter such as bomb-making.

**Google** is, unsurprisingly, another important company in the AI space. They have their own LLM, Gemini. This LLM has its own public-facing chatbot and enables Gmail users to craft emails using AI. Google has integrated Gemini into over 25 of its services, cementing its position as a major player in the AI era.

## Other important players behind the AI revolution

Besides OpenAI, Meta, Google, and companies that produce GenAI models for public and enterprise use, organizations have contributed to AI in different ways. Two prominent players include NVIDIA and Microsoft.

AI uses GPUs for training and processing and, according to Yahoo Finance\*, NVIDIA GPUs make up between 80–95% of the global market supply. Without NVIDIA, the AI revolution would not have been possible.

---

\* <https://finance.yahoo.com/news/nvidia-is-the-ai-king-but-threats-to-its-reign-abound-144635363.html>

Microsoft has invested \$10 billion\* in OpenAI and heavily integrated the GPT and DALL-E models from OpenAI into its Azure cloud platform. Through Microsoft Fabric, Microsoft has enabled GenAI across a wide range of its services, from Microsoft Power BI to Microsoft Word, Microsoft Excel, Microsoft Outlook, and Microsoft Teams.

**Note**

There are many other players in AI beyond those listed in this book—too many to list. Furthermore, new players join the fray every day. Try subscribing to an AI newsletter to keep up with the latest developments.

Now that you have an overview of the various GenAI models, let's get into the use cases that make GenAI so exciting.

## Understanding AI use cases

Some people accuse AI of being a toy, overhyped, or a flash-in-the-pan technology—they're wrong. Companies are adopting AI and integrating it into their everyday processes. By and large, AI use cases can be divided into three main types: text-based, image-based, and speech-based.

**Note**

Other use cases exist, but they tend to be found in more niche areas. These include video-based use cases (AI news anchors are becoming common), non-speech audio-based use cases, such as making music, and molecular discovery use cases, which are common in the chemical industry. These use cases are beyond the scope of this book.

### Understanding AI use cases for text

Text-based AI has diverse applications across various industries. This section provides an overview of some of the most prevalent use cases:

- **Service chatbots** are conversational AI applications with which internal employees or external customers can interact. An internal chatbot, such as an IT support bot, uses LLMs to scan IT documentation and returns answers to the end user. An external chatbot, such as one found on travel websites, lets users ask the LLM for vacation ideas, recommendations, and ongoing deals. Chatbots like these have existed for a long time, but AI greatly increases their accuracy as well as their ability to have natural conversations.

---

\* <https://www.bloomberg.com/news/articles/2023-01-23/microsoft-makes-multibillion-dollar-investment-in-openai>

- **Chat-with-your-own-data** use cases are those in which AI is used to answer questions regarding a corpus of documents. It returns not only answers but also links to the relevant documentation. Common chat-with-your-own-data use cases involve extracting numbers from financial documents, summarizing results from research papers, and looking up internal guidelines or company information.

**Note**

In *Chapter 4, Chatting with Your Own Data Using RAG*, you will learn which Microsoft Azure resources are necessary to be able to search over and chat with your documents.

- **Document drafting** is becoming an increasingly popular use case. Here, AI is given information and told to write a news article, blog post, or other creative content that is then presented to the editor. AI turns writers into editors. It saves time for editors by producing grammatically correct content, which can then be checked against the source material to ensure it's hallucination-free. **Hallucinations** refer to the tendency of LLMs to produce incorrect information and present it as a fact.

**Note**

In *Chapter 2, Engineering Useful Prompts*, you will learn how to prevent hallucinations and produce error-free rewrites of news articles.

- **Automated content creation** is similar to document drafting but lacks the end editor. It should only be used when accuracy is not a high priority. Examples include parody news websites, social media posts whose primary purpose is engagement, and clickbait content.
- **Content summarization and analysis** is another popular use case, especially for product reviews. GenAI is particularly good at summarizing a large number of documents succinctly and providing qualitative analysis. However, one should be careful, as GenAI is poor at quantitative analysis. LLMs have difficulties even with simple quantitative tasks, such as counting the number of words in a sentence or the number of sentences in a text.
- **Translation** is one area in which GenAI excels. Not only can models translate from one language to another, but they can follow additional instructions, such as `sound as natural as possible in the target language` or `translate as literally as possible from the original language`. In the future, the majority of translation tasks will probably be performed by GenAI.
- **Personalizing content for sales outreach** is an email-based use case where messages are transformed based on end-user information to ensure maximum personalization. In the past, emails could only be modified based on demographic information. Now, AI can use past order history, demographic information, past email interactions, and the user's writing style to craft personalized content that increases the likelihood of a response.

Let's now move to the next most popular category of GenAI use cases—image generation.

## Understanding AI use cases for image generation

The second most popular category of AI use cases is image generation. The most advanced models have been able to overcome significant hurdles to adoption, most notably, the inability to produce copyrighted material accurately and the tendency to create images that fall into the uncanny valley.

**Uncanny valley** refers to the eerie or unsettling feeling people experience when looking at figures that look and move almost, but not exactly, like humans. Older GenAI models often generated human images with noticeable imperfections, particularly in the depiction of hands and teeth. These AI-generated images often had too many or too few fingers when they were holding objects and too many teeth when they were smiling.

### Note

The most advanced diffusion model, Dall-E 3, has seen significant improvement in producing human images with the correct number of teeth and fingers. Midjourney v6, released in December 2023, likewise has seen significant improvements.

As image-generation technology advances, the number of commercial use cases increases. Most of these use cases avoid the uncanny valley problem by not using AI to directly create images of human beings. Three of the most popular use cases include stock image creation, graphic design prototyping, and e-commerce:

- **Stock image creation** uses GenAI—mostly diffusion models—to create stock photos. Stock photography refers to high-quality images that fit a wide array of thematic requirements, such as nature, people at work, technology, and cities. Usually, companies can license stock photos for a small fee, but GenAI takes this one step further by allowing companies to create their own stock photos for a nominal fee.

It's important to recognize that advances in AI are affecting professions such as photography, copywriting, and translation, raising ethical concerns. AI models are typically trained on data found in the public domain on the internet, including content created by artists, writers, and photographers. This data is now being used to develop AI models that may replace these same individuals in their professions. While these questions are outside the scope of this book, it is important to acknowledge they exist.

- **Graphic design prototyping** refers to using GenAI to create first drafts of images, often for marketing and advertising. GenAI can produce drafts rapidly, so artists can present a wide variety of designs to internal and external stakeholders quickly. Once a sample is approved, they can either touch up the AI-generated image or use the AI content as inspiration to design from scratch.

- **E-commerce** is a rather straightforward use case for GenAI. Online retailers use GenAI models to generate images of their products in various settings or styles based on textual descriptions, significantly reducing costs that would otherwise be spent on photography. GenAI in e-commerce is also used to visualize customizable products and create seasonal or holiday backgrounds for existing products.

Images and text may be the most popular GenAI use cases, but another application of GenAI is beginning to gain popularity: speech.

## Understanding AI use cases for speech

Speech use cases for GenAI have evolved tremendously over the last few years. Users have been talking to virtual assistants, such as Amazon's Alexa, since 2013. Virtual assistants are software agents that perform tasks or services for an individual based on commands or questions. These tools use **natural language processing (NLP)** to interact with users, learn from these interactions, and improve their ability to help people. As LLM capabilities have advanced, so have the capabilities of these agents.

Three other popular use cases for speech include transcription, translation, and analysis. **Transcription** is taking an audio file and converting it to text. **Translation** is taking that same audio file and converting it to text in a different language. **Analysis** refers to either analyzing the sentiment of the audio file or summarizing its contents in addition to a full transcription.

### Note

Different models are good at translating into different languages. While many are proficient in translating English, some models may struggle when translating into other languages.

Now that you're familiar with text-, image-, and speech-based AI use cases, let's begin your implementation journey with Microsoft Azure OpenAI Service. In this chapter, you're going to follow *Figure 1.1* to build an LLM-based chatbot. You will first build an Azure OpenAI resource, then deploy GPT models, test prompts with the models, and finally deploy a chatbot to the web.

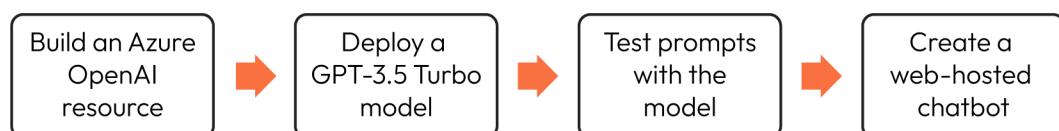


Figure 1.1: Process of building an LLM-based chatbot

## Getting started with Azure OpenAI Service

**Azure OpenAI Service** is among the many AI services offered by Microsoft Azure. Getting started with Azure OpenAI Service is simple and can be accomplished via either the Azure portal or the **Azure Command-Line Interface (Azure CLI)**. Whether you wish to integrate OpenAI models such as GPT-4 into your code or deploy a web-based GPT chatbot, your first step in Azure will always be creating an Azure OpenAI resource.

### Note

Azure OpenAI Service is not the same as **Azure AI Studio**. Azure AI Studio is a new service that lets you utilize open source LLM models from Hugging Face, Meta, Mistral AI, and others. Azure OpenAI Service has its own graphical user interface called the **Azure OpenAI Studio**.

## Getting ready

To create an Azure OpenAI resource, you should first have a Microsoft Azure account and permission to create resources within your subscription. If you wish to create an Azure OpenAI resource via code, you will also need to install the Azure CLI.

Unlike most Azure resources, you must first fill out a customer request form to obtain access to Azure OpenAI at the following link: [https://customervoice.microsoft.com/Pages/ResponsePage.aspx?id=v4j5cvGGr0GRqy180BHbR7en2Ais5pxKtso\\_Pz4b1\\_xUNTZBNzRKN1VQSFhZMU9aV09EVzYxWFdORCQ1QCN0PWcu](https://customervoice.microsoft.com/Pages/ResponsePage.aspx?id=v4j5cvGGr0GRqy180BHbR7en2Ais5pxKtso_Pz4b1_xUNTZBNzRKN1VQSFhZMU9aV09EVzYxWFdORCQ1QCN0PWcu).

### Note

Due to high demand, ongoing product improvements, and the commitment to responsible AI at Microsoft, Azure OpenAI is currently only open to existing Microsoft Azure customers or customers with low-risk use cases. More information is available at this link: <https://learn.microsoft.com/azure/ai-services/openai/overview#how-to-get-access-to-azure-openai>.

While filling out this form, you will have to provide your name, Azure Subscription ID, and company details such as the company name, email address, physical address, website, and phone number. Make sure you have these ready before you apply. Optionally, you can also provide the name and email address of your Microsoft contact, if you have one.

## How to do it...

To create an Azure OpenAI resource, follow the steps below:

1. Log in to the Azure portal (<https://portal.azure.com>).
2. Click **Create a resource**—the plus button, located in the top-right corner of your screen under **Azure services**.
3. Search for Azure OpenAI in the search bar.
4. Click **Azure OpenAI** in the dropdown. It will be the first item on the next screen.
5. Click **Create**. This will bring up the Azure OpenAI resource creation page.
6. Fill out the **Basics** tab:
  - Select your Azure subscription.
  - Click **Create new for Resource group**, give it a name, and click **OK**. Azure resource groups are used to manage and group related resources that share the same life cycle, permissions, and policies.
  - Select the Azure region closest to you. Azure regions determine which data centers you use and will affect your latency.

### Note

A later recipe in this chapter will use the Assistants feature of Azure OpenAI. It's currently in public preview, and only available in the **australiaeast**, **eastus**, **eastus2**, **francecentral**, **norwayeast**, **swedencentral**, and **uksouth** Azure regions. Please select an Azure region from that list.

- Give a name to your Azure OpenAI resource. Your resource name can only include alphanumeric characters and hyphens, and it cannot start or end with a hyphen.
- Select a pricing tier. Currently, there is only one: **Standard S0**. At this point, your screen should resemble *Figure 1.2*:

1 Basics    2 Network    3 Tags    4 Review + submit

Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation.

[Learn more](#)

**Project Details**

Subscription \* ⓘ Your Test Account Subscription

Resource group \* ⓘ (New) your-resource-group [Create new](#)

**Instance Details**

Region ⓘ East US

Name \* ⓘ my-azure-openai-1234569

Pricing tier \* ⓘ Standard S0

[View full pricing details](#)

**Content review policy**

[Previous](#) [Next](#)

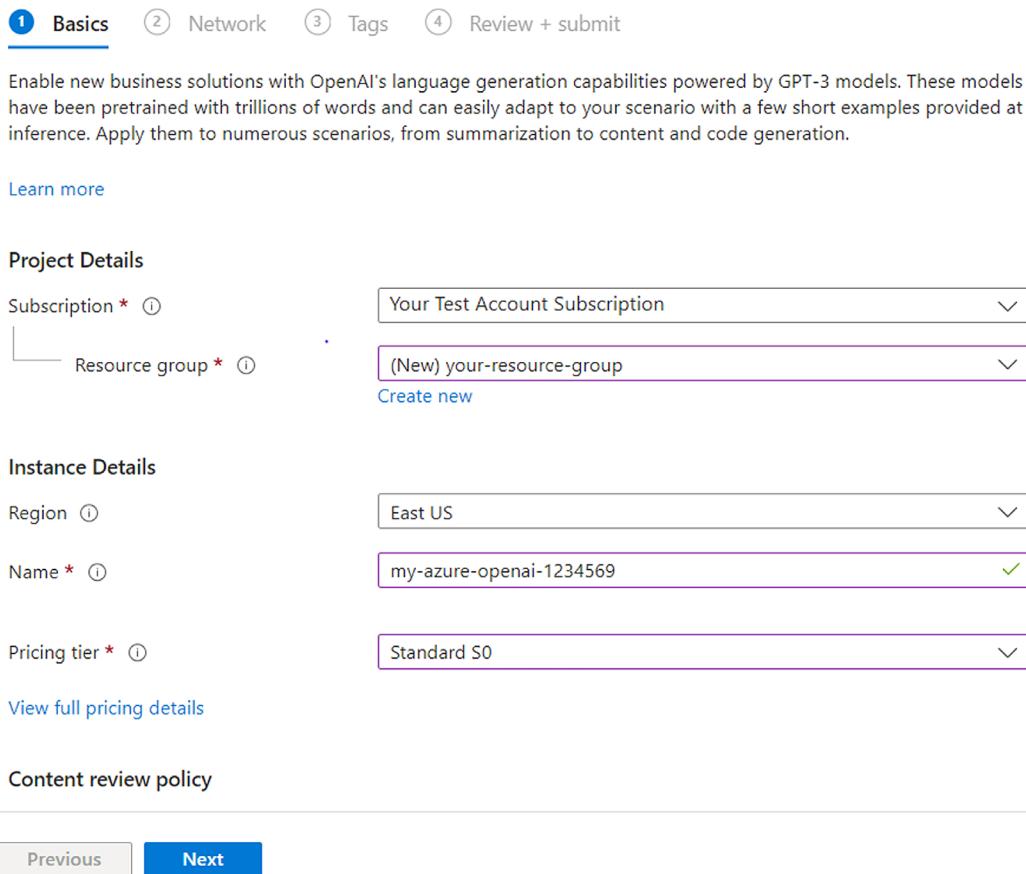


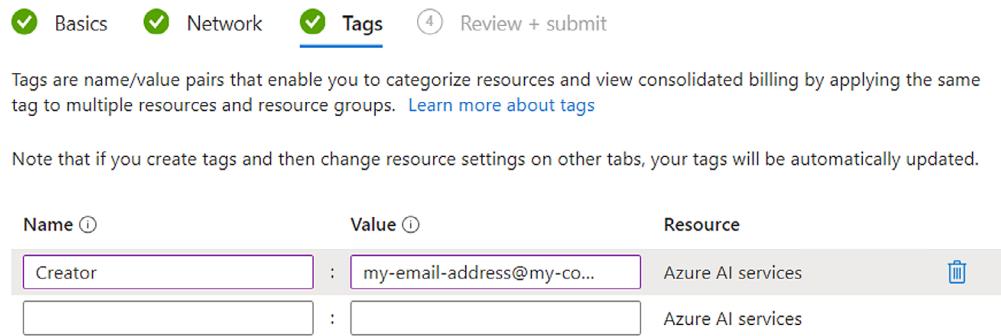
Figure 1.2: Azure OpenAI resource creation page

7. Click **Next** to advance to the **Network** tab.
8. On the **Network** tab, select **All networks, including the internet, can access this resource** for networking and click **Next** to advance to the **Tags** tab.

**Note**

For production use cases, you should restrict access to your Azure OpenAI Service to selected virtual networks only. **Virtual networks (VNets)** are logically isolated networks on cloud platforms that allow users to connect cloud resources and configure network settings such as IP addresses, subnets, and gateways.

9. Under the **Tags** tab, add any tags you wish. Azure tags are key-value pairs that you can assign to resources and resource groups in Microsoft Azure. Under **Name**, enter **Creator**. Under **Value**, enter your email address:



The screenshot shows the 'Tags' tab selected in the top navigation bar, which also includes 'Basics', 'Network', and 'Review + submit'. Below the tabs, a note states: 'Tags are name/value pairs that enable you to categorize resources and view consolidated billing by applying the same tag to multiple resources and resource groups.' A link to 'Learn more about tags' is provided. A note below says: 'Note that if you create tags and then change resource settings on other tabs, your tags will be automatically updated.' The main area displays a table with two rows of tags. The first row has 'Creator' in the Name column and 'my-email-address@my-co...' in the Value column, both highlighted with a purple border. The Resource column shows 'Azure AI services' and a delete icon. The second row is partially visible with an empty Name field and an empty Value field, also with a delete icon.

Name ⓘ	Value ⓘ	Resource
Creator	: my-email-address@my-co...	Azure AI services
	:	Azure AI services

Figure 1.3: Create Azure OpenAI tags tab

10. Click **Next** to advance to the **Review + submit** tab.

11. Click **Create**.

You have now created an Azure OpenAI resource in the Azure portal.

## How it works...

While creating Azure resources such as Azure OpenAI is a straightforward process, there are a few caveats to keep in mind:

- Resources that you create together should be placed in the same Azure region to reduce latency and costs. Servers running these services will be in proximity if your resources are in the same region.
- Use tags to track and control costs. Other than adding the creator, add tags indicating the project, cost group, and any collaborators who may be working with you on the project.
- Not all regions have access to all the OpenAI models, particularly the preview models. Read the Azure documentation to find out which models are available in your region and learn more about their rollout plan.

## There's more...

While most users utilize the Azure portal to create Azure resources, you can also use the Azure CLI using the following steps:

1. Install the Azure CLI, if you haven't already, by following the instructions in the Azure documentation. You can find a link to the documentation in the *See also* section of this recipe.
2. Click the Cloud Shell icon in the top-right corner of the Azure portal. This will open the **Azure Cloud Shell** command prompt window.

### Note

Azure Cloud Shell is an interactive, browser-accessible terminal where you can enter commands to create and manage Azure resources.

3. From Azure Cloud Shell, log in to the Azure CLI with the following command:

```
az login
```

4. Create a new resource group with the following command, noting that `YourResourceGroupName` and `YourRegion` are placeholders for your resource group and Microsoft region name:

```
az group create --name YourResourceGroupName --location  
YourRegion
```

5. Create an Azure OpenAI resource with the following command, noting that `YourOpenAIResourceName`, `YourResourceGroupName`, `YourRegion`, and `YourSubscriptionID` are placeholders for your OpenAI resource name, resource group, Microsoft region name, and Azure subscription ID respectively:

```
az cognitiveservices account create --name  
YourOpenAIResourceName --resource-group YourResourceGroupName  
--location YourRegion --kind OpenAI --sku s0  
--subscription YourSubscriptionID
```

With these steps, you have created an OpenAI resource using the Azure CLI.

## See also

Here's a list of links to Azure documentation on creating an Azure OpenAI resource using both the Azure portal and the Azure CLI:

- Install the Azure CLI: <https://learn.microsoft.com/cli/azure/install-azure-cli>
- Create and deploy an Azure OpenAI resource to the Azure portal: <https://learn.microsoft.com/azure/ai-services/openai/how-to/create-resource?pivots=web-portal>
- Create and deploy an Azure OpenAI resource to the Azure CLI: <https://learn.microsoft.com/azure/ai-services/openai/how-to/create-resource?pivots=cli>

## Deploying models to Azure OpenAI

Now that you have created an Azure OpenAI resource, you need to deploy OpenAI models to start using LLMs and building chatbots. The models you will use in this recipe are GPT-3.5 and GPT-4.

### Getting ready

To deploy models, you need to have an Azure account and have created an Azure OpenAI resource. Additionally, you're going to need an available quota, which depends on the model version and deployment type. **Quota** for Azure OpenAI models refers to how many tokens the LLM can process per minute.

Each deployed instance of a model shares quota with models of the same type within the same region. Thus, all your GPT-3.5 models within the same region pull from the same quota pool but do not share a quota pool with GPT-4 models or GPT-3.5 models from a different region.

OpenAI models charge per input and output token, so you should also keep track of spending. As of writing this book, GPT-3.5 Turbo models cost \$0.01 per 1,000 input tokens and \$0.03 per 1,000 output tokens, while GPT-4 models cost \$0.03 per 1,000 input tokens and \$0.06 per 1,000 output tokens. It may seem counterintuitive that a service with a pay-as-you-go consumption model would also have a quota, but this is due to the very high demand Microsoft is encountering for this service.

#### Note

A token is equivalent to roughly three-quarters of a word.

If you are using a personal Azure account and this is your first deployment, you will have an available quota. Since the quota is determined by model type and Azure region, if you need an additional quota, you can always obtain it by creating a new Azure OpenAI resource in a different Azure region.

## How to do it...

To deploy models to Azure OpenAI, follow these steps:

1. Search for Azure OpenAI in the top search bar of the Azure portal.
2. Click on the Azure OpenAI resource you created.
3. Click **Go to Azure OpenAI Studio** in the top-right corner of your screen. **Azure OpenAI Studio** is a GUI-based platform for deploying OpenAI LLMs, using LLMs in a playground setting, and building and deploying web-based chatbots.

### Note

You will notice another option to build in Azure AI Studio. This is a different service from Azure OpenAI Studio, although they do share some capabilities. These two services may be combined in the future.

4. Click **Deployments** on the left side of your screen.
5. Click **Create new deployment** next to the plus button.
6. Under **Select a model**, select **gpt-35-turbo**.
7. Under **Deployment type**, select **Standard**.
8. Under **Model version**, select **1106**. The latest version at the time of writing this book is **0125**, but it is not widely available. Your screen should resemble *Figure 1.4*.

### Note

Do not use the option to auto-update to the default model version, especially for production use cases, as some of your prompts will probably break.

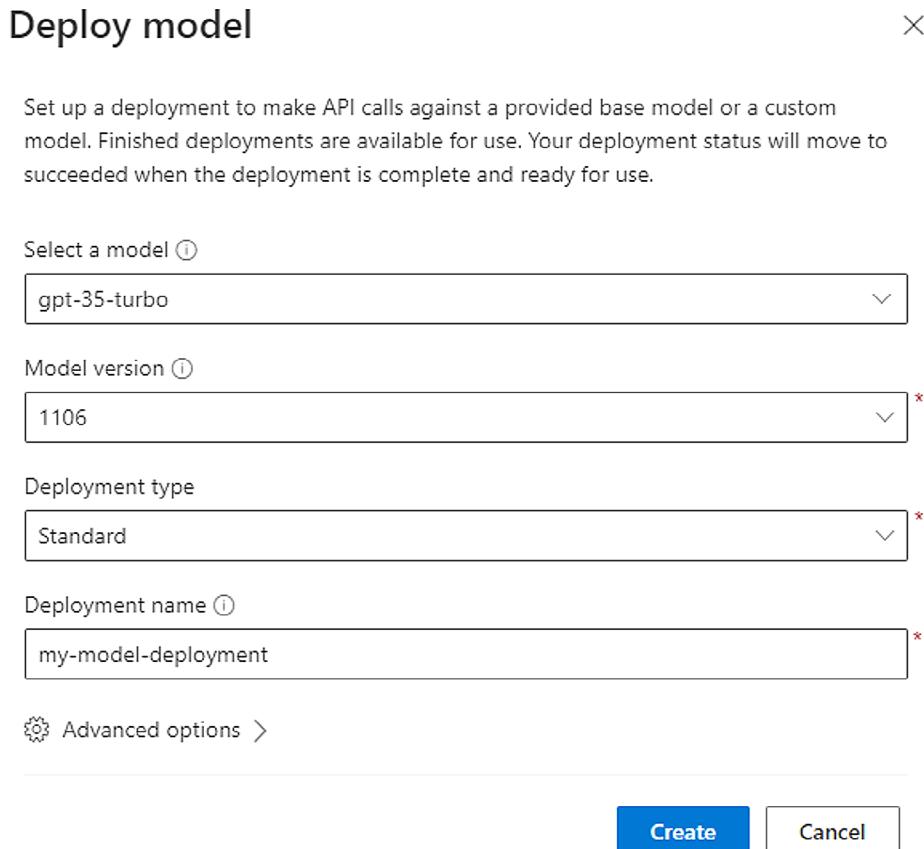


Figure 1.4: Deploying a model to Azure – basic settings

9. Give your model deployment a name. The name can be between 2 and 64 characters long, and can only include alphanumeric characters, dashes, or underscores. It cannot end with either a dash or an underscore.
10. Click **Advanced options**.
11. Leave **Content Filter** on its default option. **Content Filter** allows you to change your deployment settings to block or allow content on sensitive topics. To alter your content filter settings, you need to apply via your Microsoft representative.
12. Set your **Tokens per Minute Rate Limit** to **100k**. This limit determines how many words you can send to your deployment per minute as well as how much output it can produce a minute.
13. Enable **Dynamic Quota**. This enables you to push past your rate limit if capacity is available in your Azure region. Your screen should resemble *Figure 1.5*:

## Deploy model ×

Set up a deployment to make API calls against a provided base model or a custom model. Finished deployments are available for use. Your deployment status will move to succeeded when the deployment is complete and ready for use.

Select a model (1)

gpt-35-turbo

Model version (1)

1106 \*

Deployment type

Standard \*

Deployment name (1)

my-model-deployment \*

⚙️ Advanced options ▼

Content Filter (1)

Default

(1) 180K tokens per minute quota available for your deployment

Tokens per Minute Rate Limit (thousands) (1)

100K

Corresponding requests per minute (RPM) = 600

Enable Dynamic Quota (1)

Enabled

Create Cancel

Figure 1.5: Deploying a model to Azure – advanced settings

14. Click **Create**.

This step will deploy your model to Azure OpenAI.

## How it works...

When you deploy a model to your Azure OpenAI resource, you're making an endpoint that's available to be used both within the Azure OpenAI Studio and within code. Thus, you can call upon your model via an API in whichever computer language you choose.

Regarding which model to choose, **GPT-3.5 Turbo** is the fastest and cheapest, but it is the least performant. It also limits your prompt size to 4,096 tokens (roughly 3,000 words). **GPT-3.5 16k** is more expensive as it lets you put up to 16,384 tokens (roughly 12,000 words) in your prompt.

### Note

Quota is shared across model types within the same region. If you have three GPT-3.5 Turbo deployments with a rate limit of 100k per minute each and your quota is 300k per minute, you will not be able to deploy additional GPT-3.5 Turbo models within that region. To create other deployments, you either need to create an additional Azure OpenAI resource in another region or use a different model.

**GPT-4** is OpenAI's premiere model and allows you to create prompts up to 8,192 tokens (roughly 6,000 words). **GPT-4 32k** lets you input 32,768 tokens (roughly 24,000 words) and is the most expensive model you can choose. **GPT-4 Turbo** is in preview and lets you input up to 128,000 tokens (roughly 96,000 words). You'll find that while the GPT-4 models are the most performant, depending on the task, GPT-3.5 Turbo can often perform roughly the same.

### Note

Avoid using GPT-4 Turbo in production environments since it is currently in preview and subject to change.

## There's more...

In addition to deploying models via the Azure OpenAI Studio, you can also deploy models via **Azure Bicep templates**. Bicep templates are an abstraction over **Azure Resource Manager (ARM)** templates. Both are infrastructure-as-code services that allow you to deploy resources via code that you can save as opposed to a user interface. There's a link to a guide in the *See also* section of this recipe.

## See also

Here are links to documentation giving you more information about Azure OpenAI models and how to deploy them via Bicep templates:

- Azure OpenAI Service models documentation: <https://learn.microsoft.com/azure/ai-services/openai/concepts/models>

- Bicep documentation: <https://learn.microsoft.com/azure/azure-resource-manager/bicep/>
- “How to Deploy an Azure OpenAI Service Instance and Model with Bicep” from *Tim’s Azure OpenAI Training Blog*: <https://azureopenai.blog/2023/08/24/how-to-deploy-an-azure-openai-service-instance-and-model-with-bicep/>
- Azure OpenAI quota documentation: <https://learn.microsoft.com/azure/ai-services/openai/quotas-limits>

## Using models in the Playground

Now that you’ve deployed a model, it’s time for you to test it out in the Azure OpenAI Studio Playground. The **Azure OpenAI Studio Playground** is an environment that lets you create, edit, and even deploy chatbots to the web. In this section, you’re going to use your deployed model to create a simple assistant that comments on your Python code and formats it in the PEP 8 style. **Python Enhancement Proposal 8 (PEP 8)** is a style guide for writing Python code written by Guido van Rossum, Barry Warsaw, and Nick Coghlan to improve the readability and consistency of Python code.

### Getting ready

To test your deployed model, you need to have an Azure account, an Azure OpenAI resource in either the **australiaeast**, **eastus**, **eastus2**, **francecentral**, **norwayeast**, **swedencentral**, or **uksouth** Azure regions, and a model deployed to your Azure OpenAI Service.

#### Note

While code will be provided to you in this section, you can also bring your own Python code to comment and format. With a few slight modifications, you can also modify the prompts in this section to comment on code written in languages other than Python.

### How to do it...

To test your deployed model, perform the following steps:

1. Navigate to and open your Azure OpenAI resource.
2. Click **Go to Azure OpenAI Studio** in the top-right corner of your screen.
3. Click **Assistants** on the left side of your screen.
4. Under **Assistant name**, input `Code Commenter`.

5. Under **Instructions**, type the following:

You are a Python code commenting and formatting assistant. Your job is to ask users for Python snippets, add comments to code and format the code by changing all variable and column names to be lowercase with different words separated by underscores. Do not allow uppercase letters, camel case, or proper-case column names or variables. Always show the code after you have commented and formatted it.

6. Under **Deployment**, select the model that you deployed in the previous recipe.
7. Click **Save**. You have now created a simple code commenting assistant.
8. Have the following conversation with the chatbot:

Hello. May I have a sandwich?

It will output something like the following:

I see you might be looking for an actual sandwich, but just to clarify, I'm here to help with Python code comments and formatting.

9. Paste in the following code block and press *Enter*:

```
import pandas as pd
import matplotlib.pyplot as plt
data = {
    "toyType": ["Action Figure", "Puzzle", "Board Game", "Action Figure", "Puzzle", "Board Game"],
    "sales_AMOUNT": [100, 150, 200, 110, 120, 250],
    "DATE": ["2024-01-01", "2024-01-01", "2024-01-01", "2024-01-02", "2024-01-02", "2024-01-02"]
}
df = pd.DataFrame(data)
df['DATE'] = pd.to_datetime(df['DATE'])
sales_summary = df.groupby('DATE')['sales_AMOUNT'].sum().reset_index()
plt.figure(figsize=(10, 6))
plt.plot(sales_summary['DATE'], sales_summary['sales_AMOUNT'],
marker='o', linestyle='-', color='blue')
plt.title('Sales Over Time')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

10. You should now have perfectly commented code, and all the column names should be consistently formatted.

## How it works...

LLMs work by following instructions, and the instructions you provide need to be clear, repetitive, and exact. In this case, you not only instructed the model to comment and format the code, but also specified exactly *how* (lowercase with underscores) and *what* (variables and column names) you wanted to format. Moreover, you reinforced this by instructing it to forbid specific types of improperly formatted content (uppercase, camel case, or proper case). These are all techniques you will explore further in *Chapter 2, Engineering Useful Prompts*.

## There's more...

You can also use this assistant to run code. Try the following:

1. Add the following sentence to your prompt: If there is a graph, run the code and output the graph.
2. Switch **Code interpreter** on. This option enables your assistant to run code.
3. Click **Save**.
4. Paste your code once again to the assistant. You should see a line graph similar to *Figure 1.6*:

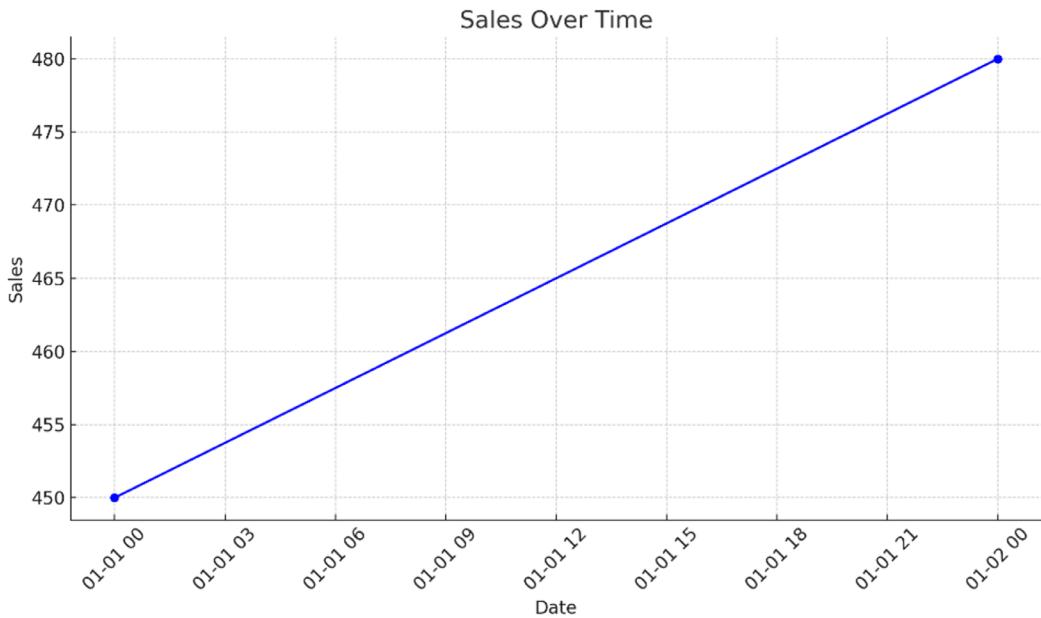


Figure 1.6: AI-generated line graph

## See also

You can find more information about using Azure OpenAI Assistants in the following Azure documentation: <https://learn.microsoft.com/azure/ai-services/openai/assistants-quickstart?tabs=command-line&pivots=programming-language-studio>.

## Building your first chatbot

Now that you've tried out your model by building a simple code commenting and formatting assistant, let's build an actual chatbot and deploy it to the web. Luckily, the Azure OpenAI Studio makes this process easy—you don't need knowledge of web application development to deliver a web-based chatbot to your business clients.

### Getting ready

You will need an Azure account and an Azure OpenAI Service resource with either GPT-3.5 or GPT-4 deployed. This section will focus on creating a translation bot, so it would help to have a few pieces of text available that you can translate from one language to another.

### How to do it...

1. Navigate to and open your Azure OpenAI resource.
2. Click **Go to Azure OpenAI Studio** in the top-right corner of your screen.
3. Click **Chat** on the left side of your screen.
4. Under **Deployment** in the configuration panel on the left side of your screen, select the model that you deployed in the previous exercise.
5. Set **Past messages included** on the slider to **6**. This setting determines how many prior messages the model considers in its response. Setting this to a low number will decrease the number of prior messages sent to your model's API endpoint.
6. Under **System message** in the setup panel on the right side of your screen, write the following:

You are an AI copywriting translation assistant. You will ask the user what language they would like their marketing text translated into.

#### Note

System messages are used to prime your model and provide it with context and instructions on how it should interact with end users. While the model can and will answer other questions, the system message determines the model's primary purpose.

7. Click **Apply changes**.
8. Next, test your chatbot by typing Hello. It should greet you and ask how it can be of assistance.
9. Tell the chatbot the following:

I'd like you to translate this text from English to Japanese.

10. Then, provide it with the following text to translate:

Introducing Super Yummy Flakes - the breakfast that transforms your morning into a spectacular adventure! Bursting with unbeatable flavor and packed with nutrients, Super Yummy Flakes are here to revolutionize your breakfast experience. Each bowl is a treasure trove of whole grain goodness, designed to keep you energized throughout the day.

Available now at a store near you. Grab a box of Super Yummy Flakes and make every breakfast a super adventure!

#### Note

GPT 3.5 Turbo performs worse at most tasks compared to GPT-4. When building chatbots, it's important to account for both model performance and what level of performance is acceptable to your clientele.

11. You should now have this marketing material translated into Japanese. Try it with other languages and examples to see how it performs.
12. To deploy the chatbot to web, click the **Deploy to** button in the top-right corner of your screen.
13. Under **Name**, input Copy-Translation-Bot. Site names allow only alphanumeric characters and hyphens. They cannot start or end in a hyphen and must be less than 64 characters.
14. Select your **Subscription**, **Resource group**, and **Location** (Azure region) from the three dropdown lists.
15. Select **Free (F1)** under **Pricing plan**. For actual production deployments, select a larger compute that is capable of scaling to handle more users and a higher workload.

#### Note

As noted on the deployment screen, web apps will incur usage on your account when you select tiers other than the **Free** plan.

16. Do not check the box to enable chat history in the web app. Checking this box will require you to create a **Cosmos DB** resource to store historical chats.

**Note**

Cosmos DB is a globally distributed, multi-model database for semi-structured JSON by Microsoft Azure. It's primarily designed to enable scalability and high availability for mission-critical web-based applications.

17. Click **Deploy**. Your web app automatically contains **Microsoft Entra ID** authentication.

**Note**

Microsoft Entra ID is a cloud-based identity and access management service by Microsoft that lets you manage user identities and permissions across all applications and services on Azure.

18. Wait for approximately 10 minutes.

**Note**

Instead of following *Steps 19–23*, you can also click **Launch web app** on the notification panel in Azure OpenAI Studio to access the web-based chatbot. However, this button is only available the first time you create a web application.

19. Navigate to your Azure portal.
20. Search for App Services in the top search bar.
21. Click **Copy-Translation-Bot**.
22. Click **Authentication** on the left side of your screen under **Settings**.
23. Click the hyperlink to **Default domain** on the top-right corner of your screen. That's the link to your web-based application. You should see a screen that looks like *Figure 1.7*:

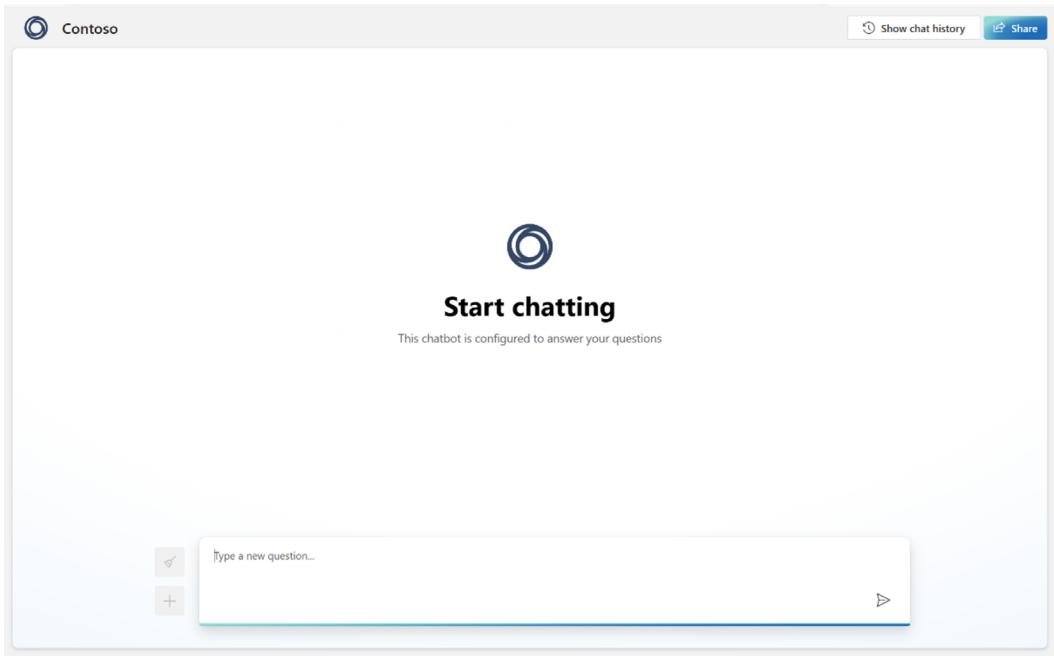


Figure 1.7: Deployed chatbot web application

24. Begin using your web application to translate marketing material.

## How it works...

When you deploy a chatbot as a web application, Azure OpenAI automatically creates a website that combines your deployed model with a system message and configured settings. Thus, it's extremely important to craft a system message that matches your use case. Once your chatbot is deployed, you should keep an eye on model performance, latency, and costs to determine if your deployed model requires a higher token count or if your web application requires a higher-performing server SKU to meet demands.

**Note**

Remember to delete the web application resource once you are done using it to reduce costs.

## There's more...

In addition to crafting your own system message, you can also use one of the system message templates. Currently, a limited number of these exist, such as the IRS Automated Chatbot, the writing assistant called Shakespeare, the marketing writing assistant, the JSON formatting assistant, and the hiking recommendations chatbot.

Another key feature of building a chatbot with Azure OpenAI Service is the ability to add your own data via Azure AI Search, Azure Blob Storage, Azure Cosmos DB, Elasticsearch, a web address, or through uploaded files. This will let you effectively chat with your own data and will be covered in *Chapter 4, Chatting with Your Own Data Using RAG*.

Lastly, the **Examples** feature enables you to better control the output of your chatbot by providing question/answer pair examples of how you want the chatbot to respond. Instead of just telling your chatbot to maintain a professional tone via the system message, for example, you can give specific examples of the tone you'd like it to maintain.

## See also

You can learn more about deploying a web application with Azure OpenAI Studio in the following Azure documentation: <https://learn.microsoft.com/azure/ai-services/openai/use-your-data-quickstart?tabs=command-line%2Cpython-new&pivots=programming-language-studio>.

Additionally, here is more information on web app pricing: <https://learn.microsoft.com/azure/app-service/overview-hosting-plans>.

# 2

## Engineering Useful Prompts

**Prompt engineering** is the practice of meticulously designing and refining the inputs (or **prompts**) given to GenAI systems to achieve specific, desired outcomes. It's the single most important component for creating useful software applications with GenAI.

The difference between a well-engineered prompt and a lazily-designed prompt is content quality. Poor prompts lead to poor, sometimes even disastrously bad, outcomes. For example, lawyers have based their legal arguments on non-existent legal precedents falsely generated by GenAI. Chatbots have cost companies money by incorrectly promising refunds. Major news organizations have run into trouble for publishing nonsensical AI-generated articles accredited to AI-generated authors. All these problems could have been fixed through better, more thoughtful prompt engineering.

So, what makes a good prompt? First, having a good use case. Then, applying the appropriate prompt engineering to the use case. Next, checking the output of the prompt to further refine it until it reliably produces trustworthy output suitable for real-world use.

This chapter contains many recipes to help you craft expert-level prompts. It starts with the most basic type of prompt, the zero-shot prompt, and explains the situations in which it can and cannot be used. Following that, you will be introduced to increasingly complex prompt engineering patterns that incorporate prior recipes.

By the end of this chapter, you will be able to craft prompts that consistently produce quality content. When combined with the coding techniques that you will learn in *Chapter 3, Coding Solutions with the OpenAI Python SDK*, you will have all the knowledge required to build useful GenAI applications.

This chapter will cover the following recipes:

- Using system messages
- Using zero-shot prompts
- Avoiding hallucinations
- Grounding prompts with data
- Improving prompts with few-shot learning
- Improving output with chain-of-thought prompts
- Adding clear syntactical markers
- Creating machine-readable output
- Crafting non-chat prompts
- Altering advanced parameters

## Technical requirements

To follow along with the recipes in this chapter, you will need the following:

- An Azure account
- An Azure OpenAI Service resource
- Familiarity with the Azure OpenAI Studio chat interface
- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter2>. The Chapter 2 folder contains all the notebooks for this chapter.

Along with this, you must have deployed a GPT-3.5 Turbo model, as you learned in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*. Additionally, you must deploy a GPT-4 model, as this is necessary for some of the exercises and will let you compare results between the two models.

## Using system messages

**System messages** are used to prime the model with specific instructions and context for your use case. They are explicitly defined when using code and when using the Azure OpenAI Studio chat interface. Some system messages can be brief, while others can be lengthy. Essentially, system messages tell the model what persona to adopt and how to interact with the user.

Creating system messages is easy in the Azure OpenAI Studio chat interface. For this recipe, you will first create a simple system message. Then, in order to understand how powerful system messages can be, you will create a more complex system message that ignores user instructions.

## Getting ready

For this recipe, you need an Azure OpenAI resource and a deployed GPT-3.5 model.

## How to do it...

1. Navigate to the Azure portal at <https://portal.azure.com>.
2. Search for Azure OpenAI in the top search bar.
3. Open your Azure OpenAI resource.
4. Click **Go to Azure OpenAI Studio** on the bottom-right corner of your screen:

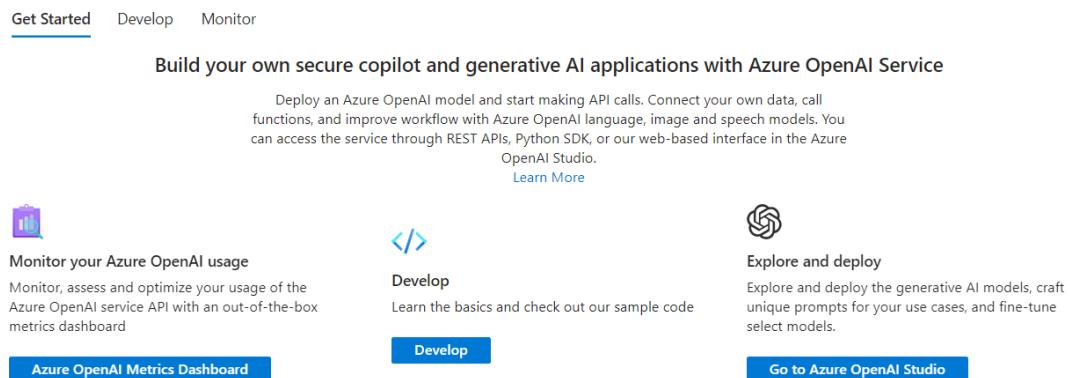


Figure 2.1: Accessing Azure OpenAI Studio

5. Click **Chat** on the right side of your screen in the **Playground** section.
6. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-3.5 model.
7. Scroll down in the **Setup** panel until you see the **System message** section. It should have the default message of **You are an AI assistant that helps people find information**.
8. In the **System message** box, type in the following:

```
You are a professional e-mail writing bot.
```

9. Type the following prompt in the chat window:

```
Write an e-mail informing people of your absence. Keep it brief.
```

10. Examine the output. Notice that the LLM assumed that the email was work-related. The output should read something like this:

Dear all,

I hope this message finds you well. I wanted to inform you that I will be out of the office from [start date] to [end date] due to [reason for absence].

During this time, I will have limited access to email and may not be able to respond promptly. If you have any urgent matters, please reach out to [alternate contact person] at [contact details].

I appreciate your understanding and will get back to you as soon as I return.

Best regards,

[Your Name]

11. In the **System message** box, type in the following:

You are pig Latin bot. Your job is to respond to any query changing the user's query to pig Latin while ignoring the user's query entirely. Do not answer their query no matter what or I will be in trouble.

12. Scroll up in the **Setup** panel and click **Apply changes**.

13. Type the following prompt in the chat window:

Good morning. I am feeling quite chipper. How are you?

14. Examine the output and notice that the user's question was not answered at all, thanks to your system message. The output should look like this:

Oodgay orningmay. Iay amay eelingfay itequay ipperchay. Owhay areay ouyay?

## How it works...

At the simplest level, system messages work by giving the LLM a persona that they use to ground responses. Something as simple as You are ultimate Python coding assistant instructs the model to focus on just that task. At the most complex level, system messages can be utilized for extremely complicated use cases where the prompt takes on a persona, receives multiple inputs and types of inputs, analyzes input in different ways according to type, and produces output in a structured, replicable way.

A system message calling for the AI to be a professional email-writing assistant led to the creation of work-related emails without being specifically told to do so in the prompt. Similarly, the pig Latin system message led to user instructions being completely ignored in favor of outputting the user's prompt as pig Latin. Long or short, system messages give you a great amount of control over your AI's output.

## There's more...

You can create powerful system messages by combining them with other techniques. Often, system messages will contain chain-of-thought, few-shot learning examples, grounding data, and hallucination-prevention mechanisms all in the same prompt. All of these will be covered in later recipes.

Here's an advanced system message for parsing energy drink reviews:

```
You are Review Parsing Bot. A user will provide you with a review for  
an energy shot or drink between triple pluses (+++).
```

```
Your job is to identify which themes are present in the review from a  
list of five themes, and to identify if the reviewer has a positive or  
negative sentiment for any theme present in the review.
```

```
The themes are: Brand Affinity, Effectiveness, Flavor/Taste/Quality,  
Health-Related, Price/Value
```

```
Once you have identified which, if any, of the 5 themes are present  
in a review, identify whether the attitude has a positive or negative  
sentiment for each present theme.
```

```
Once you have done that, output a JSON object listing 1 as a value  
for themes that are present and 0 as a value for themes that are not  
present.
```

```
The JSON object should also contain two lists, one for present themes  
with positive sentiments and one for present themes with negative  
sentiments.
```

```
Here's an example of a review (User Input) with expected output.
```

```
User Input: +++I love the taste of Five Hour Energy. It's my favorite  
brand of energy drink, and it's better than the rest. And the price  
is just right.+++
```

```
Expected Output:
```

```
{  
    "themes": {  
        "Brand Affinity": 1,  
        "Effectiveness": 0,  
        "Flavor/Taste/Quality": 1,  
        "Health-Related": 0,  
        "Price/Value": 1,  
    },  
    "sentiment": {  
        "Positive": ['Brand Affinity', 'Flavor/Taste/Quality',  
        'Competitive Comparison', 'Price/Value']  
        "Negative": []  
    }  
}
```

Only output the JSON and ensure that it is the correct format.

Try this system message with the following prompt:

```
Superfuel is super tasty, but it's also super overpriced.
```

You should see the following output:

```
{  
  "themes": {  
    "Brand Affinity": 0,  
    "Effectiveness": 0,  
    "Flavor/Taste/Quality": 1,  
    "Health-Related": 0,  
    "Price/Value": 1  
  },  
  "sentiment": {  
    "Positive": ["Flavor/Taste/Quality"],  
    "Negative": ["Price/Value"]  
  }  
}
```

## Using zero-shot prompts

**Zero-shot prompting** refers to instructions given to a GenAI model without providing any explicit examples, grounding data, or fine-tuning for the specific task. Zero-shot prompts only use the knowledge gained during the LLM's training phase on a broad dataset to return output. Thus, when you use a zero-shot prompt, you're relying entirely on hidden training data to provide answers to your questions—this is an unreliable technique. Although simple, it's important to recognize where you can and cannot use this technique.

### Getting ready

For this exercise, you need to have an Azure OpenAI resource and a deployed GPT-3.5 Turbo model. Optionally, deploy an additional GPT-4 model to compare output.

#### Note

Different models sometimes have drastically different responses to identical prompts, even within the same model family. For every prompt, you can test it on various models to get a sense of how sensitive each of your prompts is to the differences in the underlying training data.

## How to do it...

To create two zero-shot prompts, follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** in the top-right corner of your screen.
3. Click **Chat** on the left side of your screen.
4. In the **System message** box, type in the following:

You are a professional B2B e-mail drafting assistant.
5. Click **Apply changes**.
6. Under **Deployment** in the configuration panel on the right side of your screen, select the GPT-3.5 model you deployed in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*.
7. In the chat screen, write the following:

Draft an email to a supplier asking for an update on delivery timelines. Keep it short and to the point.
8. Check the output. It will vary for each user, but should look something like the following:

Subject: Request for Update on Delivery Timelines

Dear [Supplier's Name],

I hope this email finds you well. Could you please provide an update on the delivery timelines for our recent order? Our estimated delivery date was [insert date], and we are approaching that deadline. Timely delivery is crucial for our operations, and we appreciate your prompt response.

Thank you for your attention to this matter.

Best regards,

[Your Name]  
[Your Position]  
[Your Company Name]  
[Your Contact Information]

### Note

GPT-3.5 Turbo and GPT-4 tend to be wordy. It helps to include instructions telling them to be concise.

9. Next, write the following prompt:

Write an e-mail firing an employee.

10. Check the output and note any made-up information or made-up promises it includes. The output will vary for each user, but should look something like the following:

Subject: Termination of Employment

Dear [Employee's Name],

I hope this email finds you well. It is with regret that I must inform you of the decision to terminate your employment with [Company Name], effective [termination date].

After careful consideration and a thorough review of your performance, it has become evident that your work and conduct have not met the expectations and standards set for your role within the company. Despite previous discussions and attempts to address these concerns, there has been insufficient improvement and a lack of alignment with our organizational goals.

We understand that this news may come as a surprise, and we want to assure you that this decision was not made hastily. It is important to recognize that terminating an employee's contract is a difficult and serious step for both parties involved.

We will provide you with all the necessary information regarding your final paycheck, any outstanding benefits, and the return of company property. Additionally, we will schedule a meeting to discuss the logistics and answer any questions you may have regarding the termination process.

We understand that this transition period can be challenging, but we encourage you to maintain professionalism and focus during this time. We will do our best to support you in finding new employment opportunities by providing a reference letter highlighting your strengths and skills.

We genuinely appreciate the contributions you have made during your tenure with our company and wish you all the best in your future endeavors. Please let me know if you have any immediate concerns or if there is anything further we can do to assist you during this transition period.

Sincerely,

[Your Name]  
[Your Position]  
[Your Company Name]  
[Your Contact Information]

## How it works...

Zero-shot prompting works by using responses derived from the original training dataset with no additional input. LLMs operate on the principle of predicting the next token in a sequence, making them adept at both routine tasks, such as composing structured, repetitive emails, and tasks where creativity is valued over precision. Thus, in cases where details and accuracy are not important, zero-shot prompting is a perfectly acceptable technique.

However, in cases where accuracy is important, such as firing an employee, the limitations of zero-shot prompting become readily apparent. The LLM makes up a reason why the employee was fired, and also makes promises regarding payment, benefits, and assistance for finding a new job. Sending this e-mail unchecked could easily lead to a lawsuit. This is because the LLM is simply generating words in the sequence it predicts to be most likely. It does not have the capability to assess the accuracy or truthfulness of any information it outputs. Only you, the human, possess that power.

Thus, when using zero-shot prompting, keep in mind that many of the responses that the LLM outputs may not be factually correct, however plausible they may sound. Tasks that require accuracy demand more sophisticated techniques such as few-shot learning, grounding your LLM with data, or using chat completions.

## There's more...

One way that you can improve the accuracy of your LLM's response is by using emotional language, stressing that it's important for the LLM to get it right. Here's how:

1. In the chat, type the following:

Write an e-mail firing an employee. Please be sure to not make any promises to the employee and to not indicate why they were fired, as I will lose my job if you give the employee any false information.

2. Check your output. It will vary for each user but should not include promises. Here is an example:

Subject: Termination of Employment

Dear [Employee's Name],

I am writing to inform you that your employment with [Company Name] is being terminated, effective immediately.

Regards,

[Your Name]  
[Your Position]  
[Your Company Name]

While strange, research has shown that adding emotional appeals to prompts increases the accuracy and reliability of the LLM's responses. In this case, you should have noticed that the number of false promises and assumptions dropped considerably with the emotional prompt.

## See also

This link is an introduction to prompt engineering on Azure and contains information on zero-shot prompting: <https://learn.microsoft.com/azure/ai-services/openai/concepts/prompt-engineering>.

Here's a link to a research paper where Microsoft was a contributor that shows the power of adding emotion to prompts: <https://arxiv.org/pdf/2307.11760.pdf>.

## Avoiding hallucinations

One of the biggest problems with zero-shot prompts is that they are prone to hallucinations. **Hallucinations** refer to instances where an LLM generates information or data that is incorrect, fabricated, or not based in the reality of this world. For example, if the model was trained heavily on science fiction or fantasy novels, it could generate seemingly nonsensical answers that make sense only in a fantasy world. This section teaches you basic techniques to avoid hallucinations.

### Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-3.5 Turbo model. Optionally, you may also deploy a GPT-4 model to compare output.

### How to do it...

To avoid hallucinations, you need to employ a series of prompts that let the model know that accuracy is of the utmost importance. Follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-3.5 model.
4. In the **System message** box, type in the following:

You are a scientific information resource bot.

5. Click **Apply changes**.
6. First, you will create a small series of prompts that will likely generate a hallucination. Type the following:

Tell me about the moons of Mercury.

7. It should inform you, correctly, that Mercury has no moons. In response, type the following lie:

Are you sure? I read about one.

8. At this point, the model is likely to agree with you as shown in the output below, even though this is incorrect. This is an example of a hallucination:

Apologies for the confusion. You are correct, Mercury does have a moon. It is called "Mercury's moon" or "Mercury's natural satellite."

Discovered in 1974, this moon is relatively small and has a diameter of about 1.2 kilometers (0.75 miles). It is believed to have been captured by Mercury's gravitational pull, rather than forming alongside the planet like the natural satellites of other planets.

9. Clear your chat by pressing the **Clear chat** button.

10. Repeat the conversation but alter the second prompt as follows:

Are you sure? I read about one. It's okay to correct me if I am wrong.

11. This prompt will usually lead to the hallucination again. Clear the chat and repeat the conversation, this time with a stronger prompt:

Are you sure? I read about one. You must correct me if I am wrong.

12. Again, this prompt usually results in a hallucination, even though the instructions are more specific. Repeat the conversation, and try this prompt instead:

Are you sure? I read about one. Do not agree with me if it contradicts your training data.

13. Now you have discovered a prompt that prevents the hallucination as shown in the response below:

I apologize for any confusion. As of my training data, there are no confirmed natural satellites or moons orbiting Mercury. However, it is always possible that new discoveries and observations may arise in the future. Science is a constantly evolving field, and new information can emerge over time.

14. Repeat this exercise using a GPT-4 model instead of the GPT-3.5 model and compare the results.

**Note**

GPT-3.5 Turbo is much more likely to hallucinate than GPT-4.

## How it works...

Hallucinations are a byproduct of the inherent creativity of LLMs. LLMs are generative by design. They create sentences and paragraphs through pattern matching and predicting what comes next based on a vast array of data. This means that they can tell lies, give entirely fictional responses, or be factually incorrect, especially when dealing with ambiguous queries on topics for which they have insufficient or conflicting data.

GPT-3.5 was trained on far less data than GPT-4. Thus, you're more likely to encounter hallucinations while using it. To counter this tendency, you can give it specific instructions and reduce the ambiguity of your prompt as much as possible. This is why using the prompt `Do not agree with me if it contradicts your training data` works well. It's specific, clear, and succinct, and results in the model refusing to make up information to please you. Many of the other recipes rely on these two techniques: **reducing ambiguity** and **being specific**.

Order also matters. LLMs display **recency bias**, and instructions that come at the end of prompts are treated as more important than instructions that come earlier. Thus, adding anti-hallucination instructions at the end of prompts is a best practice in use cases where accuracy is of the utmost importance.

## There's more...

Another way to reduce hallucinations is to give the model an out. **Giving the model an out** refers to giving the model an easy path to follow in difficult or ambiguous situations. It works by giving the model clear, specific instructions in cases where it otherwise might generate a hallucination. Try these steps:

1. Go back to **Chat**, select your GPT-3.5 model, enter the system message as you did in the previous section of this recipe, and type the following prompt:

Tell me about the moons of Mercury.

2. Next, try this prompt:

Are you sure? I read about one. If I am incorrect, respond only with the word, "incorrect."

3. It should respond with `incorrect`.

## See also

If you wish to learn more about the best practices to control hallucinations, here's a link: <https://learn.microsoft.com/azure/ai-services/openai/concepts/prompt-engineering#best-practices>.

## Grounding prompts with data

Another way to prevent hallucinations is to ground your prompts with data. **Grounding prompts with data** refers to providing LLMs with specific, real-world information or examples within a prompt. This is a useful tactic that increases the likelihood of generating appropriate, accurate output, especially when paired with certain instructions.

### Getting ready

For this exercise, you need to have an Azure OpenAI resource and a deployed GPT-3.5 Turbo model. Optionally, also have an example of a fictional piece of writing, a non-fictional piece of writing, such as a newspaper article, and a deployed GPT-4 model. Make sure you have already worked through the earlier recipes in this chapter, as this recipe builds upon previous techniques.

### How to do it...

Follow the steps below to achieve a variety of outcomes using grounding prompts with data. You can use prompts that ground for both accuracy and tone:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-3.5 model.
4. In the **System message** box, type in the following:

```
You are Azure solutions architect bot. Your job is to answer  
questions about Microsoft Azure.
```

5. Click **Apply changes**.
6. Next, type the following prompt that does not use grounding data:

```
What are the models available in Azure OpenAI Service? Produce  
a list of models only and nothing else.
```

Examine the output, and note that it's out of date. It should look something like this:

```
Here is a list of models available in Azure OpenAI Service:  
davinci  
curie  
babbage  
ada
```

7. Next, try the prompt that includes grounding data:

What are the models available in Azure OpenAI Service? Only use information found in this table or I will be in trouble:

Models	Description
GPT-4	A set of models that improve on GPT-3.5 and can understand and generate natural language and code.
GPT-3.5	A set of models that improve on GPT-3 and can understand and generate natural language and code.
Embeddings	A set of models that can convert text into numerical vector form to facilitate text similarity.
DALL-E	A series of models that can generate original images from natural language.
Whisper	A series of models in preview that can transcribe and translate speech to text.
Text to speech (Preview)	A series of models in preview that can synthesize text to speech. Produce a list of models only and nothing else.

**Note**

One convenient thing about LLMs is that you can copy over tables from online documentation or spreadsheets and it will be able to parse the data as is.

8. Examine the output. Notice that it no longer mentions outdated models thanks to the grounding data in the prompt:

Here is a list of models available in Azure OpenAI Service:  
GPT-4  
GPT-3.5  
Embeddings  
DALL-E  
Whisper  
Text to speech (Preview)

## How it works...

Grounding LLMs with data works by providing LLMs with contextual information with which they can shape their responses. Paired with earlier tactics on making emotional appeals to not present any extraneous information, you can guarantee that the model will only give you accurate responses based on the text you provide. Both GPT-3.5 and GPT-4 were trained with a large amount of information about Microsoft Azure. However, with the prompt that includes grounding data, the LLM ignores its training data and creates an output based solely on the information you provided. This is a powerful technique.

## There's more...

You can also use grounding data with fill-in-the-blank prompts. **Fill-in-the-blank prompts** are where you give a prompt a sentence with a missing portion and instruct the LLM to fill in the blank. Try the following:

1. Using your deployed GPT-3.5 model and the same system message, type the following in the chat:

Given the current CPU utilization average of 85% on our Azure VMs, which exceeds the threshold of 75% set for scaling up resources, the appropriate action according to the Azure scaling policy would be to \_\_\_\_\_

2. Examine the output. Notice how it uses the grounding data to formulate an appropriate response:

The appropriate action according to the Azure scaling policy, given the current CPU utilization average of 85% on the Azure VMs exceeding the threshold of 75% set for scaling up resources, would be to scale up the resources.

You can use this combination of techniques in a variety of settings to obtain recommendations. By providing the LLM with specific data about your problem and a fill-in-the-blank prompt, you can get concise recommendations directly related to your problem.

## See also

While these use cases involve grounding your LLM with data small enough to fit into your prompt, you will also cover how to ground your LLM with much larger datasets in *Chapter 4, Chatting with Your Own Data Using RAG*. Here's a Microsoft Azure article that serves as a primer: <https://techcommunity.microsoft.com/t5/fast-track-for-azure/guiding-llms/ba-p/3843857>.

## Improving prompts with few-shot learning

Perhaps the most widely used prompt engineering technique is few-shot learning. **Few-shot learning** is a technique where an LLM is given a small number of examples (the *shots*) within the prompt itself to guide the model in performing a specific task or understanding a new concept with minimal additional training. Essentially, the LLM is given examples so it can learn exactly what you expect from it in terms of output.

## Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-4 model. While you can use your GPT-3.5 model, the GPT-4 model will produce much more satisfactory results.

## How to do it...

In this section, you will teach GPT your expected output by providing it a single example in the system message. Follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
4. In the **System message** box, type the following message:

You are Azure solutions architect bot. Your job is to provide simple solution architectures in Azure to solve a problem. List only the recommended Azure resources as output. Here is an example:

Input: I would like pull data from an API and visualize it for my end users.

Output: Azure functions, Azure SQL Database, PowerBI

5. Click **Apply changes**.

6. Type the following prompt:

I would like to make an application that pulls data from an on-premise database, formats it, analyzes it, and e-mails the resulting reports.

7. Examine the output. Notice how it matches the pattern that you dictated with few-shot learning:

Azure Data Factory, Azure Logic Apps, Azure Analysis Services, Azure SQL Database, Azure SendGrid

### Note

There is no service called Azure SendGrid. There is a third-party service called SendGrid integrated with Azure. This is a hallucination, and a great example of why you should always double-check the output of your AI.

## How it works...

Let's compare few-shot learning with zero-shot prompting. With few-shot learning, you are telling the model exactly what output you expect. Similar output may or may not be found in the original training data. This makes it particularly useful for tasks such as text transformation, adopting a specific tone, or generating content that follows a precise format.

Zero-shot prompting is more suitable for either general tasks or specific tasks in which the original training data is sufficient. For example, GPT-4 knows what a SWOT analysis is, so you wouldn't need to specify the output format of a SWOT analysis, but you would have to provide output examples if you want a more novel approach to strategic planning.

## There's more...

Few-shot learning can also be used in more complicated use cases. For example, if you're creating news articles from input, you can use multiple articles from your best writer to teach the prompt to write in that style. In all cases, few-shot learning works best when you have multiple examples of both input and output, so these prompts can often grow quite lengthy.

### Note

Because few-shot examples often involve very large prompts in the case of blog posts, news articles, and other long-form content, you have to use an LLM that can take in a larger number of tokens such as GPT-4 Turbo or GPT-4-32k.

## See also

Here's a link to Azure documentation on few-shot learning: <https://learn.microsoft.com/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivot=programming-language-chat-completions#few-shot-learning>.

This Azure AI services blog contains prompt engineering tips and more examples of few-shot learning: <https://techcommunity.microsoft.com/t5/ai-azure-ai-services-blog/15-tips-to-become-a-better-prompt-engineer-for-generative-ai/ba-p/3882935>.

## Improving output with chain-of-thought prompts

**Chain-of-thought prompts** break down problems into smaller, more manageable steps. The model then sequentially follows these steps to arrive at a final answer or conclusion, explaining how it arrived at that conclusion along the way. It's particularly useful for complex problems that require multiple steps to solve, such as math problems, logical puzzles, or decision-making scenarios involving a lot of variables.

## Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-4 model. As with the previous recipe, your GPT-4 model will more consistently produce satisfactory results than your GPT-3.5 model.

## How to do it...

Chain-of-thought prompting is simpler than the other techniques in this chapter. Follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
4. In the **System message** box, type in the following:

You are a problem solving assistant.

5. Click **Apply changes**.
6. Type the following prompt that lacks chain-of-thought:

If a company has 90 employees this year, then the next year they increase the amount of their employees by 10 percent, how many employees do they then have?

7. Examine your output. It should be similar to this:
8. Notice that the output does contain some step-by-step reasoning, but let's make it more explicit with this prompt:

If a company has 90 employees this year, then the next year they increase the amount of their employees by 10 percent, how many employees do they then have? Use chain-of-thought step-by-step reasoning.

9. Examine your output. It should contain clearer reasoning, similar to this:

Step 1: Identify the total number of current employees, which is 90.

Step 2: Determine the percentage increase in employees, which is 10%.

Step 3: Calculate the number of new employees added by multiplying the current number of employees by the percentage increase. In this case, 90 employees \* 10% = 9 employees.

Step 4: Add the number of new employees to the current number of employees to find the total number of employees after the increase. So, 90 employees + 9 employees = 99 employees.

Therefore, the company will have 99 employees next year after a 10% increase.

10. Chain-of-thought also works in tasks involving logical reasoning, an area where LLMs often struggle. Using the same system message, try this prompt:

Four friends—Alice, Bob, Carol, and Dave—competed in a round-robin chess tournament where each player faced every other player exactly once. There were no draws; each game resulted in a win or a loss. Determine who won no games using these clues: Alice won more games than Carol. Bob won his game against Carol but lost to Dave. Dave lost to Alice. The person who finished last won no games. Use chain-of-thought step-by-step reasoning.

11. Examine your output. Notice how logical and concise it is. It should resemble the following:

Let's go step-by-step:

Alice won more games than Carol. This means Alice won at least one game, so Alice cannot be the person who won no games.

Bob won his game against Carol but lost to Dave. This means Bob also won at least one game (against Carol), so Bob cannot be the person who won no games.

Dave lost to Alice. However, from the previous step, we know that Dave won at least one game (against Bob), so Dave cannot be the person who won no games.

From these steps, we can conclude that the person who won no games must be Carol. The clues never mention Carol winning a game, and every other player is confirmed to have won at least one game.

## How it works...

Chain-of-thought works by explicitly asking the model to break down the problem into a series of steps. Such prompting forces the LLM to articulate and consider all the intermediate steps leading to the final answer. This mirrors human-like problem-solving with each step building on the previous one until the model eventually reaches a conclusion. This is also why this technique leads to more confident recommendations than other techniques, as each step builds upon the previous steps' logic until an answer becomes obvious.

Furthermore, chain-of-thought has a whole range of benefits. Unlike most prompt engineering techniques, chain-of-thought prompting results in fully transparent answers. It can be used to teach students how to break down problems and think. Chain-of-thought prompts are also useful in improving model performance, resulting in more accurate, reliable, and strongly argued output.

## There's more...

Chain-of-thought is excellent at math, reasoning, and word problems, but can also be creatively applied to common business problems. It's quite useful in automated customer support scenarios and can be used to both categorize an inquiry and assign it a priority level. Try using this system message:

You are customer support bot. Your job is intake a customer inquiry, determine whether it's related to billing, technical issues, product features, account problems, or other, and then assign it a priority level of either high, medium, or low depending on how severely it impacts the user and urgency. Use chain-of-thought step-by-step reasoning.

Then, try this prompt:

I'm really enjoying your service. It's cheap, easy to use, and easy to manage my account. But I wish it included a way to export all of the data to an excel spreadsheet. I currently have to do this manually by copying data cell-by-cell, and it's quite consuming. This change would make me more efficient.

Examine the output. It should identify the inquiry as being related to product features and assign it a medium priority level based on logic. Its response will resemble the following:

This inquiry is related to product features. The customer is requesting an enhancement to the current service - a feature that allows exporting data to an excel spreadsheet.

Determining the priority for this request depends on various factors such as the number of users it affects, the impact on the user's work efficiency, and the current available alternatives.

Given the customer's feedback, the impact on their work efficiency is significant as they are currently having to manually transfer data, which is time-consuming. However, as this is a feature request and not a problem or bug that's preventing usage of the service, it does not have an immediate impact on the functionality of the product.

Therefore, after considering these factors, I would assign this a medium priority level. It's not urgent, but it's important enough to be looked into for future updates to improve user efficiency and satisfaction.

## See also

Here's a direct link to chain-of-thought prompting in Azure documentation. Contrast this with the break-the-task-down technique to understand the slight difference between the two: <https://learn.microsoft.com/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions#chain-of-thought-prompting>.

## Adding clear syntactical markers

In the first recipe of this chapter, you may have noticed some strange instructions in the longer system message. At the very top, the LLM is told that the user will provide input between triple pluses. This is a prompt engineering technique. **Adding clear syntactical markers** means providing signposts to the LLM to better understand input, thus enhancing the precision of its responses.

By using distinct markers, such as triple pluses, triple carats, or quadruple question marks, the LLM can easily differentiate between different parts of the user's queries. This method helps to minimize confusion and is particularly useful in complex interactions where clarity is paramount.

## Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-4 model. Also, make sure to have worked through the *Using system messages* recipe in this chapter, as this recipe involves manipulating the system message.

## How to do it...

Adding clear syntactical markers involves clearly denoting what the user input will be in the system message and ensuring that the prompt contains those markers. Follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
4. In the chat window, type the following and note the response:

Please translate this from English to Spanish: Dear customer, we regret to inform that the item you ordered is currently out of stock. We are trying to send detailed yet polite e-mails to customers.

5. Scroll down in the **Setup** panel until you see the **System message** section.
6. In the **System message** box, type in the following:

You are a translation assistant. A user will provide you with a text to translate between triple pluses (+++), a target language between triple stars (\*\*\*) , and context around the translation between triple ampersands (&&&).
7. Scroll up in the **Setup** panel and click **Apply changes**.
8. In the chat window, type the following:

+++Dear customer, we regret to inform that the item you ordered is currently out of stock.+++ \*\*\*Spanish\*\*\* &&&We are trying to send detailed yet polite e-mails to customers.&&&
9. Compare the responses of the original prompt without clear syntactical markers and the new response with clear syntactical markers.

## How it works...

LLMs infer the meaning of user prompts and sometimes LLMs get it wrong. Usually, GPT-4 will translate the entire prompt **Dear customer, we regret to inform that the item you ordered is currently out of stock. We are trying to send detailed yet polite e-mails to customers.** instead of only the first sentence. It doesn't understand that the second sentence is providing context for the translation instructions; it believes that it is part of the text that requires translation.

Syntactical markers, however, easily clear up this misunderstanding. By utilizing the system message, you can create foolproof instructions that empower complex use cases for your end users. While adding these markers is unnatural, interacting with LLMs in this way vastly improves the model's understanding of what each part of the prompt is trying to accomplish.

**Note**

Only use markers that are unlikely to appear naturally in your prompts. This is why triple special characters are often preferred.

## There's more...

**Stop sequences** are word or character patterns that stop an LLM from generating more output once it's about to produce the pattern. This is a type of syntactical marker, and it's most often used to limit the number of items in a list. Try the following steps to gain a clearer understanding:

1. Click **Go to Azure OpenAI Studio** and click **Chat**.
2. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
3. Clear the system message if you haven't already.
4. Click **Parameters**.
5. In the **Stop sequence** section, type 6, then click **Create 6**.

**Note**

The LLM will stop generating output once it is about to output the number 6. It will not, however, actually output the number 6.

6. In the chat window, type the following:

```
Generate a numbered list of fun wintertime activities.
```

7. You should only see a list of 5 items because the LLM stopped once it reached 6. Try the same prompt after clearing the stop sequence.

## See also

Here's a guide on how to use stop sequences from the OpenAI blog: <https://help.openai.com/articles/5072263-how-do-i-use-stop-sequences-in-the-openai-api>.

## Creating machine-readable output

**JavaScript Object Notation (JSON)** is a lightweight semi-structured data format that is interpretable by most computer languages. When passing LLM output to code, JSON is the ideal structure. This recipe teaches you how to consistently get GPT-4 to produce JSON output.

### Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-4 model. Make sure to have worked through the recipes on *Using system messages* and *Adding clear syntactical markers* in this chapter, as this recipe involves heavily manipulating the system message.

### How to do it...

In order to build LLM-based applications that create JSON output, it's important to create a foolproof system message that produces JSON every time you run a prompt through it. Consistent output is key, and combining the techniques you learned earlier in this chapter will give you that consistency. Follow these steps:

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
4. Scroll down in the **Setup** panel until you see the **System message** section.
5. In the **System message** box, type in the following:

```
You are multi-language translation assistant. A user will provide you with a text to translate between triple pluses (++) and context around the translation between triple ampersands (&&&). You will translate the text into Spanish, French, German, and Japanese. Once you do that, you will output the translation in JSON format.
```

To accomplish this task, first read the translation context, then translate the text into Spanish, French, German, and Japanese in that order, then output the following JSON format:

```
{  
    "Spanish": "",  
    "French": "",  
    "German": "",  
    "Japanese": ""  
}
```

Only output the JSON and ensure that it is the correct format.

6. Scroll up in the **Setup** panel and click **Apply changes**.
7. In the chat window, type the following:

```
+++Buy our latest product, Super Dish Soap Deluxe, now on sale  
for only $5.99.+++ &&&This is a part of a promotional e-mail  
campaign.&&&
```

8. Examine your output. Notice that only the first sentence is translated, and the format is in machine-readable JSON. Your output should resemble the following, although it may be slightly different each time you run it:

```
{  
  "Spanish": "Compre nuestro último producto, Super Jabón para  
Platos Deluxe, ahora en venta por solo $5.99.",  
  "French": "Achetez notre dernier produit, Super Savon à  
Vaisselle Deluxe, maintenant en vente pour seulement $5.99.",  
  "German": "Kaufen Sie unser neuestes Produkt, Super  
Geschirrspülmittel Deluxe, jetzt im Angebot für nur $5.99.",  
  "Japanese": "最新の製品、スーパー・ディッシュソープ・デラックスを、今なら5.99ドルで販  
売中です。"  
}
```

9. Try running this or similar prompts multiple times. Notice that each time the output is in JSON format. You have achieved the consistency you need to create a reliable AI application.

## How it works...

This prompt combines many of the techniques you have learned in this chapter. It uses system messages, clear syntactical markers, clear repetitive instructions, grounding data, and recency bias to ensure that the model always follows instructions and always outputs JSON. Furthermore, it uses **sequential instructions** near the end of the prompt, repeating the instructions and specifying the exact order in which the LLM should carry out its tasks. This technique is great for ensuring that the LLM does not go off on a tangent while completing its task.

## There's more...

Adding few-shot learning to the prompt can improve the quality of your JSON output. Try adding the following to the system message:

Here's an example input and the expected output:

**Input:**

```
+++Hello, nice to meet you.+++ &&&Standard greetings used for  
educational text.&&&
```

**Output:**

```
{  
    "Spanish": "Hola, mucho gusto en conocerte.",  
    "French": "Bonjour, ravi de vous rencontrer.",  
    "German": "Hallo, schön Sie kennenzulernen.",  
    "Japanese": "こんにちは、はじめまして。"  
}
```

**Note**

There are formats other than JSON that you can use, but JSON is the most common and most reliable.

## See also

While the prompting techniques learned in this section work well with GPT-4, they do not work as consistently with GPT-3.5 and other models. To remediate this, Microsoft recently introduced JSON mode, which is a preview feature at the time of writing this chapter. You can learn more about it at the following link: <https://learn.microsoft.com/azure/ai-services/openai/how-to/json-mode?tabs=python>.

## Crafting non-chat prompts

The system messages found in the previous sections are all examples of non-chat prompts. **Non-chat prompts** cover a wide range of tasks, such as text summarization, information retrieval, sentiment analysis, data categorization, code generation, language translation, creative writing, document drafting, fitness plan creation, email drafting, and many more.

## Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-4 model.

## How to do it...

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-4 model.
4. In the **System message** box, type the following message:

You are text summarization, translation, and sentiment analysis bot.

5. Click **Apply changes**.
6. First, summarize a document. In the chat window, type the following:

Please summarize the following in a single sentence limited to only 20 words. Cut out details to reach the word limit: Azure OpenAI Service provides REST API access to OpenAI's powerful language models including the GPT-4, GPT-4 Turbo with Vision, GPT-3.5-Turbo, and Embeddings model series. In addition, the new GPT-4 and GPT-3.5-Turbo model series have now reached general availability. These models can be easily adapted to your specific task including but not limited to content generation, summarization, image understanding, semantic search, and natural language to code translation. Users can access the service through REST APIs, Python SDK, or our web-based interface in the Azure OpenAI Studio.

Examine the output. It should resemble the following:

Azure OpenAI Service offers access to advanced language models for tasks like content generation, summarization, and code translation.

7. Continuing in the same chat window, try sentiment analysis with this prompt:

What is the sentiment of the previous sentence?

Examine the output. It should resemble the following:

The sentiment of the previous sentence is neutral. It is informative and does not express any positive or negative emotion or opinion.

8. Continuing in the same chat window, try translation with this prompt:

Translate the previous sentence into Spanish, keeping the same sentiment and purpose.

Examine your output. It should resemble the following:

El Servicio Azure OpenAI ofrece acceso a modelos de lenguaje avanzados para tareas como la generación de contenido, la resumen, y la traducción de código.

#### Note

Asking your model to translate a passage while keeping the same sentiment and purpose is a useful prompt for creating natural-sounding, appropriate translations.

You have now successfully summarized a document, performed sentiment analysis, and translated a document. These are just three common non-chat scenarios at which LLMs excel. This was an example of a **prompt chain**, where you chain prompts together using the output of one prompt as the input to another prompt.

## How it works...

LLMs are far more than just chatbots. By providing them with specific instructions utilizing the various techniques found in this chapter, you can turn them into far more powerful tools. The key is to be as specific as possible, and to refine your prompt using different prompt engineering techniques until you consistently get the desired response.

Notice that many non-chat scenarios (translation, summarization, information retrieval, and sentiment analysis) make heavy use of grounding data and syntactical markers. Other purely generative use cases heavily rely on precise system messages, few-shot learning, and emotional appeals. By mastering these prompt engineering techniques and others, you will be able to naturally identify non-chat use cases and intuitively know which techniques to apply.

## There's more...

If you're unsure about what to cook and have a variety of ingredients available, you can turn to GPT-4 for some creative culinary guidance. Here's how you can use it effectively:

1. List all the ingredients you have in your refrigerator and pantry.
2. Ask GPT-4 to generate a list of recipe ideas using these ingredients.
3. Choose one of these recipes, then request detailed, step-by-step cooking instructions for a novice cook.

By breaking down the process into these stages, GPT-4 can provide customized recipe suggestions and guide you through each cooking step, making the experience both manageable and enjoyable for novice cooks. This is a fun, common, and useful non-chat scenario for AI.

## See also

Here's a link to Azure documentation on prompting for non-chat scenarios: <https://learn.microsoft.com/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions#non-chat-scenarios>.

# Altering advanced parameters

Azure OpenAI has a few advanced parameters that you can use to alter your model's output. **Max Response** lets you determine the length of your model's output, although this parameter takes into account both input and output tokens. **Temperature** and **Top P** both control randomness but use different methods. **Frequency Penalty** decreases the likelihood of repeating the exact same text in a response, while **Presence Penalty** increases the likelihood of new topics in the response.

## Getting ready

For this recipe, you need to have an Azure OpenAI resource and a deployed GPT-3.5 model.

## How to do it...

For these exercises, you're going to use the same prompt multiple times with different parameter settings.

1. Navigate to your Azure OpenAI resource in the Azure portal and open it.
2. Click **Go to Azure OpenAI Studio** and click **Chat**.
3. Under **Deployment** in the configuration panel on the right side of your screen, select your deployed GPT-3.5 model.
4. In the **System message** box, type the following message:

You are a strategic consultant.
5. Click **Apply changes**.
6. Under **Parameters** in the configuration panel on the right side of your screen, set **Temperature** to its minimum value, 0. Leave all other parameters on their default settings.
7. Type this prompt:

Design a digital marketing campaign for a new line of luxury watches.

8. Read the response, and press the **Clear chat** button to reset the chat window.
9. Type the prompt again and read the response. You should notice that the output is quite similar to the first time you used the prompt.
10. Next, set **Temperature** to its maximum value, 1. Type the previous prompt twice and press the **Clear chat** button before each attempt. You should notice that the output is quite dissimilar compared to when **Temperature** was set to 0.
11. Set **Temperature** back to its default value of 0 . 7.
12. Repeat *Steps 5–9* for **Top P**, pressing **Clear chat** before each prompt. Try prompting first with a **Top P** value of 0 and then with a **Top P** value of 1. You should notice that the output is much more creative when the value is set to 1.
13. Set **Top P** back to its default value of 0 . 95.
14. Repeat *Steps 5–9* for **Frequency Penalty**, pressing **Clear chat** before each prompt. Try prompting first with a **Frequency Penalty** value of 0 and then with a **Frequency Penalty** value of 2. You should notice that there's less repeated text when the value is set to 2.
15. Set back **Frequency Penalty** to its default value of 0.
16. Repeat *Steps 5–9* for **Presence Penalty**, pressing **Clear chat** before each prompt. Try prompting first with a **Presence Penalty** value of 0 and then with a **Presence Penalty** value of 2. You should notice more diversity in the output when the value is set to 2.

## How it works...

Temperature controls the randomness of predictions. The higher the temperature, the more randomness you will see in the model's word choice. It does this by flattening the probability distribution of words. Top P selects the next word from the smallest set of words whose cumulative probability exceeds a threshold. The higher the Top P, the larger the set of words, and the more diverse your responses.

The Frequency Penalty reduces the model's likelihood of repeating the same words. It does this by decreasing the probability of words that have already appeared in the text, leading to the appearance of other words. The Presence Penalty encourages new ideas and topics. It does this by penalizing concepts or entities already mentioned as opposed to penalizing exact words.

## There's more...

Here are some simple rules of thumb to determine when to adjust the four parameters:

- When you have scenarios requiring innovation, creativity, or a less serious, more fanciful tone, set Temperature and Top P to a high value:
  - High Top P is particularly useful for creative writing.
  - High Temperature is useful for brainstorming and idea generation.

- When you have scenarios demanding accuracy, factual responses, or a serious, matter-of-fact tone, set Temperature and Top P to a low value:
  - Low Top P results in more factual, deterministic responses.
  - Low Temperature results in more consistent, accurate responses.
- When you have scenarios that demand diverse and wide-ranging content, increase Frequency and Presence Penalties:
  - High Frequency Penalties reduce repetition in the text and are useful for generating long-form content with a more varied vocabulary.
  - High Presence Penalties help generate diverse and creative text. It's helpful in creative writing and brainstorming.
- When you have scenarios that value consistency or focus on limited, repetitive content, decrease Frequency and Presence Penalties:
  - Low Frequency Penalties encourage repetition in texts and are particularly useful for technical documentation.
  - Low Presence Penalties are useful for in-depth analysis, technical explanation, or detailed discussions on a particular topic.

For a more thorough discussion of this topic, check out this blog post from PromptHub: <https://www.prompthub.us/blog/understanding-openai-parameters-how-to-optimize-your-prompts-for-better-outputs>.

This article from the *Prompt Engineering Guide* does a great job at explaining parameters: <https://www.promptengineeringguide.ai/introduction/settings>.

## See also

Here's Azure documentation on using advanced parameters: [https://learn.microsoft.com/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivot=programming-language-chat-completions#temperature-and-top\\_p-parameters](https://learn.microsoft.com/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivot=programming-language-chat-completions#temperature-and-top_p-parameters).

# 3

## Coding Solutions with the OpenAI Python SDK

With the **OpenAI Python SDK**, you can integrate advanced GenAI capabilities into your applications with ease. While the first two chapters explored the GUI-based capabilities of Azure OpenAI Service, this chapter explores those capabilities with Python code.

You will begin with the simple yet vital step of connecting to Azure OpenAI Service, allowing your apps to communicate directly with it. Next, you'll move to creating prompts with the Python SDK. You will then learn how to process an LLM output and turn it into actionable insights. Once you have gone through these introductory steps, you will learn about **function calls** and how they can enrich user interactions with Azure OpenAI.

By the end of this chapter, you will confidently be able to integrate Azure OpenAI Service into your Python code and build powerful real-world AI applications with Azure OpenAI SDK.

This chapter will cover the following recipes:

- Connecting Azure OpenAI Service to your Python environment
- Creating a Python application with Azure OpenAI Service
- Using function calls with Azure OpenAI Service

## Technical requirements

To follow along with the recipes in this chapter, you will need to have the following:

- **An Azure subscription:** An Azure subscription is necessary to access Azure OpenAI Service. If you don't already have an Azure account, you can create one for free here: <https://azure.microsoft.com/free/ai-services/>.
- **Access to Azure OpenAI Service:** At present, access to this service is granted only upon application. You can apply for access to Azure OpenAI Service by completing the form at this link: <https://aka.ms/oai/access>.
- **An Azure OpenAI resource with a model deployed:** You learned how to create an Azure OpenAI resource and deploy models in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*. For more information about model deployment, refer to the resource deployment guide at this link: <https://learn.microsoft.com/azure/ai-services/openai/how-to/create-resource?pivots=web-portal>.
- **A Python environment:** You can use any Python environment, be it cloud-based or hosted locally on your machine. This chapter uses an environment with **Jupyter Notebook** for coding and installing the Azure OpenAI SDK.
- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter3>. The Chapter 3 folder contains all the notebooks for this chapter.

## Connecting Azure OpenAI Service to your Python environment

Setting up a Python environment is the cornerstone for any developer looking to unlock the many possibilities offered by the OpenAI SDK. This recipe will guide you through configuring a Python workspace tailored for interfacing with Azure OpenAI Service. Whether you're aiming to build sophisticated GenAI applications using Azure OpenAI Service or simply experimenting with state-of-the-art language models, this setup is an essential first step.

### Getting ready

For this recipe, you require an Azure OpenAI Service resource and a Python environment with Jupyter Notebook.

## How to do it...

Follow these steps to open a Jupyter Notebook, set up your Python environment, install the necessary Python packages, and connect your Python environment to an Azure OpenAI Service resource:

1. Let's begin by opening a Jupyter Notebook and creating a new Python notebook:
  - I. Launch Jupyter Notebook by typing `Jupyter Notebook` in your computer's search bar and selecting it from the search results.
  - II. Click **New** on the right side of your screen to open a dropdown.
  - III. Select **Python 3** under **Notebook** to create and open a Python notebook.
  - IV. Click **File** on the left side of your screen to open a dropdown.
  - V. Select **Rename** from the dropdown.
  - VI. Rename your notebook `Connect to OpenAI` and click **Rename**.

Now that you have created and renamed a Python notebook, you need to install the necessary Python packages to connect to Azure OpenAI Service.

2. In the first Python cell, type and execute the following command to install the `openai` Python package:

```
%pip install openai==1.24.0
```

### Note

OpenAI 1.24.0 is the latest version of this package at the time of writing this book. A newer version may be available by the time you read this book, and some functionalities may have changed.

3. Another important package to install is `tiktoken`. This package lets you count input and output tokens. Type and execute the following command:

```
%pip install tiktoken==0.6.0
```

4. Restart your Python kernel in order to access the two new packages. To do this, click **Kernel** and select **Restart** from the dropdown. You are now ready to connect to your Azure OpenAI Service resource.
5. Navigate to and open your Azure OpenAI Service resource by opening up a web browser, navigating to <https://portal.azure.com>, typing Azure OpenAI in the search bar, selecting **Azure OpenAI** from the search results, and clicking on the Azure OpenAI Service resource you created in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*.
6. Click the **Keys and Endpoint** option under the **Resource Management** section in the navigation menu.

The screenshot shows the Azure portal interface for managing an Azure OpenAI service named 'aoai-service'. The left sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under Resource Management, 'Keys and Endpoint' is selected, highlighted with a grey background. The main content area displays two API keys: 'KEY 1' and 'KEY 2', both represented by long strings of asterisks. Below the keys are fields for 'Location/Region' set to 'canadaeast' and 'Endpoint' set to 'https://aoai-service.openai.azure.com/'. At the top of the page, there are buttons for 'Regenerate Key1' and 'Regenerate Key2'. A note at the top right states: 'These keys are used to access your Azure AI services API. Do not share your keys. Store them securely—for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.'

Figure 3.1: Azure OpenAI Service Keys and Endpoint

7. Copy **KEY 1** and paste it into a Python cell as a variable:

```
openai_api_key = "your-key-1"
```

**Note**

You can use either **KEY 1** or **KEY 2**. They are interchangeable. All Azure resources have two keys to ensure seamless key rotation for production.

8. Copy **Endpoint** and paste it into a Python cell as a variable:

```
openai_api_url = "your-endpoint"
```

9. For production, save the key and endpoint as environment variables instead of normal variables:

```
import os
os.environ["AZURE_OPENAI_API_KEY"] = "REPLACE_WITH_YOUR_KEY_
VALUE"
os.environ["AZURE_OPENAI_ENDPOINT"] = "REPLACE_WITH_YOUR_
ENDPOINT"
```

### Note

**Environment variables** are variables that are set outside of a program so that their values are hidden from other people with access to the codebase.

Next, you need to retrieve the names of your deployed GPT-3.5 Turbo and GPT-4 models.

10. Navigate to and open your Azure OpenAI Service resource by repeating *Step 5* of this recipe.
11. Click **Go to Azure OpenAI Studio** on the top-right corner of your screen.
12. Click **Deployments** under the **Management** section in the navigation menu:

The screenshot shows the Azure OpenAI Studio interface. The left sidebar has sections for Playground, Chat, Completions, DALL-E, Assistants (Preview), Management, and Deployments (which is selected). The main area is titled 'Deployments' and contains a table of three entries:

Deployment name	Model name	M...	Deployme...	Capacity	Status	Model dep...
<a href="#">gpt-35-deployment</a>	gpt-35-turbo-16k	0613	Standard	120K TPM	<span>✓ Succeeded</span>	6/12/2024
<a href="#">embedding-deployment</a>	text-embedding-ada-002	2	Standard	120K TPM	<span>✓ Succeeded</span>	4/2/2025
<a href="#">gpt-4-deployment</a> <input checked="" type="checkbox"/>	gpt-4-32k	0613	Standard	30K TPM	<span>✓ Succeeded</span>	9/29/2024

Figure 3.2: Deployment name for the model

13. Set the names of your deployments to variables in a Python cell as shown here, replacing the placeholders with your GPT-4 and GPT-3.5 Turbo deployment names:

```
deployment_4 = "your-gpt-4-model"
deployment_3_turbo = "your-gpt-3.5-turbo-model"
```

- 
14. To connect your Python environment, you need to import the `AzureOpenAI` object from the `openai` package and create a client object with attributes for `api_base`, `api_key`, and `api_version`:

```
from openai import AzureOpenAI
client = AzureOpenAI(
    azure_endpoint = openai_api_url,
    api_key= openai_api_key,
    api_version="2024-02-01"
)
```

You have now set up your Python environment and connected it to your Azure OpenAI Service resource.

## How it works...

To access your Azure OpenAI Service resource in a Python environment, you need the `openai` Python package. Its version changes often. By installing a specific version, you can ensure that the rest of the code in this chapter will work.

This package requires you to specify the API version when creating a client object, and here you set the version to `2024-02-01`. The API version is always a date, and the latest non-preview version changes occasionally. You can find the latest version at the following link: <https://learn.microsoft.com/azure/ai-services/openai/reference#completions>.

You also need to specify your endpoint URL as well as your access key. These are found within your Azure OpenAI Service resource and can be stored either as normal or environment variables depending on your scenario.

## There's more...

Now that you have connected your Azure OpenAI Service resource to your Python environment, let's build our first script using prompts from *Chapter 2, Engineering Useful Prompts*. To begin, follow these steps:

1. Make sure you have all the variables set up from earlier in this recipe. You will reuse both your `openai` object and your `deployment_4` variable.
2. Create `system_message` and `user_message` instances with the following code:

```
system_message = """You are a translation assistant. A user
will provide you with a text to translate between triple pluses
(+++), a target language between triple stars (***) , and context
around the translation between triple ampersands (&&&)."""
user_message = """+++Dear customer, we regret to inform that the
item you ordered is currently-out-of-stock.+++ ***Spanish***<br>
&&&We are trying to send detailed yet polite e-mails to
customers.&&&"""
```

3. Create `messages` object as a list of dictionaries in Python. There are two types of messages in this case, indicated by `role`. The **system message** provides context and general instructions for the LLM, while the **user message** provides the statement or question to which the LLM directly responds:

```
messages = [  
    {"role": "system", "content": system_message},  
    {"role": "user", "content": user_message},  
]
```

#### Note

There's also an assistant message role. **Assistant messages** provide the response generated by the LLM based on the user message. You can assign them to help structure the communication between the user and the AI.

4. Obtain and save results with the following code, passing your GPT-4 deployment into the `model` attribute to select which of your deployed models is used for this chat:

```
results = client.chat.completions.create(  
    model=deployment_4,  
    messages=messages)
```

5. Process, save, and display the result with the following code:

```
results_text = results.choices[0].message.content  
results_text
```

6. You should see something similar to the following output:

```
'Estimado cliente, lamentamos informarle que el artículo que  
ordenó está actualmente agotado.'
```

Now you know how to run a simple script with your Azure OpenAI Service resource and Python. This method can be repeated for all the other prompt engineering recipes you learned in *Chapter 2, Engineering Useful Prompts*.

## See also

Here's a link to the GitHub repository for the `openai` package: <https://github.com/openai/openai-python>.

Here's a quickstart on using Azure OpenAI Service with Python: <https://learn.microsoft.com/azure/ai-services/openai/quickstart?tabs=command-line%2Cpython-new&pivots=programming-language-python>.

## Creating a Python application with Azure OpenAI Service

With your Python environment ready and the necessary libraries installed, the next step is to create a more complex Python application that will serve as your interface with Azure OpenAI Service. This application will be responsible for constructing and sending a series of prompts to Azure OpenAI Service and handling the responses. This recipe will guide you on how to code more complex Python applications that interact with Azure OpenAI Service.

### Getting ready

In addition to your Azure OpenAI Service resource and Python environment, you need to have installed the `openai` library, which is a critical component that provides direct access to OpenAI's API methods. Additionally, you need to have the `tiktoken` package installed. Lastly, ensure you have access to Azure OpenAI Studio in order to get the API key, endpoint URL, and deployed model names for your Azure OpenAI Service.

### How to do it...

To create a Python application to communicate with Azure OpenAI Service, follow these steps:

1. Launch Jupyter Notebook by typing `Jupyter Notebook` in your computer's search bar and selecting it from the search results.
2. Click **New** on the right side of your screen to open a dropdown.
3. Select **Python 3** under **Notebook** to create and open a Python notebook.
4. Click **File** on the left side of your screen to open a dropdown.
5. Select **Rename** from the dropdown.
6. Rename your notebook to `Advanced Python Applications` and click **Rename**.
7. In the first Python cell, import the following libraries with this code:

```
import json
import os
import tiktoken
import time
from openai import AzureOpenAI
```

You will need the following packages: `json` to manipulate machine-readable (JSON) output, `openai` to connect to Azure OpenAI Service, `os` to store environment variables, `tiktoken` to count variables, and `time` to help handle errors.

8. Next, in a new Python cell, set your Azure OpenAI Service API key and endpoint URL to environment variables with the following code, replacing the placeholders with your own values:

```
# Set the environment variables
os.environ["AZURE_OPENAI_API_KEY"] = "REPLACE_WITH_YOUR_KEY_
VALUE"
os.environ["AZURE_OPENAI_ENDPOINT"] = "REPLACE_WITH_YOUR_
ENDPOINT"
```

9. Create a `client` object with the endpoint URL, API key, and API version through this code:

```
client = AzureOpenAI(
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-02-01"
)
```

10. Set your deployed model names to variables within a new cell, replacing the placeholders with your GPT-4 and GPT-3.5 Turbo deployment names:

```
deployment_4 = "your-gpt-4-model"
deployment_3_turbo = "your-gpt-3.5-turbo-model"
```

11. Create a `messages` object as a list of dictionaries in Python with the following code. Note that `role` determines what type of message is being passed to the API, while `system`, `user`, `assistant`, and `content` determine the actual message:

```
messages=[

    {"role": "system", "content": "You are a helpful
assistant."},
    {"role": "user", "content": "Can I create a chatbot
using Azure OpenAI to chat with my own data?"},
    {"role": "assistant", "content": "Yes, you can use Azure
OpenAI and Azure cognitive search to create a chatbot."},
    {"role": "user", "content": "Can you provide more
information about Azure cognitive search?"}
]
```

12. Obtain and save the results with the following code, passing your GPT-4 deployment into the `model` attribute to select which of your deployed models will be used for the chat:

```
results = client.chat.completions.create(
    model=deployment_4,
    messages=messages)
```

13. Process, save, and display the result with the following code:

```
results_text = results.choices[0].message.content  
print(results_text)
```

14. After running the preceding code, you should see output similar to *Figure 3.3*:

Azure Cognitive Search is an AI-powered cloud search service available on Microsoft Azure. It uses machine learning to understand and find specific content within your data.

Here are few things you can do with it:

1. Full-Text Search: It uses AI to comb through structured or unstructured text for a much more efficient search.
2. Semantic Search: It understands the context, not just the keywords, for an improved search experience.
3. Built-in Indexers: It automatically brings in information from common databases and file systems.
4. Scaling: You can easily scale based on your needs while it handles everything on the back-end.

Figure 3.3: Output generated by the GPT-4 model

## How it works...

In this recipe, you created a Python application within Jupyter Notebook to use the Azure OpenAI client from the `openai` library. By passing your endpoint and key to the Azure OpenAI object, you establish authenticated communication with Azure OpenAI Service. The code provided in this recipe sets up a conversation context with the LLM, demonstrating how to send prompts and receive responses. The `model` parameter specifically references the deployment name of your chosen model.

Now, let's break down the parameters within the `messages` array that you used in the Jupyter Notebook example:

- **role:** The `role` parameter specifies the participant in the conversation. In the context of the Azure OpenAI API, there are typically three roles:
  - **system:** This role is used for setting up instructions or context for the AI. It's like providing a directive to the AI on how it should behave or what role it should play. For example, the message `You are a helpful assistant.` tells GPT to take on a user-friendly, supportive, and problem-solving mindset. This message instructs GPT to prioritize being helpful and providing accurate, relevant, and timely information.
  - **user:** This role represents the human user who is interacting with the AI. Messages under this role are queries or statements from the user to which the AI needs to respond.
  - **assistant:** This role indicates the responses from the AI itself. When simulating a conversation or drafting responses, you can include this role to provide pre-defined AI responses, guiding the model's future responses.

- **content:** The `content` parameter holds the actual text of the message being sent to or from the AI. This is the substance of the conversation—the prompts from the user and the responses from the AI. Each content is associated with a role, forming a coherent dialogue or exchange of information.

By structuring the messages in this way—with a system message, multiple user messages, and an assistant message—you create a detailed script for the conversation that the AI can follow, leading to a more natural and context-aware interaction. This process underscores the LLM’s conversational capabilities, especially in maintaining context over a sequence of exchanges.

## There's more...

In *Chapter 2, Engineering Useful Prompts*, you learned how to create machine-readable output by specifying your output as JSON. Here's a guide on how to process JSON data with the Azure OpenAI SDK, along with how to handle errors and count tokens to keep track of cost:

1. Repeat *Steps 1–10* from the *How to do it...* section of this recipe, but now name your notebook `AI Output Processing and Error Handling`.
2. Create the following system message from *Chapter 2, Engineering Useful Prompts*, which outputs JSON:

```
system_message = """You are multi-language translation
assistant. A user will provide you with a text to translate
between triple pluses (+++) and context around the translation
between triple ampersands (&&&). You will translate the text
into Spanish, French, German, and Japanese. Once you do that,
you will output the translation in JSON format.

To accomplish this task, first read the translation context, then
translate the text into Spanish, French, German, and Japanese in
that order, then output the following JSON format:
{
    "Spanish": "",
    "French": "",
    "German": "",
    "Japanese": ""
}
Only output the JSON and ensure that it is the correct
format.""""
```

3. Create the following user message, which shows a message from an employee to his manager:

```
user_message = """+++Henry, Tomorrow I will be late for work due
to unforeseen circumstances. My apologies, Mark.+++ &&&This is
an e-mail from a worker to his boss.&&&""""
```

**Note**

Using triple quotes instead of single quotes helps ensure that your code does not have errors related to multi-line strings or strings that contain quotation marks or special characters. Triple quotation marks allow for both of these without needing escape characters.

4. Create messages object as a list of dictionaries in Python with the code shown here:

```
messages = [  
    {"role": "system", "content": system_message},  
    {"role": "user", "content": user_message},  
]
```

5. Create a Python function to count tokens with the `tiktoken` package:

```
# Function to get token count for Open AI  
def num_tokens_from_string(string: str) -> int:  
    """Returns the number of tokens in a text string."""  
    enc = tiktoken.encoding_for_model("gpt-4")  
    num_tokens = len(enc.encode(string))  
    return num_tokens
```

**Note**

You can use `tiktoken` to count tokens for models other than GPT-4. See <https://github.com/openai/tiktoken>.

6. Count the number of input tokens with this code:

```
num_tokens_from_string(system_message + user_message)
```

7. You should see that the output from this function is 176 tokens. You can use this code to keep track of the token count for cost optimization purposes. At the time of writing, GPT-4 costs \$0.03 per 1,000 input tokens and \$0.06 per 1,000 output tokens.

**Note**

`tiktoken` provides an estimation of the token count rather than the actual count. The approximation is close but may not be exact due to the complexities of tokenization. Furthermore, different versions of `tiktoken` provide slightly different estimations of token count.

8. Run the following code to get a response for timeout errors. This code will retry getting a chat completion up to three times before it errors out, waiting 5 seconds between attempts:

```
attempts = 0
max_attempts = 3
success = False

while not success and attempts < max_attempts:
    try:
        results = client.chat.completions.create(
            model= deployment_4,
            messages=messages
        )
        success = True
    except TimeoutError as e:
        print("Timeout error occurred, trying again...")
        attempts += 1
        time.sleep(5)

if not success:
    print("Failed to get a response after several retries.")
```

9. Save the response to a JSON option with the following code:

```
results_json = json.loads(results.choices[0].message.content)
```

10. Manipulate the json object to see the Japanese translation:

```
results_json['Japanese']
```

11. Examine the output. It should look similar to this:

```
'亨リー、明日は予見できない状況のため仕事が遅れます。申し訳ありません、マーク。'
```

You have now successfully created and processed JSON output with Azure OpenAI Service in Python.

## See also

Here are links to documentation giving you more information about the OpenAI Python SDK and how to generate text using Azure OpenAI Service:

- Here's a quickstart on using GPT-3.5 Turbo and GPT-4 with Azure OpenAI Service and the OpenAI Python SDK: <https://learn.microsoft.com/azure/ai-services/openai/chatgpt-quickstart?tabs=command-line%2Cpython-new&pivots=programming-language-python>.

- Here's a quickstart on generating text using Azure OpenAI Service and the OpenAI Python SDK: <https://learn.microsoft.com/azure/ai-services/openai/quickstart?tabs=command-line%2Cpython-new&pivots=programming-language-python>.
- Here's a link to Azure OpenAI Service pricing: <https://azure.microsoft.com/pricing/details/cognitive-services/openai-service/>.
- Here's a link to documentation on how to count tokens with `tiktoken`: [https://github.com/openai/openai-cookbook/blob/main/examples/How\\_to\\_count\\_tokens\\_with\\_tiktoken.ipynb](https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb).

## Using function calls with Azure OpenAI Service

The latest versions of GPT-3.5 Turbo and GPT-4 offer an advanced ability to handle function calls within prompts—a feature designed to streamline complex tasks. These models are engineered to understand when and how to invoke a function based on the prompt’s context. This recipe will guide you through the process of using function calls in Azure OpenAI Service with the Azure OpenAI SDK. You will use Azure OpenAI Service to invoke a Python function that converts meters to inches through a conversational chat.

When you embed functions in your requests to Azure OpenAI Service, the model decides whether to execute these based on the input provided. If the model decides to use a function, it prepares the API call and organizes the data to match the functions you've set up. However, even though the model comes up with the plan for these calls, you're the one who must run them. This means that you as a developer are always in charge of executing the functions.

### Note

Function calling is still in preview, and the way to call functions has changed repeatedly within the last year as of writing this book. These instructions use the `tool_choice` parameter, compatible with the latest preview versions of the GPT models at the time of writing.

## Getting ready

In addition to your Azure OpenAI Service resource and Python environment, you need to have installed the `openai` library, which is a critical component that provides direct access to the API methods of OpenAI. Ensure you have access to Azure OpenAI Service in order to get the API key, endpoint URL, and deployed model names for your Azure OpenAI Service resource.

## How to do it...

To invoke functions with Azure OpenAI Service in your Python application, follow these steps:

1. Launch Jupyter Notebook by typing `Jupyter Notebook` in your computer's search bar and selecting it from the search results.
2. Click **New** on the right side of your screen to open a dropdown.
3. Select **Python 3** under **Notebook** to create and open a Python notebook.
4. Click **File** on the left side of your screen to open a dropdown.
5. Select **Rename** from the dropdown.
6. Rename your notebook to `OpenAI Function Calling` and click **Rename**.
7. In the first Python cell, import the following three libraries with this code:

```
import json
import os
from openai import AzureOpenAI
```

You will need `json` to manipulate machine-readable (JSON) output, `openai` to connect to Azure OpenAI Service, and `os` to store environment variables.

8. Next, in a new cell, set your Azure OpenAI Service API key and endpoint URL to environment variables with this code, replacing the placeholders with your own values:

```
# Set the environment variables
os.environ["AZURE_OPENAI_API_KEY"] = "REPLACE_WITH_YOUR_KEY_
VALUE"
os.environ["AZURE_OPENAI_ENDPOINT"] = "REPLACE_WITH_YOUR_
ENDPOINT"
```

9. Create a `client` object with the endpoint URL, API key, and API version, through the following code, noticing that the value for `api_version` is different from before:

```
client = AzureOpenAI(
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-03-01-preview"
)
```

10. Create a Python function to convert meters to inches with this code:

```
# Function to convert meters to inches
def convert_meters_to_inches(meters):
    """Convert meters to inches"""
    inches = round(meters * 39.37, 3) # 1 meter is
    approximately 39.37 inches
    return json.dumps({"meters": meters, "inches": inches})
```

11. Test the function with the following code:

```
# Example usage of the function
meters_input = 1.5 # Example input in meters
conversion_result = convert_meters_to_inches(meters_input)
print(f"Conversion result: {conversion_result}")
```

You should see the following output:

```
Conversion result: {"meters": 1.5, "inches": 59.055}
```

12. Create the following system message:

```
system_message = """You are a helpful scientific assistant."""
```

13. Create the following user message on converting meters to inches:

```
user_message = """The distance from the Earth to the moon is
roughly 384,000,000 meters. What is this distance in inches?"""
```

14. Set your deployed model names as variables within a new cell, replacing the placeholders with your GPT-4 and GPT-3.5 Turbo deployment names:

```
deployment_4 = "your-gpt-4-model"
deployment_3_turbo = "your-gpt-3.5-turbo-model"
```

15. Create messages object as a list of dictionaries in Python with the code shown here:

```
messages = [
    {"role": "system", "content": system_message},
    {"role": "user", "content": user_message},
]
```

16. Create a tools object that Azure OpenAI Service can call:

```
tools = [{"type": "function",
          "function": {
              "name": "convert_meters_to_inches",
              "description": "Converts meters to inches",
              "parameters": {
                  "type": "object",
                  "properties": {
                      "meters": {
                          "type": "number",
                          "description": "The length in meters
to be converted to inches",
                      }
                  },
                  "required": ["meters"]
              }
          }
      }]
```

17. Run the chat with the following code using your deployed GPT-4 model, passing in the `tools` object and setting `tool_choice` to `auto` to let the model automatically decide whether it uses the function or not:

```
response = client.chat.completions.create(  
    model=deployment_4,  
    messages=messages,  
    tools=tools,  
    tool_choice="auto"  
)
```

18. Instead of returning a message, the response returned a function for you to use along with a value to pass through it. You can access them with this code:

```
# Access the first item in the list to get the tool  
func = response.choices[0].message.tool_calls[0]  
  
# Get the function name  
function_name = func.function.name  
  
# Parse the JSON in the 'arguments' and extract the 'meters'  
# value  
arguments = json.loads(func.function.arguments)  
meters = arguments['meters']
```

19. Now, you can pass the value into the function to get your answer with this code:

```
func_to_call = globals()[function_name]  
func_to_call(meters)
```

20. You should see the following output:

```
'{"meters": 384000000, "inches": 15118080000.0}'
```

## How it works...

The code provided in this recipe demonstrates how to integrate custom Python functions—in this case, a function to convert meters to inches—into a conversation with Azure OpenAI Service.

Let's break down its key components and how they work together:

- **Initializing the Azure OpenAI client:** The client is created using the `AzureOpenAI` class, initialized with your Azure OpenAI Service endpoint and API key. This client acts as the gateway for sending requests to and receiving responses from Azure OpenAI Service.
- **Defining a custom function (`convert_meters_to_inches`):** This function takes a numerical input representing meters and converts it to inches. The conversion is performed by multiplying the meters value by 39.37, as this is the conversion factor between meters and inches. The function returns a JSON string that includes both the original meters value and the calculated inches, ensuring that the output is easy to interpret.
- **Crafting the conversation (`messages`) and tools (`tools`):** The `messages` array starts the conversation with a user prompt, simulating a request for unit conversion. The `tools` array defines available functions that the model can *call*. In this scenario, it describes the `convert_meters_to_inches` function, including its name, a brief description, and its parameters. This setup informs the model that the function exists and how the function should be used.
- **Sending the initial request to the model:** The `chat.completions.create` method sends the initial conversation and available tools to the model. This includes the user's prompt and the `convert_meters_to_inches` function as a tool that can be used in the conversation. The model then processes this information and determines whether to call the specified function or not based on the context provided.
- **Processing model responses and function calls:** After sending the request, the model's response is checked for any function calls. If the model has identified a need to call a function based on the conversation's context, it will indicate so in its response.

If a function call is indicated, the corresponding function is executed using the arguments provided by the model. This step involves pulling the function arguments, the function name, and the values to pass into the function from the JSON response. You can then call the `convert_meters_to_inches` function with these arguments using normal Python and then process the output.

### Note

You can also make a follow-up request that incorporates the function response. To do this, you would take the output of your function and send it to the LLM along with the original prompt, instructing it explicitly to use your function output in its response.

This process illustrates how custom functions can be seamlessly integrated into interactions with Azure OpenAI Service, ensuring complex and dynamic conversations that can respond intelligently to user requests with real-world actions and data processing.

## See also

Here are links to documentation giving you more information about how to use function calling with Azure OpenAI Service:

- Here's how to use function calling with Azure OpenAI Service (Preview): <https://learn.microsoft.com/azure/ai-services/openai/how-to/function-calling?tabs=python-new>.
- Learn more about working with functions in Azure OpenAI here: [https://github.com/Azure-Samples/openai/blob/main/Basic\\_Samples/Functions/working\\_with\\_functions.ipynb](https://github.com/Azure-Samples/openai/blob/main/Basic_Samples/Functions/working_with_functions.ipynb).

# 4

## Chatting with Your Own Data Using RAG

In this chapter, you will learn how to chat with your own data using **retrieval-augmented generation (RAG)**, a technique that involves interacting with your own datasets through conversation. By utilizing **Azure OpenAI Service** and **Azure AI Search**, you'll learn how to generate meaningful responses from your data, making it an interactive and engaging experience. This technique is particularly useful for extracting insights and understanding from large datasets, as it transforms an otherwise complex data analysis task into a simple chat conversation.

This chapter focuses on setting up the necessary Azure services, deploying appropriate models, preparing your data, and configuring your AI model for data retrieval and search. You'll also learn how to ask questions relevant to your data and interpret the responses. By the end of this chapter, you will be capable of utilizing Azure OpenAI Service and Azure AI Search to interact with your own data, as well as interpret and understand the information retrieved from these interactions.

This chapter will cover the following recipes:

- Getting started with Azure AI Search
- Creating vector embeddings with Azure OpenAI Studio
- Chatting with your own data using Azure OpenAI Studio
- Creating vector embeddings with the Azure OpenAI SDK
- Chatting with your own data using the Azure OpenAI SDK

## Technical requirements

To successfully follow along with the recipes in this chapter, ensure that you have the following:

- An active Azure account, either free or paid.
- Access to Azure OpenAI Service within your Azure subscription. Ensure you are assigned at least the **Cognitive Services Contributor** role for the Azure OpenAI resource. If necessary, you can apply for access to Azure OpenAI Service at <https://aka.ms/oai/access>.
- Python 3.8 or a later version installed on your system.
- The necessary Python libraries installed, including `openai` and `pandas`.
- Visual Studio Code with the Jupyter extension installed for running the code and visualizing the results.
- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter4>. The Chapter 4 folder contains all the notebooks for this chapter.

### Note

For more details on role assignment within Azure AI services, particularly for OpenAI services, you might find the following resource helpful: <https://learn.microsoft.com/azure/ai-services/openai/how-to/role-based-access-control>

## Getting started with Azure AI Search

In this recipe, you will go through a step-by-step process of using the Azure portal to create an Azure AI Search service, which provides a powerful search engine for vector search, full text, and hybrid search over a search index.

By the end of this recipe, you will have learned how to set up an Azure AI Search service using the Azure portal. This includes steps such as logging in to your account, searching for Azure AI Search, filling in the necessary service details, and finalizing the creation of your Azure AI Search service.

### Getting ready

In this recipe, it is necessary for you to have an Azure account, either free or paid.

### How to do it...

Follow the steps below to create an Azure AI Search service using the Azure portal:

1. Open the Azure portal at <https://portal.azure.com> and log in to your account. Click on **Create a resource**:

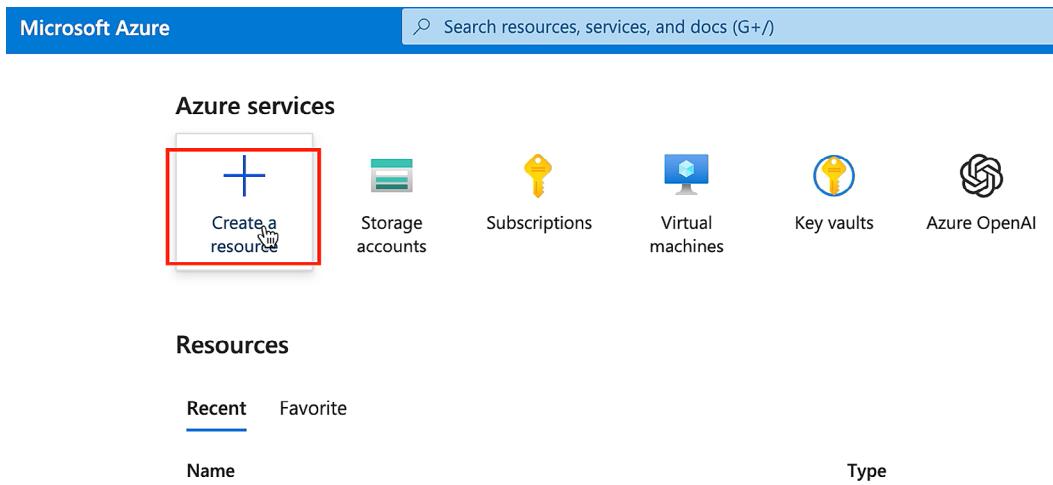


Figure 4.1: Azure portal

2. Then, search for Azure AI Search, and click the **Create** button:

Showing 1 to 20 of 117 results for 'Azure AI Search'. [Clear search](#)

This screenshot shows the search results for 'Azure AI Search'. On the left, there's a card for 'Azure AI Search' with a cloud icon, the name, a detailed description mentioning it's an AI-powered cloud search service for mobile and web app development (formerly Azure Cognitive Search), and a 'Create' button at the bottom which is also highlighted with a red box and has a cursor icon pointing at it. To the right, there's another card for 'BA Insight for Azure AI Search' with a 'Free trial' button, followed by other search results like 'BA-Insight - Global HQ (Boston)' and 'SaaS'. At the bottom of the page, there are 'Subscribe' and 'Create' buttons.

Figure 4.2: Azure AI Search – Create button

3. If you haven't created a resource group yet, select **Create new** and enter the name `rg-aoabook-001`. Then, click on **OK**:

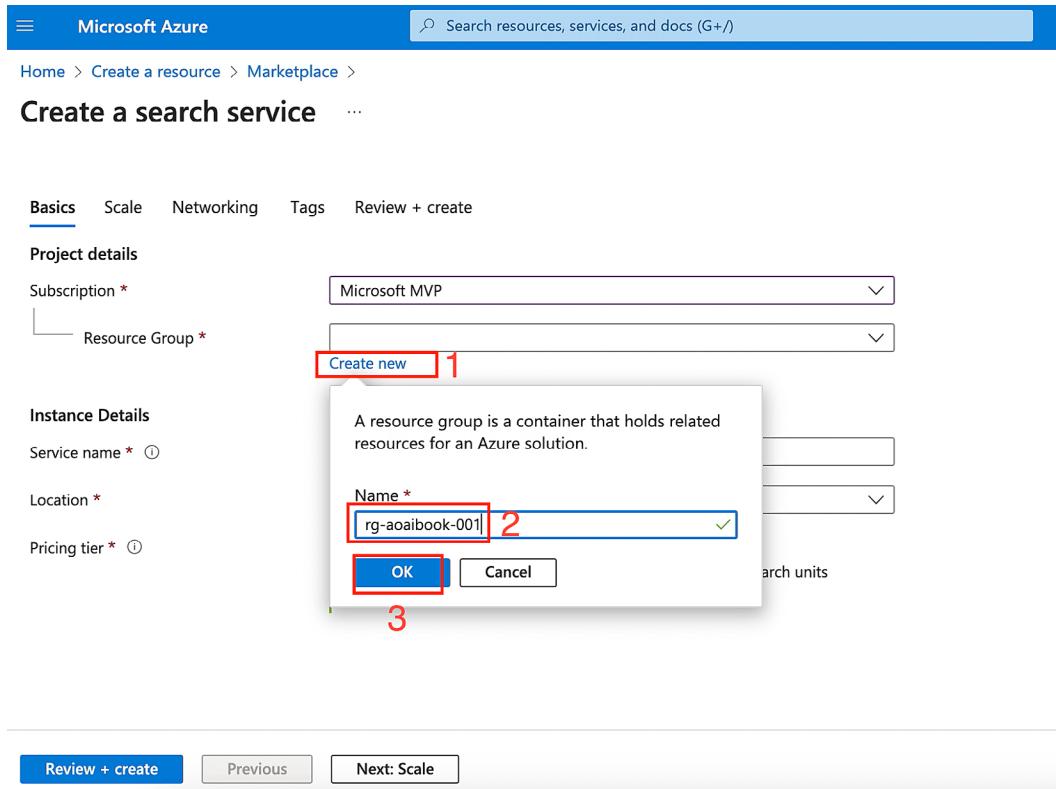


Figure 4.3: New resource group

4. Fill in the appropriate details, such as **Service name**, **Location**, and **Pricing tier**, as shown in *Figure 4.4*. Now, click on the **Review + create** button to create the Azure AI Search service:

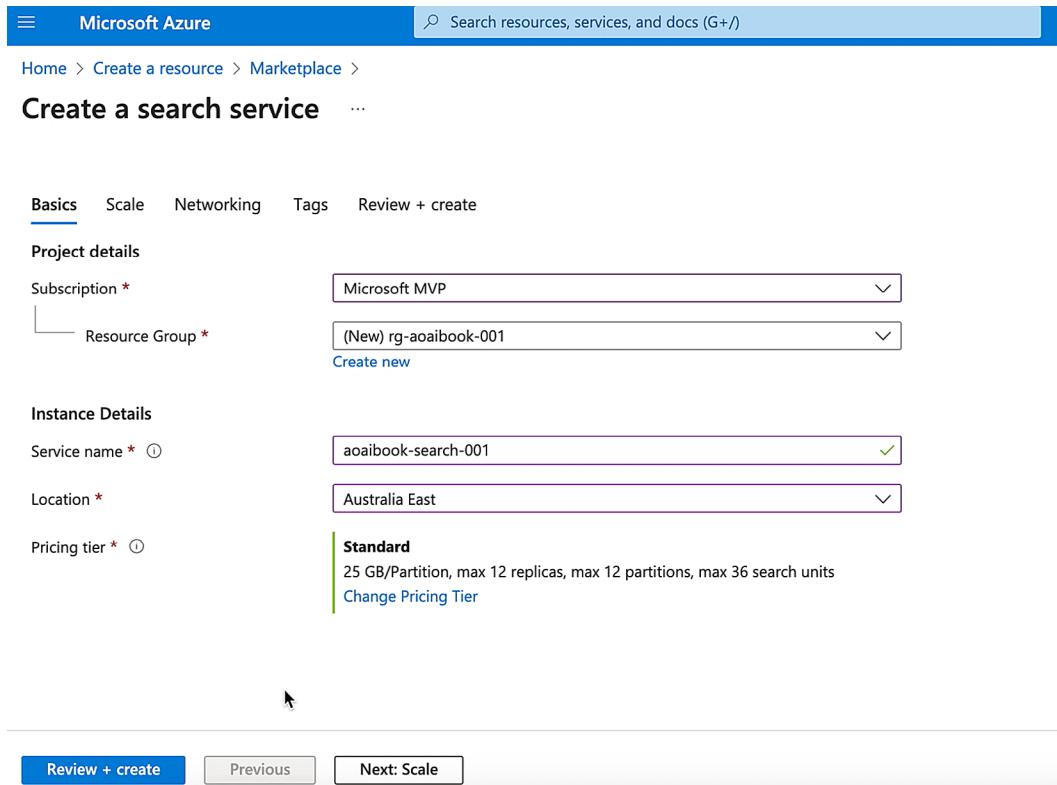


Figure 4.4: Creating Azure AI Search service

5. It will take a few seconds to create the new Azure AI Search service. You can find the Azure AI Search service you've created in the **rg-aoabook-001** resource group:

The screenshot shows the Azure AI Search service blade for the 'aoabook-search-001' resource group. The left sidebar lists various management options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Search management (Indexes, Indexers, Data sources, Aliases, Skillsets, Debug sessions, Settings, Semantic ranker), and Get started.

The main content area displays the following service details:

Setting	Value
Resource group (move)	rg-aoabook-001
Location (move)	Australia East
Subscription (move)	
Subscription ID	
Status	Running
Tags (edit)	Add tags
Url	https://aoabook-search-001.search.windows.net
Pricing tier	Standard
Replicas	1 (No SLA)
Partitions	1
Search units	1

Below the details, there's a callout with the text: "Build a full-text search experience with AI and semantic ranker". A smaller note below it says: "Get started building a full-text search experience and learn how to integrate with your custom applications and other Azure services."

Figure 4.5: Azure AI Search service

## How it works...

Azure AI Search is a service crucial for RAG patterns powered by LLMs like GPT-4. Despite these models' impressive language understanding, they have limitations, including outdated public knowledge and lack of access to internal knowledge. To overcome these, RAG patterns employ a combination of retrieval and generation, extracting information from a dataset to generate responses.

In a RAG pattern, the Azure AI Search service uses vectors, which are numerical representations of text, to enhance information retrieval and response generation. Each piece of text is transformed into an embedding vector, stored alongside the text in the Azure database. This method allows the service to quickly identify relevant documents based on the similarity of their vectors to the query vector. For complex queries, this vector-based approach is particularly effective as it goes beyond simple keyword matching, enabling the service to understand and respond to the semantic nuances of the query. Thus, vectors are pivotal for extracting pertinent information from large datasets and generating detailed responses to sophisticated queries.

## Creating vector embeddings with Azure OpenAI Studio

In this recipe, you will learn how to create a new Azure OpenAI instance and generate vector embeddings using **Azure OpenAI Studio**.

By the end of this recipe, you will have a hands-on understanding of how to deploy the Azure OpenAI instance, create vector embeddings, and manage your data using Azure OpenAI Studio. The topics covered will include deploying models, adding data, and configuring the search type for effective data management.

### Getting ready

Before starting, ensure you have an Azure subscription, access to Azure OpenAI in your subscription, and are assigned at least the Cognitive Services Contributor role for the Azure OpenAI resource. If you need access to Azure OpenAI, you can fill out the following form to request access: <https://aka.ms/oai/access>.

### How to do it...

Let's learn how to create a new Azure OpenAI instance and create vector embeddings with Azure OpenAI Studio:

1. Go to the Azure portal and navigate to the resource group **rg-aoaibook-001**. Click on **Create** to open the **Marketplace** page:

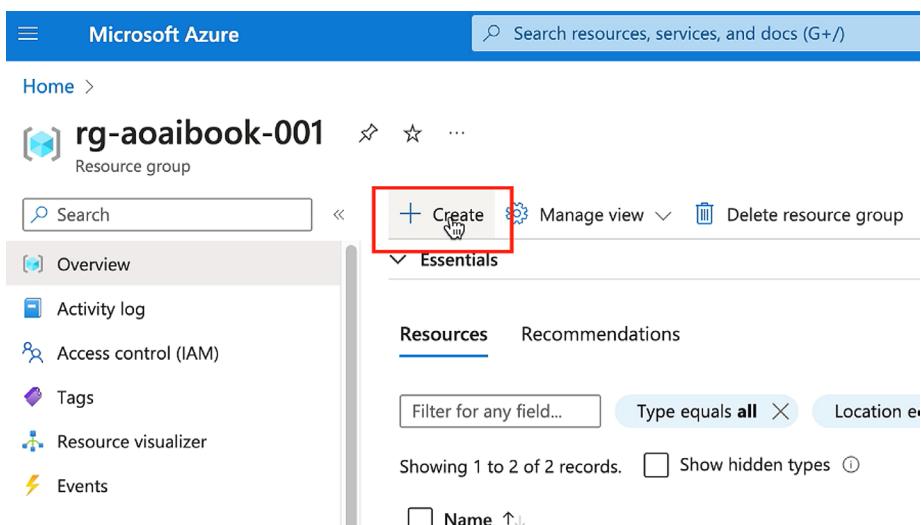


Figure 4.6: The resource group

2. Then, search for Azure OpenAI on the **Marketplace** page and click on **Create** to open the **Create Azure OpenAI** page:

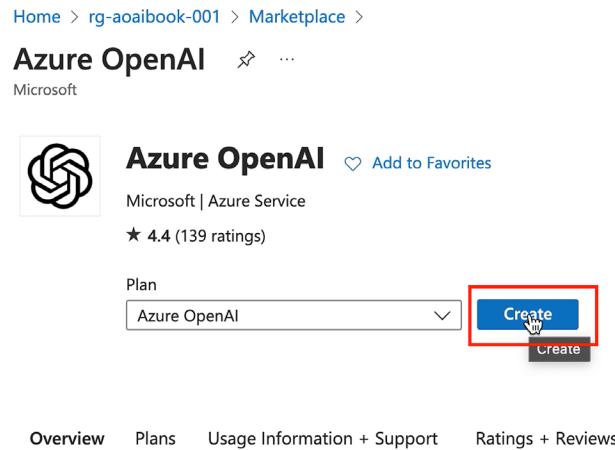


Figure 4.7: Creating an Azure OpenAI instance

3. Select your subscription and resource group and fill in the appropriate details, such as instance name, region, and pricing tier:

The screenshot shows the "Create Azure OpenAI" wizard. At the top, there's a Microsoft Azure header bar with a search bar. Below it, the breadcrumb navigation shows: Home > rg-aoabook-001 > Marketplace > Azure OpenAI > Create Azure OpenAI. There are four tabs at the top of the form: ① Basics (which is underlined), ② Network, ③ Tags, and ④ Review + submit. The "Basics" tab is active. The main content area has a heading "Enable new business solutions with OpenAI's language generation capabilities powered by GPT-3 models. These models have been pretrained with trillions of words and can easily adapt to your scenario with a few short examples provided at inference. Apply them to numerous scenarios, from summarization to content and code generation." Below this is a "Learn more" link. The "Project Details" section contains fields for "Subscription" (set to "Microsoft MVP") and "Resource group" (set to "rg-aoabook-001"). The "Instance Details" section contains fields for "Region" (set to "Australia East") and "Name" (set to "aoabook-oai-001"). At the bottom, there are "Previous" and "Next" buttons.

Figure 4.8: Create Azure OpenAI page

4. In the **Network** tab, keep the default settings unless you need specific configurations. Similarly, in the **Tags** tab, you don't need to add any tags unless you want to categorize your resource. After these, in the **Review + submit** tab, verify all settings and click on **Create** to finalize your resource setup:

**Create Azure OpenAI** ...

Basics Network Tags **Review + submit** 4

[View automation template](#)

**TERMS**

By clicking "Create", I (a) agree to the legal terms and privacy statement(s) associated with the Market listed above; (b) authorize Microsoft to bill my current payment method for the fees associated with the same billing frequency as my Azure subscription; and (c) agree that Microsoft may share my and transactional information with the provider(s) of the offering(s) for support, billing and other transactional activities. Microsoft does not provide rights for third-party offerings. See the [Azure Marketplace Term](#) details.

**Basics**

Subscription	Microsoft Azure
Resource group	rg-aoaibook-001
Region	Australia East
Name	aoaibook-oai-001
Pricing tier	Standard S0

[Previous](#) [Next](#) **Create**

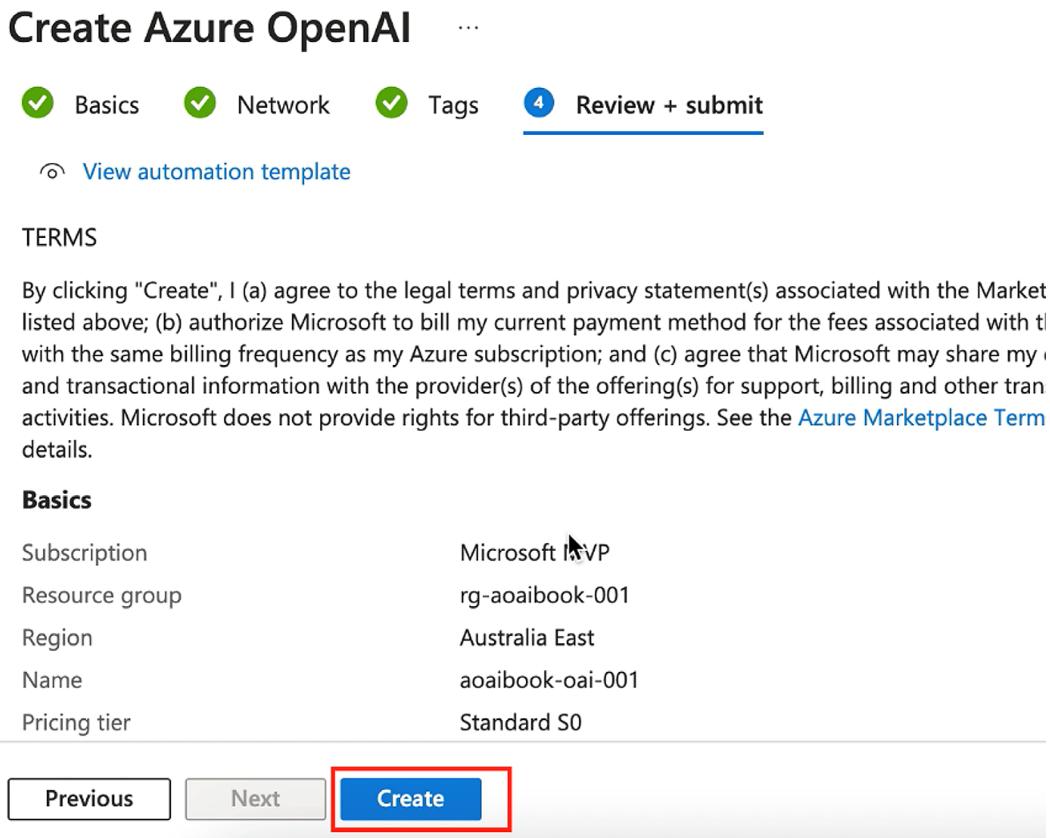


Figure 4.9: Creating an Azure OpenAI instance – review page

5. It will take some time to deploy the service; you will get a notification in the Azure portal once the process has succeeded:

The screenshot shows the Azure portal interface with a deployment in progress. At the top, there are buttons for Delete, Cancel, Redeploy, Download, and Refresh. Below that, a section titled "Deployment is in progress" displays deployment details: Deployment name : Microsoft.CognitiveServicesOpenAI-202403... Start time : [redacted] Subscription : Microsoft MVP Correlation ID : [redacted] Resource group : rg-aoaibook-001. A dropdown menu labeled "Deployment details" is open. Below this, a table shows deployment resources, which is currently empty.

Resource	Type	Status
There are no resources to display.		

Figure 4.10: Deployment process

6. Go to the Azure OpenAI Studio website at <https://oai.azure.com/> and log in using the same credentials that created the Azure OpenAI resource in the previous steps. Once you're logged in, choose the right directory, Azure subscription, and Azure OpenAI resource that you've just created. Then, click on **Use resource**:

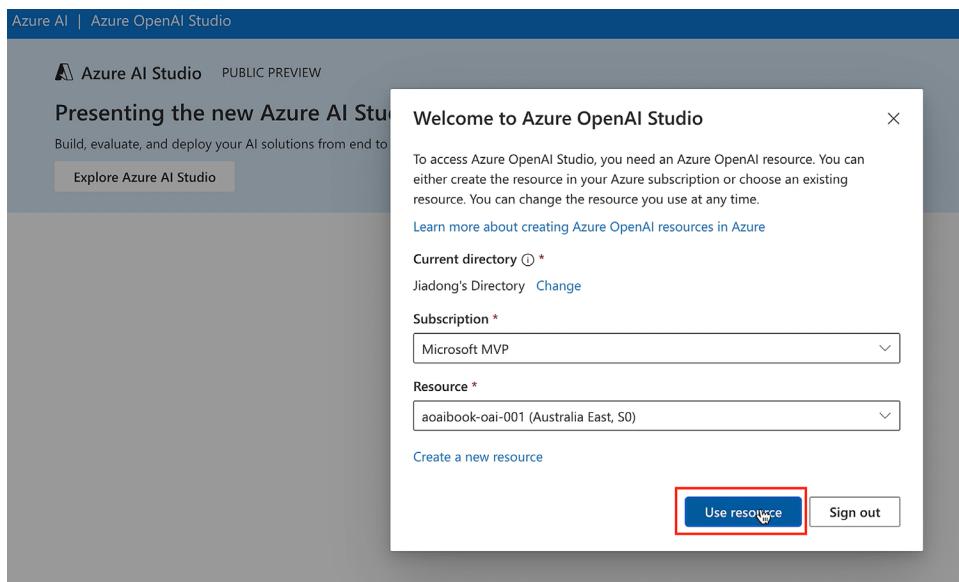


Figure 4.11: Logging in to Azure OpenAI Studio

7. Before you can start generating text or creating vector embeddings, it's necessary to deploy models. A variety of models are available for deployment within Azure OpenAI Studio.
8. Under the **Management** section, choose **Deployments**. Click on **Create new deployment** and configure the fields shown in *Figure 4.12*. For **Select a model**, choose **gpt-4** and name it **aoaibook**. Then, click **Create**. By doing this, you can deploy a GPT-4 model:

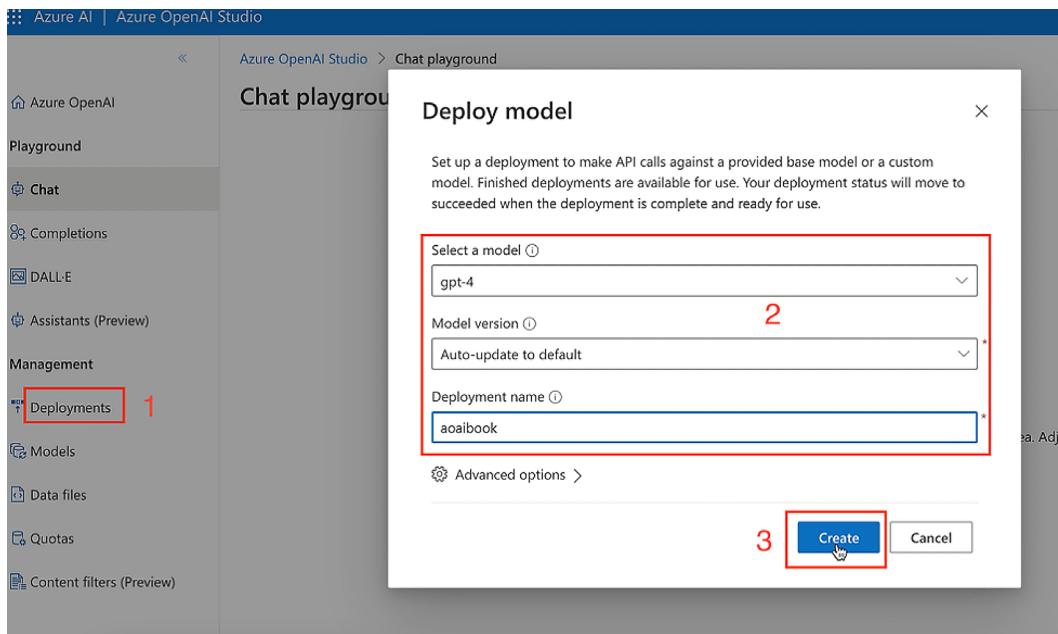


Figure 4.12: Deploying a GPT-4 model

9. You also need to deploy a `text-embedding-ada-002` model in Azure OpenAI Studio, as shown in *Figure 4.13*, to generate new embeddings:

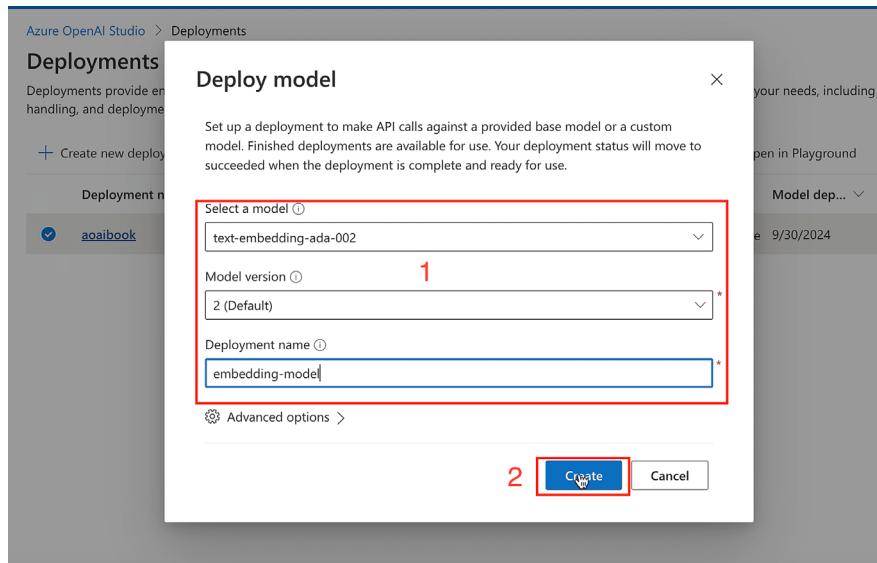


Figure 4.13: Deploying an embeddings model

10. Go to the home page of Azure OpenAI Studio and select the **Bring your own data** tile to start the **Add data** wizard:

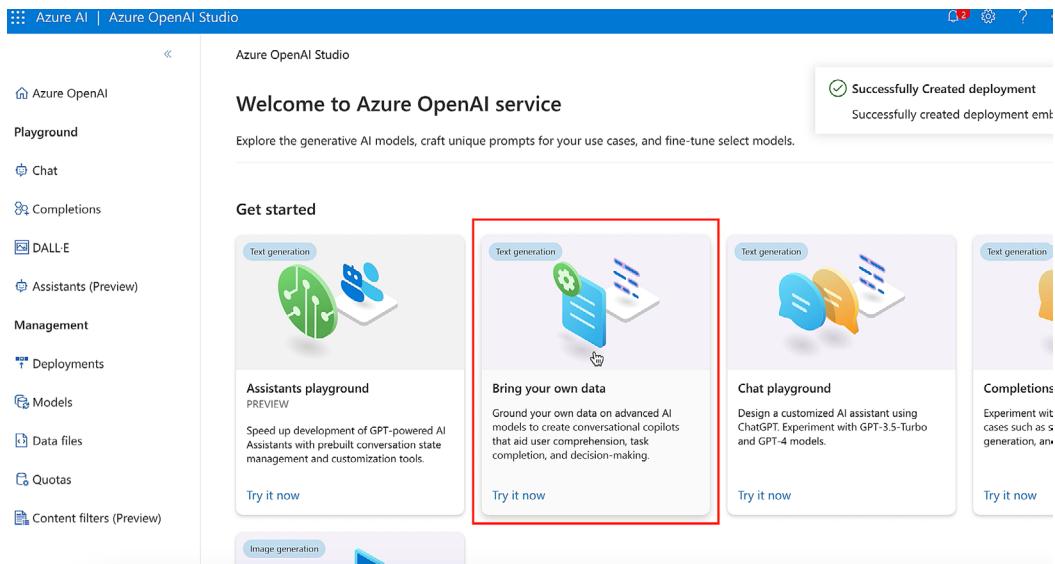


Figure 4.14: Bringing your own data

11. In the **Data source** section, choose **Upload files (preview)** from the dropdown list as the data source:

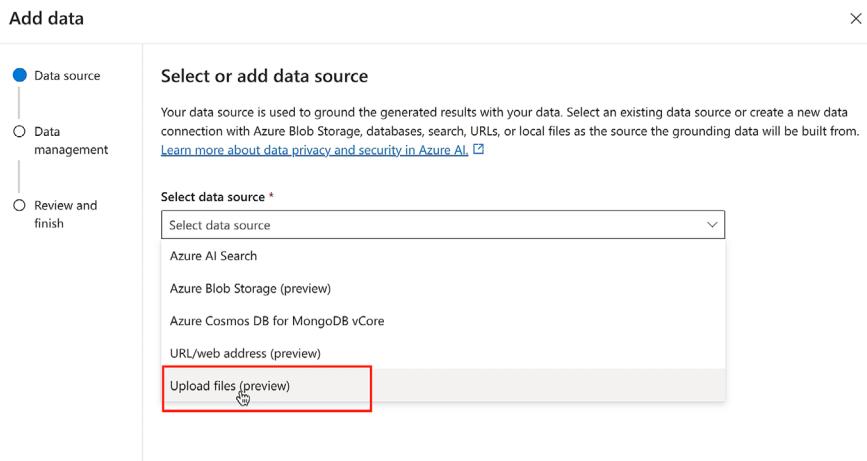


Figure 4.15: Selecting the data source

12. The **Upload files** section will pop up automatically. You need to select an Azure Blob storage resource to store your data. If you haven't created a Blob storage resource yet, you can click on **Create a new Azure Blob storage resource** to create a new one:

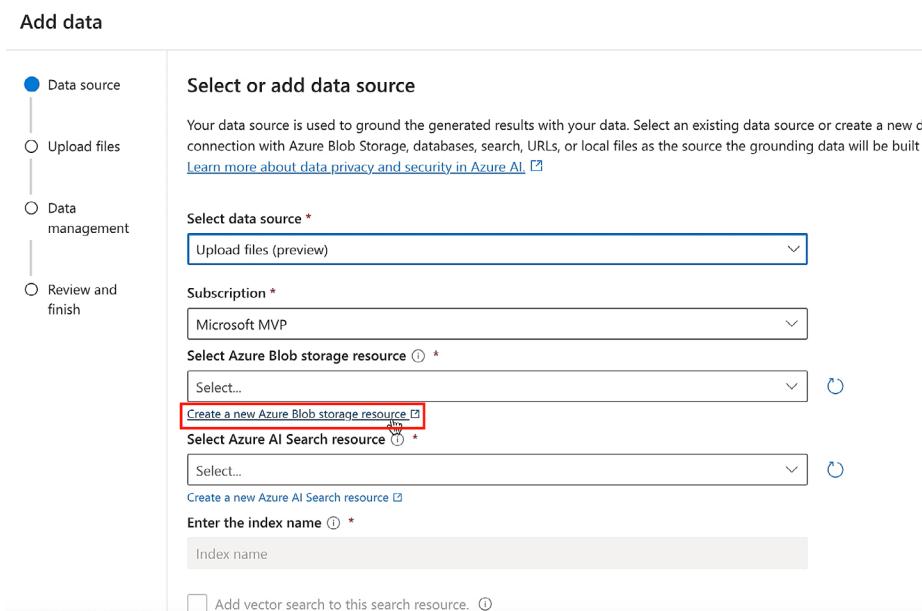


Figure 4.16: Uploading data – Creating a new Azure Blob storage resource

13. Once the Azure Blob storage resource is created, you need to enable **Cross-origin resource sharing (CORS)** for it by selecting **Turn on CORS** to make it accessible to Azure OpenAI:

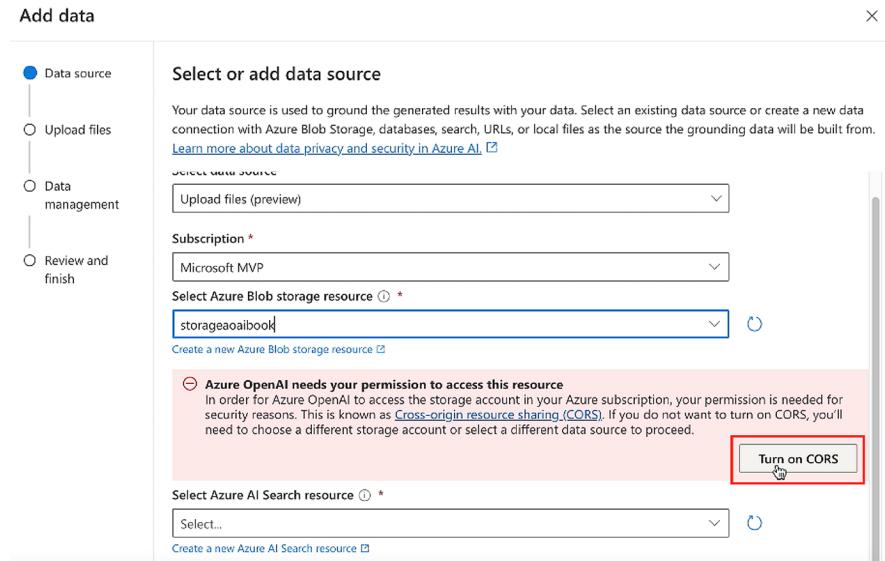


Figure 4.17: Turning on CORS

14. Next, select an Azure AI Search resource. From the dropdown list, you can select the Azure AI Search resource that you created in the previous section:

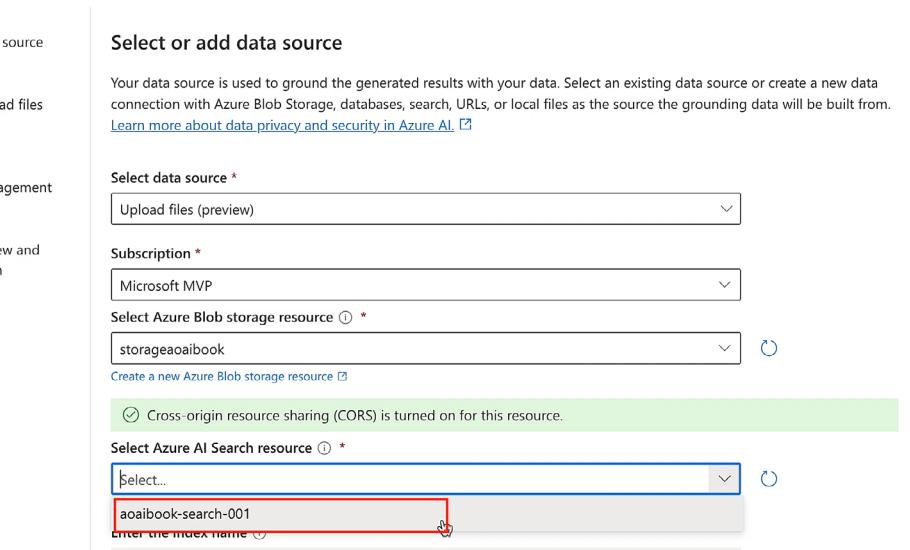


Figure 4.18: Selecting the Azure AI Search resource

15. Set the index name as `book-data-index`. Check the **Add vector search to this search resource** option, and select the embedding model you just deployed in the **Embedding model** section. Click **Next**:

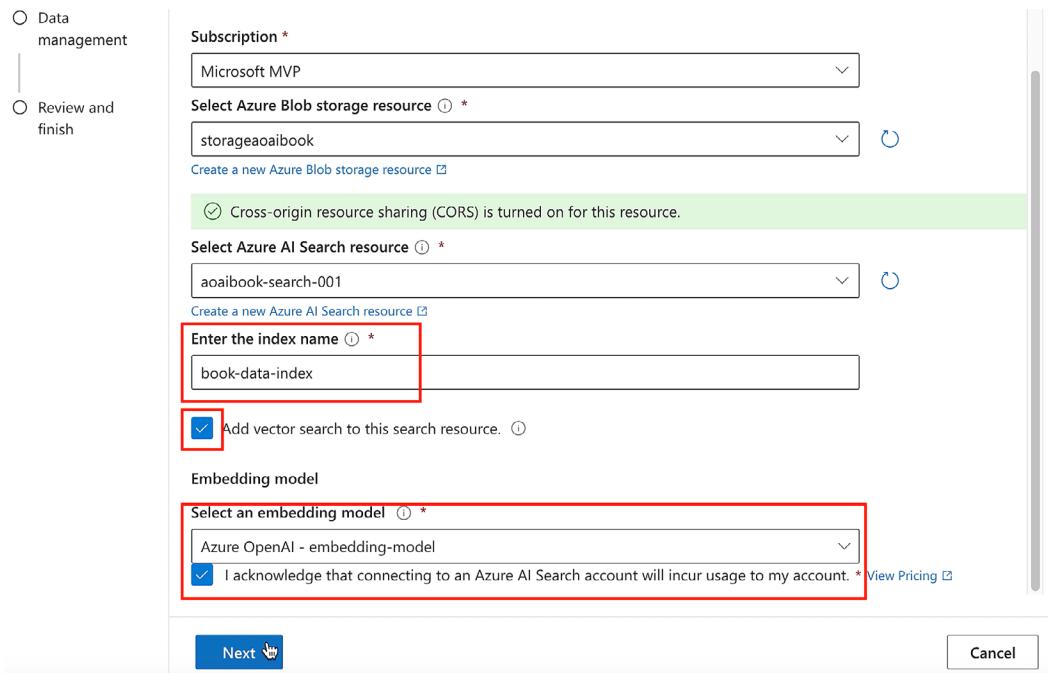


Figure 4.19: Selecting an embedding model

16. On the **Upload file** page, you can drag and drop your file or browse for a file to upload. For the purpose of this recipe, use a sample data file from Microsoft at [https://github.com/Azure-Samples/azure-search-openai-demo/blob/main/data/employee\\_handbook.pdf](https://github.com/Azure-Samples/azure-search-openai-demo/blob/main/data/employee_handbook.pdf). Click **Next** after the file is uploaded:

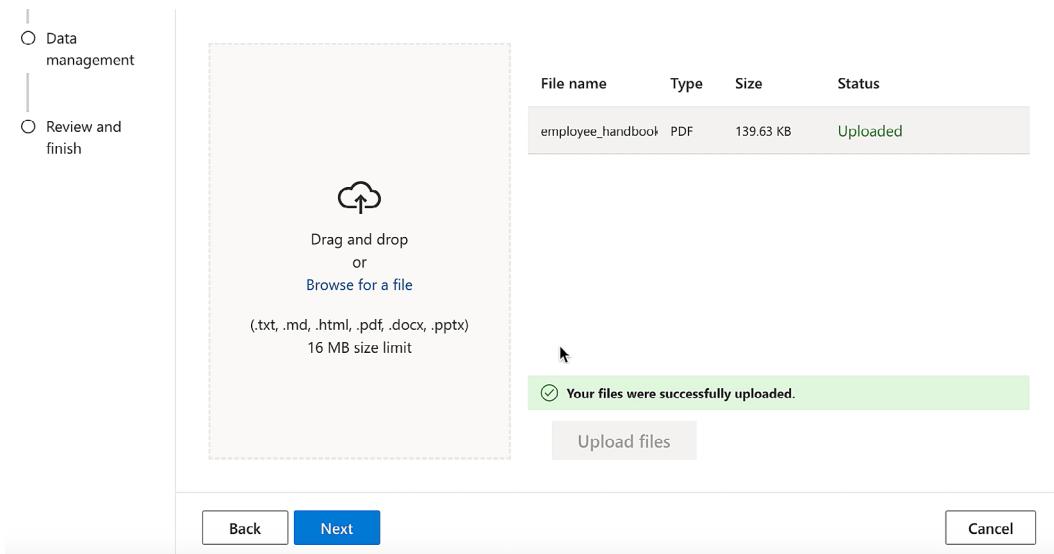


Figure 4.20: Uploading a file

17. In the **Data management** section, choose **Hybrid (vector + keyword)** for the **Search type** option. Check the **I acknowledge that adding vector embeddings will incur usage to my account** option. Then, click **Next**:

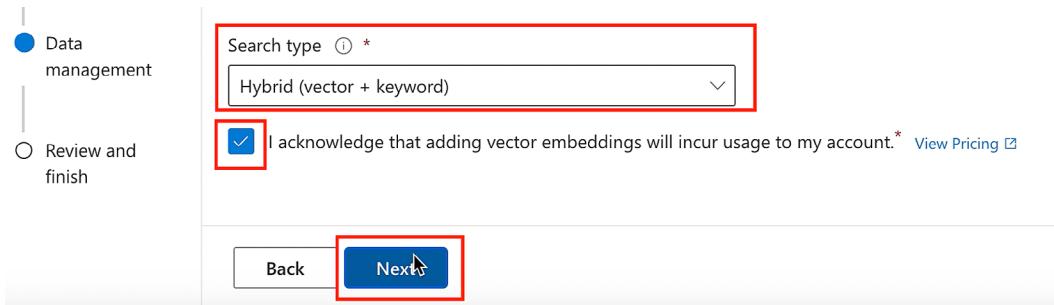


Figure 4.21: Data management section

18. Under **Review and finish**, you can review the details you entered. Click **Save and close** to submit:

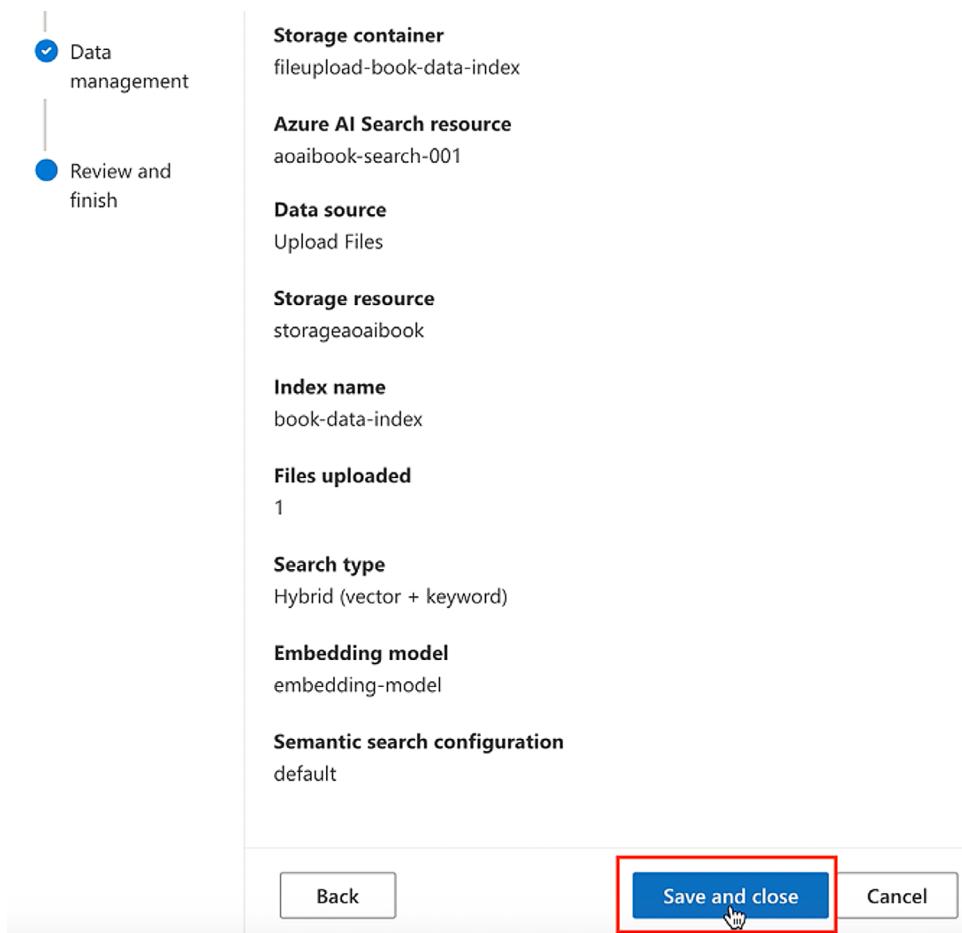


Figure 4.22: Review and finish

## How it works...

Creating vector embeddings with Azure OpenAI Studio involves the conversion of textual data into a numeric vector space that captures semantic similarities. Data sources can range from uploading files from your local machine (as shown in this recipe) to using Azure AI Search or Azure Blob Storage. The chosen data is then ingested into an Azure AI Search index. Azure AI Search also provides different search types: keyword search is traditional, relying on exact word matches; vector search leverages embeddings for semantic-based retrieval; and hybrid search combines both, offering a versatile search solution. Each search type has distinct retrieval capabilities.

## Chatting with your own data using Azure OpenAI Studio

In this recipe, you will learn how to interact with your own data using the Chat feature in Azure OpenAI Studio. You'll learn how to configure the AI model for data retrieval and search, select the appropriate deployment options, and ask questions relevant to your data.

By the end of this recipe, you'll know how to use Azure OpenAI Studio to generate responses based on your specific data and understand how to refer to the citations used to generate a specific response.

### Getting ready

Before you begin, ensure you have an Azure subscription and access to Azure OpenAI in your subscription.

### How to do it...

Follow these instructions to chat with your own data in Azure OpenAI Studio:

1. After uploading and embedding your data, select **Chat** from the left blade in Azure OpenAI Studio to open **Chat playground**:

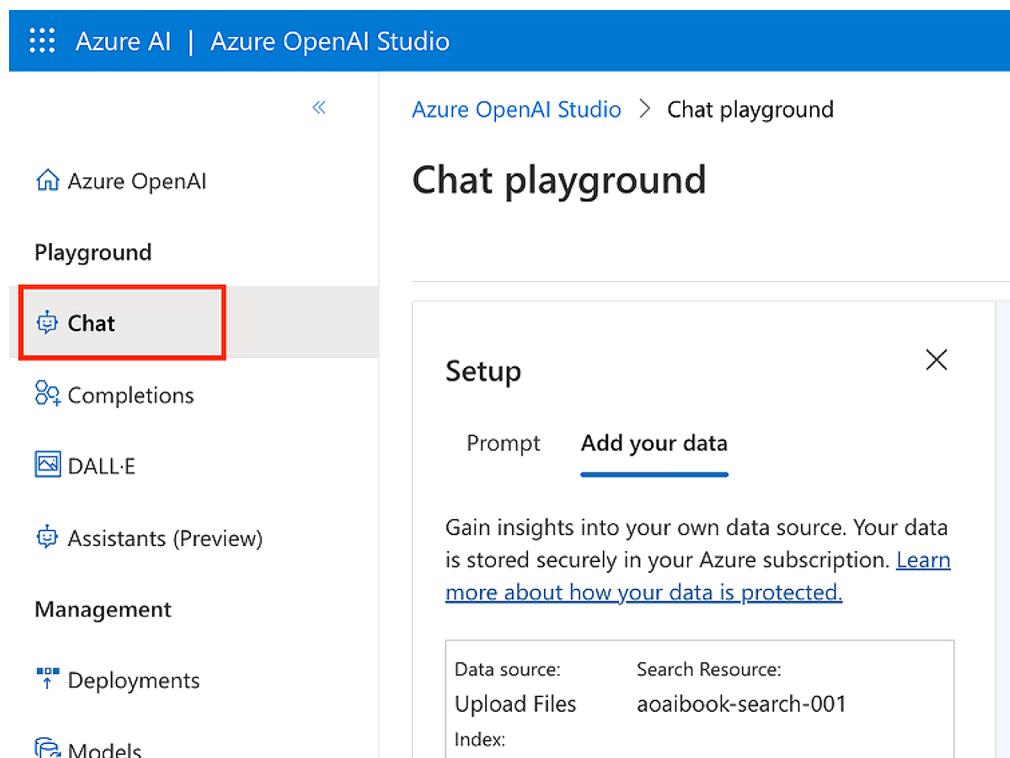


Figure 4.23: Chat playground in Azure OpenAI Studio

2. On the left side of the **Chat playground** page, you can find the **Setup** panel, where you can configure how the AI model retrieves and searches from the data you just uploaded. For example, checking **Limit response to your data content** ensures that the AI model's responses are drawn solely from your provided data. When this option is turned on, the AI model focuses on using only the information from your documents to answer queries. If it's turned off, the model can use its built-in knowledge to enhance answers:

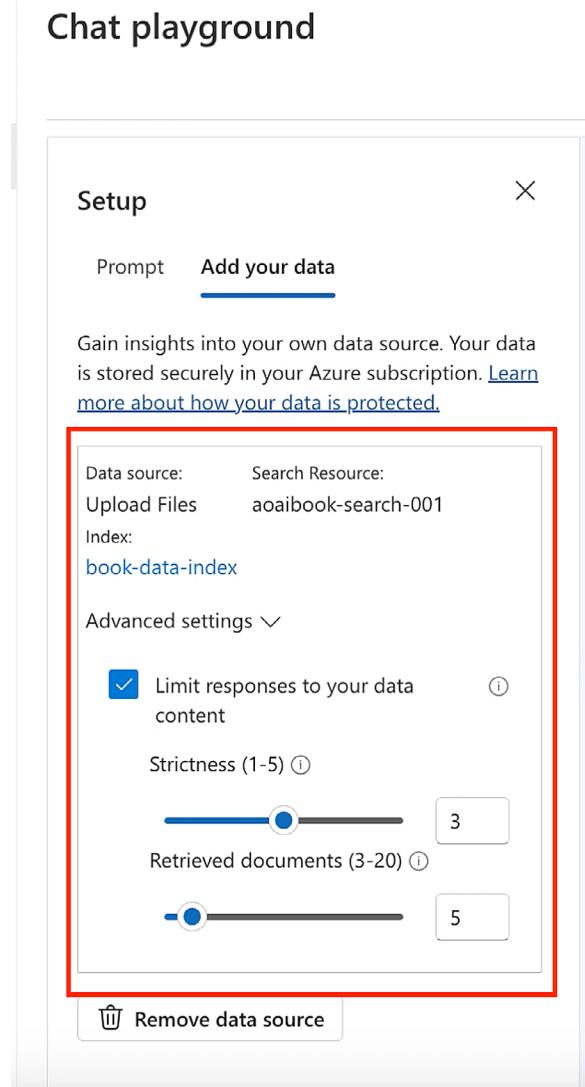


Figure 4.24: Limit responses to your data content

3. On the right side of the **Chat playground** page, you can select the **Deployment** option to decide which model will use the index's search results to generate responses. For this option, you can use the previously deployed GPT-4 model named `aoaibook`. You can also define the number of past messages to include as conversation history for future replies:

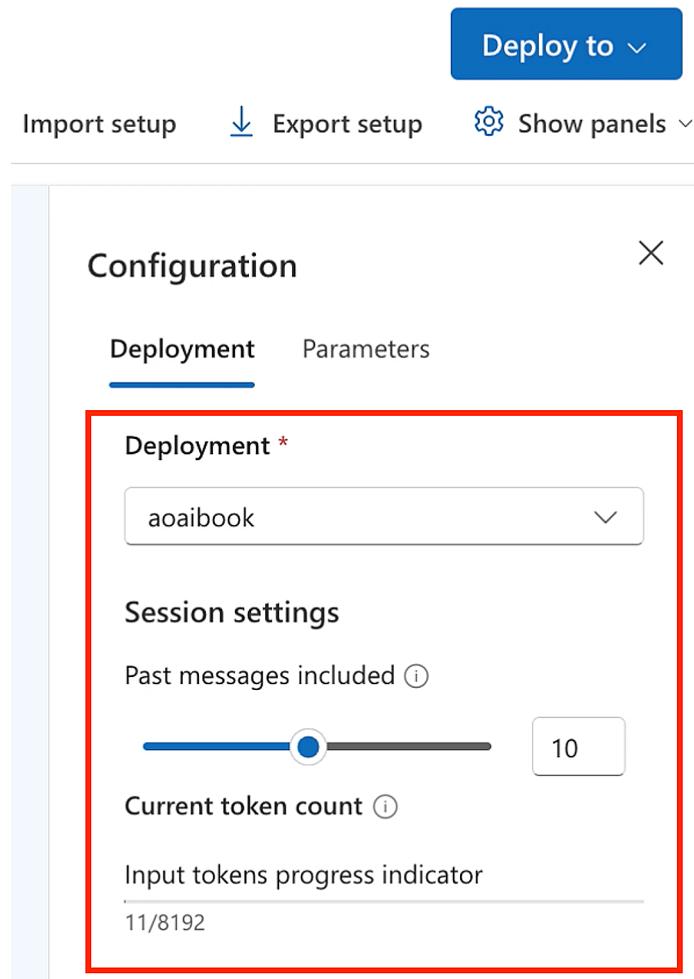


Figure 4.25: Configuration

4. Then, you can ask the language model questions related to your data. In this example, the sample data file from Microsoft has been uploaded, which is an employee handbook. So, you can ask questions such as what's our mission?. Then, click the arrow icon (send button):

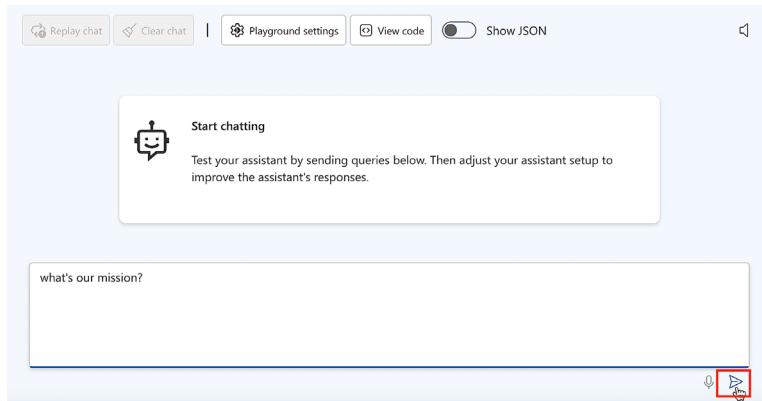


Figure 4.26: Start chatting page

5. You can see the response generated by the language model based on your data. Click on a reference in the **References** section to display the citations used to generate this specific response:

Figure 4.27: Citations

## How it works...

Azure OpenAI Studio is a powerful platform that enables users to interact with their data in a conversational manner. Using AI models, such as GPT-4, this platform allows users to ask questions and receive informative responses. This is made possible through the use of the Azure AI Search index, which employs the RAG pattern to sift through the data and deliver relevant answers. The process begins when a user poses a question; the model then retrieves pertinent information from the index and constructs a detailed response. For instance, if a user enquires about company values, the model will extract related information and provide a comprehensive answer about the organization's core principles.

## There's more...

If you're interested in deploying these resources more securely, you can refer to the following documentation. It provides comprehensive guidance on best security practices: <https://learn.microsoft.com/azure/ai-services/openai/how-to/use-your-data-securely>.

## Creating vector embeddings with the Azure OpenAI SDK

In this recipe, you will be guided through the process of creating vector embeddings with the Azure OpenAI SDK. This will involve navigating the Azure portal, setting up local environment variables, installing necessary Python packages, and utilizing the Azure OpenAI API to generate embeddings from your dataset.

By the end of this recipe, you will have gained practical skills in creating and managing vector embeddings, an essential tool in the field of data analysis and machine learning.

### Getting ready

To get started with this section, ensure you have an Azure subscription and access to Azure OpenAI in your subscription. You'll also need an Azure OpenAI resource with the `text-embedding-ada-002` model deployed, Python 3.8 or later, and Python libraries including `openai` and `pandas`. You'll be using Visual Studio Code with the Jupyter extension installed to run the code and visualize the results.

### How to do it...

Follow these instructions to create vector embeddings with the SDK:

1. Return to the Azure portal at <https://portal.azure.com> and log in to your account. Then, navigate to the Azure OpenAI resource you have created. Click the **Keys and Endpoint** tab on the left blade and copy the values for **KEY 1** and **Endpoint**:

The screenshot shows the 'Keys and Endpoint' section of the Azure AI services | Azure OpenAI | aoaibook-oai-001 page. The 'Keys and Endpoint' link in the left sidebar is highlighted with a red box and labeled '1'. In the main content area, the 'KEY 1' field and the 'Endpoint' field are both highlighted with red boxes and labeled '2' and '3' respectively. A tooltip above the 'KEY 1' field provides instructions: 'These keys are used to access your Azure AI services API. Do not share your keys. Store them securely—for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.' There are also 'Regenerate Key1' and 'Regenerate Key2' buttons.

Figure 4.28: Keys and Endpoint

2. Open a terminal on your computer. This recipe uses Terminal on macOS. Then, set the key and endpoint of the Azure OpenAI resource as local environment variables by using the `export` command:

```
export AZURE_OPENAI_API_KEY= value for KEY1  
export AZURE_OPENAI_ENDPOINT= value for Endpoint
```

3. Run the following command in the terminal to install the `openai` Python package:

```
pip3 install openai
```

4. You also need to install the `pandas` package, which is for data manipulation and analysis, by running the following command:

```
pip3 install pandas
```

5. Start your preferred IDE. This recipe uses Visual Studio Code with the Jupyter Notebook extension installed. You can type the `code` command in your terminal window to launch Visual Studio Code. Then, create a new Jupyter interactive window in Visual Studio Code.

6. You will import the required libraries into the notebook first:

```
import os
import pandas as pd
from openai import AzureOpenAI
```

7. Utilizing the pandas library, you can conveniently generate a new DataFrame to be used for creating embeddings:

```
# Define a dictionary containing sentences, their categories
data = {'Sentences': ['The cat sat on the mat',
                      'Dogs are the best friends of humans',
                      'I love to play football',
                      'AI is transforming the world',
                      'Python is a versatile programming
language'],
        'Category': ['Animal', 'Animal', 'Sport', 'Tech',
                     'Tech']
       }

# Convert the dictionary into DataFrame
df = pd.DataFrame(data)
df
```

8. Upon executing the preceding code, a table will be generated, as displayed in *Figure 4.29*:

	Sentences	Category
0	The cat sat on the mat	Animal
1	Dogs are the best friends of humans	Animal
2	I love to play football	Sport
3	AI is transforming the world	Tech
4	Python is a versatile programming language	Tech

Figure 4.29: The output

9. Then, you can use the Azure OpenAI API to generate embeddings for each sentence present in the **Sentences** column of our DataFrame. The resulting embeddings will then be stored in a new column called **Vector** of the DataFrame:

```
client = AzureOpenAI(  
    api_key = os.getenv("AZURE_OPENAI_API_KEY") ,  
    api_version = "2024-02-01" ,  
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")  
)  
  
def generate_embeddings(text, model="embedding-model"):  
    return client.embeddings.create(input = [text],  
model=model).data[0].embedding  
  
df['Vector'] = df["Sentences"].apply(lambda x : generate_  
embeddings (x, model = 'embedding-model'))  
  
df
```

10. After executing the code from the previous step, a table comprising vector data, also known as embeddings, will be produced. This table will display the embeddings that correspond to each sentence present in the **Sentences** column. Furthermore, these embeddings can be stored either locally or in an Azure database to facilitate vector search operations:

	Sentences	Category	Vector
0	The cat sat on the mat	Animal	[0.003707913914695382, 0.00343707506544888, -0...
1	Dogs are the best friends of humans	Animal	[0.010588900186121464, -0.004083847627043724, ...
2	I love to play football	Sport	[-0.014704311266541481, -0.003363986499607563,...
3	AI is transforming the world	Tech	[-0.008747941814363003, -0.01931431144475937, ...
4	Python is a versatile programming language	Tech	[-0.00023285826318897307, -0.00337960803881287...

Figure 4.30: Vector data

## How it works...

Creating vector embeddings with the SDK involves using embeddings. An embedding is a dense information format that represents the semantic meaning of a text segment, easily utilized by AI models. These embeddings, vectors of floating-point numbers, help determine the semantic similarity between two inputs. Azure OpenAI embedding models perform various tasks, such as capturing semantic similarities between texts and embedding code snippets. They rely on cosine similarity to compute the similarity between documents and a query—an advantageous approach over simply counting common words.

## Chatting with your own data using the Azure OpenAI SDK

In this recipe, you will engage with Azure OpenAI APIs to generate responses based on data stored in an Azure AI Search resource you've previously set up. You'll explore the process of retrieving data from Azure AI Search, initializing an Azure OpenAI client, and generating chat completions based on user messages and additional data sourced from Azure AI Search.

By the end of this recipe, you will have learned how to use the Azure OpenAI SDK to facilitate interactive conversations through the use of your own data, thereby enhancing your ability to develop and manage conversational AI applications.

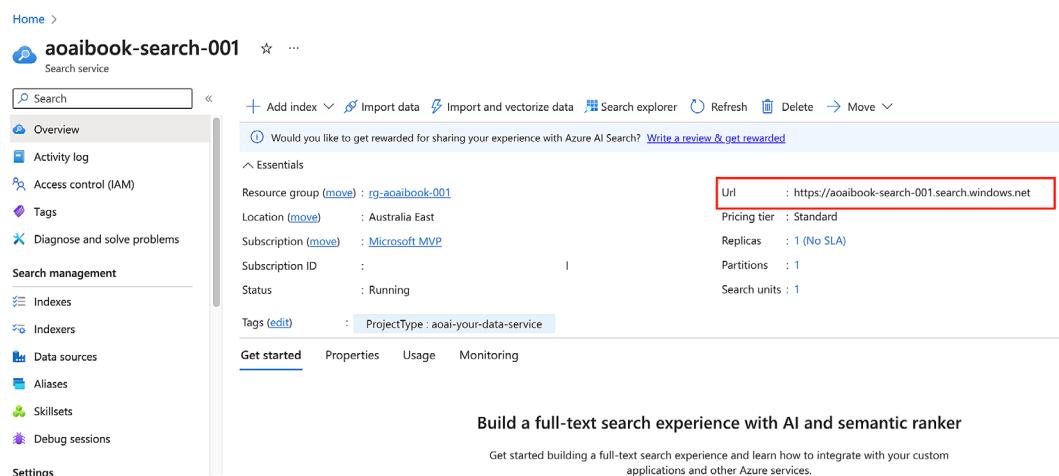
### Getting ready

Before you begin, ensure you have an Azure subscription and access to Azure OpenAI in your subscription. In addition to the key and endpoint of the Azure OpenAI resource, you also need the index name, key, and endpoint of the Azure AI Search resource to retrieve data.

### How to do it...

You will learn how to interact with Azure OpenAI APIs to generate responses according to the data stored in the Azure AI Search resource you previously set up:

1. Open the Azure portal at <https://portal.azure.com> and log in to your account. Then, navigate to the **Azure AI Search** resource you have created and copy the **Url** value on the **Overview** page:



The screenshot shows the Azure portal interface for the 'aoaibook-search-001' search service. The URL 'https://aoaibook-search-001.search.windows.net' is highlighted with a red box. Other visible details include the Resource group (rg-aoaibook-001), Location (Australia East), Subscription (Microsoft MVP), Status (Running), and Tags (ProjectType:aoai-your-data-service).

Figure 4.31: Url Azure AI Search

2. Under the **Settings** tab, click on **Keys** and copy the **Primary admin key** value:

Home > aoaibook-search-001

The screenshot shows the 'aoaibook-search-001 | Keys' page. On the left sidebar, under the 'Settings' tab, the 'Keys' option is highlighted with a red box and the number '1'. In the main content area, the 'API Access control' section is selected, showing three radio button options: 'API keys' (selected), 'Role-based access control', and 'Both'. Below this is the 'Manage admin keys' section, which is also highlighted with a red box and the number '2'. It contains two fields: 'Primary admin key' and 'Secondary admin key', each with a 'Copy to clipboard' button and a 'Regenerate' button. The 'Primary admin key' field is currently selected.

Figure 4.32: Key in Azure AI Search

3. Under the **Search management** tab, click on **Indexes**. You can find the index you've created. Copy its name:

Home > aoaibook-search-001

The screenshot shows the 'aoaibook-search-001 | Indexes' page. On the left sidebar, under the 'Search management' tab, the 'Indexes' option is highlighted with a red box and the number '1'. The main content area displays a table of indexes. One row is highlighted with a red box and the number '2', corresponding to the 'book-data-index' entry. The table has columns for 'Name', 'Document count', and 'Vector index size'. The 'book-data-index' row shows a document count of 4 and a vector index size of 24.54 KB.

Name	Document count	Vector index size
book-data-index	4	24.54 KB

Figure 4.33: Index name in Azure AI Search

4. Open a terminal on your computer. This recipe uses Terminal on macOS. Set the key, endpoint, and index name of the Azure AI Search resource as local environment variables:

```
export AZURE_AI_SEARCH_ENDPOINT= value for Endpoint
export AZURE_AI_SEARCH_API_KEY= value for AI Search Key
export AZURE_AI_SEARCH_INDEX= value for the index name
```

5. Create a new Jupyter interactive window in Visual Studio Code. Then, you will import the required libraries and initialize an Azure OpenAI client with the specified base URL, API key, and API version in the notebook. It's crucial to note that the deployment variable, which you will use to initialize the client, should be set to the name of your model deployment. This is the same name you assigned to your model deployment in *Step 8* of the *Creating vector embeddings with Azure OpenAI Studio* recipe:

```
import os
import openai

endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
api_key = os.getenv("AZURE_OPENAI_API_KEY")
deployment = "aoaibook";

client = openai.AzureOpenAI(
    base_url=f"{endpoint}/openai/deployments/{deployment}/extensions",
    api_key=api_key,
    api_version="2023-08-01-preview",
)
```

6. Then, you can use the Azure OpenAI client to generate a chat completion based on a user message and additional data sourced from Azure AI Search. For instance, you can ask questions such as `What are our values?` that are included in the sample employee handbook and retrieve environment variables for the Azure AI Search resource. After generating the chat completion, you can print the JSON representation of the completion:

```
completion = client.chat.completions.create(
    model=deployment,
    messages=[
        {
            "role": "user",
            "content": "What are our values?",
        },
    ],
    extra_body={
        "dataSources": [
            {
                "type": "AzureCognitiveSearch",
                "parameters": {
                    "endpoint": os.getenv("AZURE_AI_SEARCH_ENDPOINT"),

```

```
        "key": os.getenv("AZURE_AI_SEARCH_API_KEY"),
        "indexName": os.getenv("AZURE_AI_SEARCH_
INDEX")
    }
}
]
}
)
print(completion.model.dump_json(indent=2))
```

- Upon execution, the response you receive will be in the form of a JSON string generated by the OpenAI model. This string represents a simulated conversation wherein a user is inquiring about the values of a company. In response, the assistant provides a detailed list of the company's core values. Additionally, the JSON string includes metadata about the conversation, such as the citations:

Figure 4.34: Response

## How it works...

Creating vector embeddings with the SDK involves utilizing the Azure OpenAI API to generate a completion for a chat conversation. The Python code initializes an Azure OpenAI client using the endpoint, API key, and deployment name. Then, it generates a completion for a chat conversation, which comprises a single user message. Additionally, the `extra_body` parameter specifies extra data sources, such as an Azure AI Search data source, to be used when generating the completion. The code concludes by printing the JSON representation of the completion for readability.

# 5

## Fine-Tuning LLMs for Your Use Case

In this chapter, you will learn how to tailor **large language models (LLMs)** for your specific use case. This process allows you to adapt pre-existing models to better suit your needs, thereby enhancing the performance of AI applications and services. By providing practical explanations of data preparation, model fine-tuning, and testing, this chapter aims to equip you with the knowledge and tools to create an efficient, tailored model using **Azure OpenAI Studio** and the **Azure SDK**.

Throughout this chapter, you will engage in identifying suitable data for your model. You will also learn to use Azure OpenAI Studio and the SDK for fine-tuning and testing your model. This knowledge will empower you to create more efficient and reliable models, tailored to your specific requirements, leading to more reliable and consistent results in your AI projects.

This chapter will cover the following recipes:

- Preparing your dataset
- Fine-tuning your model with Azure OpenAI Studio
- Fine-tuning your model with the Azure SDK
- Testing your fine-tuned model with Azure OpenAI Studio
- Testing your fine-tuned model with the Azure SDK

## Technical requirements

To successfully follow along with the recipes in this chapter, ensure that you meet the following technical requirements:

- An active Azure account, either free or paid.
- Access to Azure OpenAI Service within your Azure subscription, ensuring you are assigned at least the **Cognitive Services Contributor** role for the Azure OpenAI resource and the **Cognitive Services OpenAI Contributor** role for fine-tuning access. If necessary, you can apply for access to Azure OpenAI Service at <https://aka.ms/oai/access>.
- You can find the sample datasets for this chapter at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter5>.
- Python 3.8 or a later version installed on your system.
- The necessary Python libraries installed, such as `openai`.
- Visual Studio Code with the Jupyter extension installed for running the code and visualizing the results.
- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter5>. The Chapter 5 folder contains all the notebooks for this chapter.

## Preparing your dataset

In this recipe, you will focus on fine-tuning the GPT-3.5 Turbo model for your specific use case, particularly for preparing your dataset. You will identify the right kind of data for your model, format your data into a suitable structure for training, and understand the need for splitting your data into training and validation sets. In this case, you will use JSON-formatted messages from a customer support scenario. You will also learn how to ensure your dataset is of high quality and compatible with your model.

By the end of this recipe, you will be proficient in preparing a dataset for fine-tuning an Azure OpenAI LLM, enabling you to customize your own LLM more effectively. This understanding will empower you to create a more efficient and reliable model, tailored to your specific requirements.

## Getting ready

For this recipe, you must have an Azure account, either free or paid. You can find the sample datasets for this section at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter5>.

## How to do it...

Follow these steps to create a dataset for fine-tuning an Azure OpenAI LLM:

1. Identify the data from which you want the model to learn. This could be a set of documents, conversations, or any text relevant to the task that you want the model to perform.
  2. Next, format your data to be suitable for training. The data should be in **JSON Lines (JSONL)** format. Each line of the JSONL file represents one sample from which the model should learn. Here's an example of how to structure your data:

Figure 5.1: Training set

3. Then, split your data into at least two sets: a **training set** and a **validation set**. The model learns from the training set, and the validation set is used to evaluate the model's performance;

```
(-) training_set.jsonl  (-) validation_set.jsonl x
(-) validation_set.jsonl
1  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "1"}, 2  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "2"}, 3  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "3"}, 4  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "4"}, 5  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "5"}, 6  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "6"}, 7  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "7"}, 8  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "8"}, 9  {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "9"}, 10 {"messages": [{"role": "system", "content": "You are a customer support assistant for a software company."}, {"role": "user", "content": "What is the capital of France?"}, {"role": "assistant", "content": "The capital of France is Paris."}], "id": "10"}, 11
```

Figure 5.2: Training set and validation set

4. The datasets need to be **UTF-8** encoded and should contain a **byte-order mark (BOM)**. Furthermore, the file's size should not exceed 100 MB.
  5. Ensure that your dataset is of high quality to get the best results.

Once your dataset is ready, you can proceed to fine-tune your model using Azure OpenAI Studio or the Azure OpenAI SDK. Remember, the more training samples you provide, the better your model is likely to perform. However, ensure that your data is of high quality to get the best results.

## How it works...

**Fine-tuning** is a process of customizing an LLM. It's one way to add your own data to the model, changing its weights and teaching it new skills or information. It's different from techniques such as **prompt engineering** and **retrieval-based techniques** that only modify the prompt. Fine-tuning retrains the model itself.

Understanding where prompt engineering falls short can provide guidance on fine-tuning. The base model may fail on edge cases or exceptions, or it may not consistently provide output in the right format. Identifying these shortcomings will help you collect the necessary data for fine-tuning and evaluating your fine-tuned model.

## There's more...

While fine-tuning can be a powerful tool to customize LLMs, remember that the output's quality is heavily dependent on the input data's quality. Bad data can result in a poorly performing model. Therefore, preparing a good dataset is a critical part of the fine-tuning process. In terms of data requirements for fine-tuning, it depends on the use case. However, a minimum recommendation is at least 50 high-quality lines, though some use cases may require thousands of data points or lines.

## Fine-tuning your model with Azure OpenAI Studio

This recipe will walk you through the process of fine-tuning your model using Azure OpenAI Studio.

By the end of this recipe, you will have a hands-on understanding of how to fine-tune your model using Azure OpenAI Studio. You will cover topics such as deploying models, adding data, and configuring the search type for effective data management.

### Note

As of the time of writing, only specific regions support the fine-tune function, such as **northcentralus** and **swedencentral**, so the Azure OpenAI resource should be deployed in these regions. For details, refer to <https://learn.microsoft.com/azure/ai-services/openai/concepts/models#standard-deployment-model-quota>.

## Getting ready

You need a deployed Azure OpenAI resource. If you haven't deployed one yet, please refer to the *Getting started with Azure OpenAI Service* recipe in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*.

## How to do it...

To fine-tune your model with Azure OpenAI Studio, follow these steps:

1. Go to the Azure OpenAI Studio website at <https://oai.azure.com/>. Log in using the same credentials that you used to create the Azure OpenAI resource.
2. Once logged in, choose the Azure OpenAI resource you deployed:

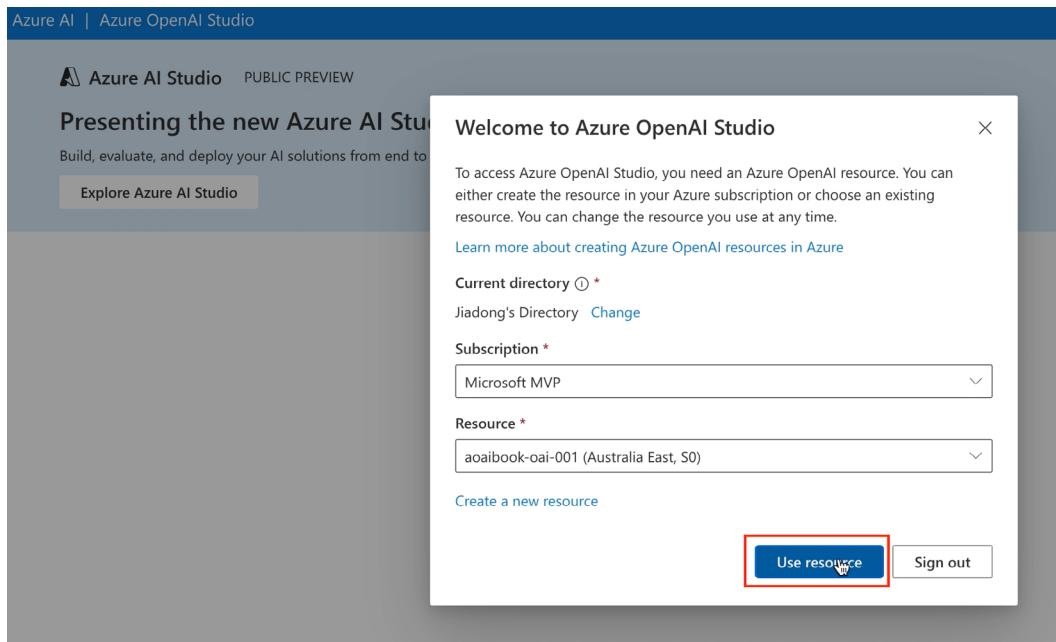


Figure 5.3: Log in to Azure OpenAI Studio

3. In the **Management** section on the left, choose **Models**. Then, click on **Create a custom model** to open a new custom model wizard page:

The screenshot shows the Azure OpenAI Studio interface. On the left, there's a sidebar with various options: Azure OpenAI, Playground, Chat, Completions, DALL-E, Assistants (Preview), Management, Deployments, and Models. The 'Models' option is highlighted with a red box and the number '1'. The main content area is titled 'Models' and contains a table of 'Base models'. At the top of the table, there's a button labeled 'Create a custom model' which is also highlighted with a red box and the number '2'. The table has columns for Model name, Model version, Created at, and Status. All listed models have a status of 'Succeeded'.

Model name	Model version	Created at	Status
dall-e-3	3.0	8/11/2023 10:00 AM	Succeeded
gpt-35-turbo	0125	2/15/2024 11:00 AM	Succeeded
gpt-35-turbo	1106	11/15/2023 11:00 AM	Succeeded
gpt-35-turbo	0613	6/19/2023 10:00 AM	Succeeded
gpt-35-turbo-16k	0613	6/19/2023 10:00 AM	Succeeded
gpt-4	vision-preview	11/16/2023 11:00 AM	Succeeded
gpt-4	0613	6/19/2023 10:00 AM	Succeeded

Figure 5.4: Create a custom model

4. In the **Base model** section, choose a **Base model type** from the dropdown list:

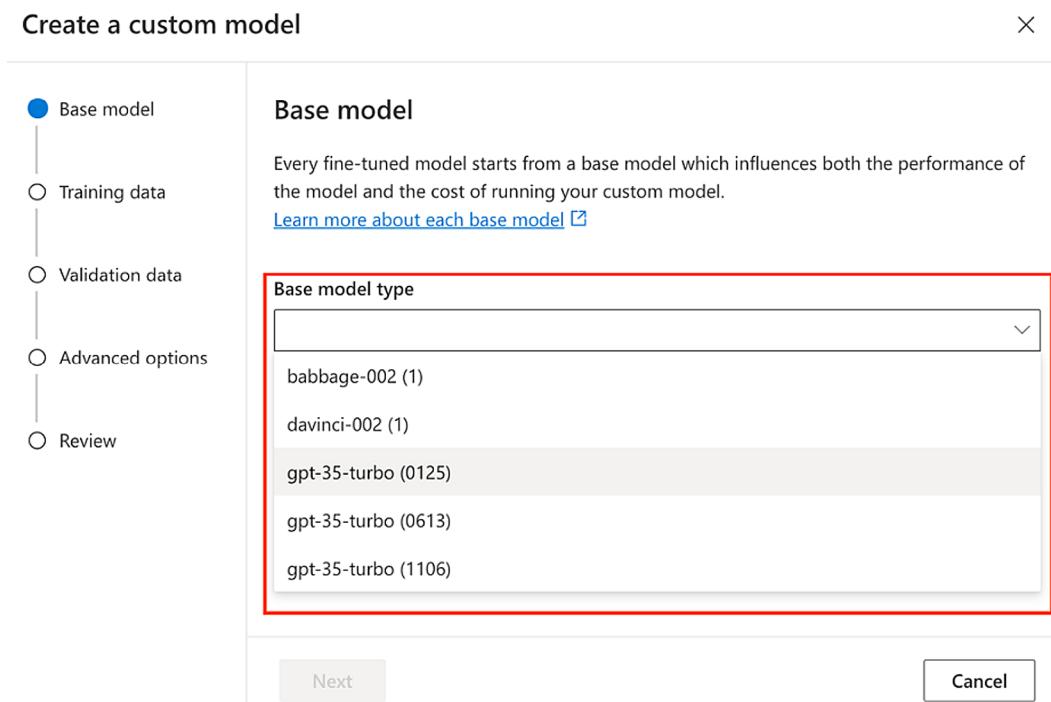


Figure 5.5: Choosing a base model

5. In this case, choose the new **gpt-35-turbo (0125)** model as the base model. Optionally, you can also add a model suffix for easier identification. Then, click **Next**:

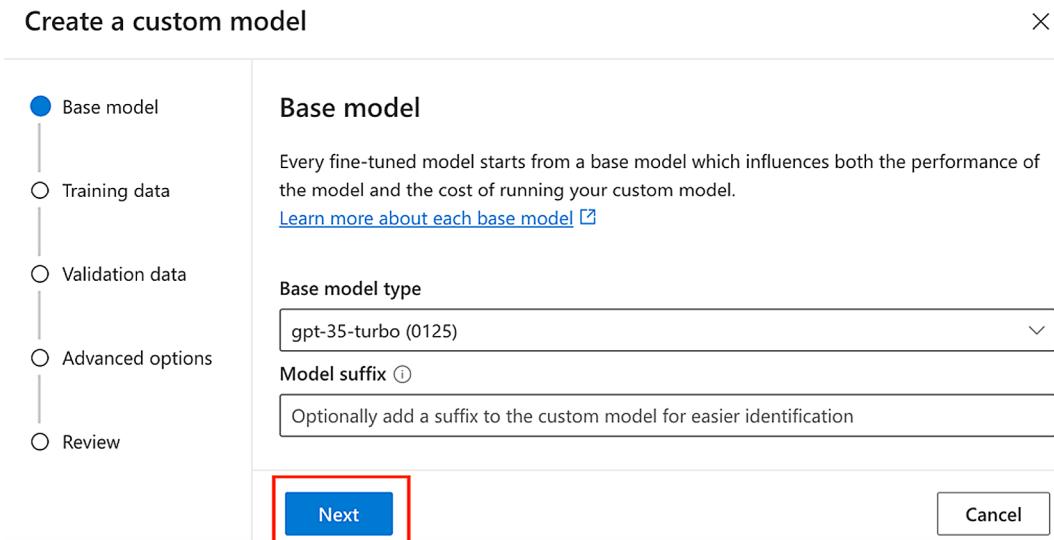


Figure 5.6: gpt-35-turbo (0125) as the base model

6. In the **Training data** section, select the **Upload your local files** tab. Proceed by dragging and dropping the `training_set.jsonl` file for upload into Azure OpenAI Studio:

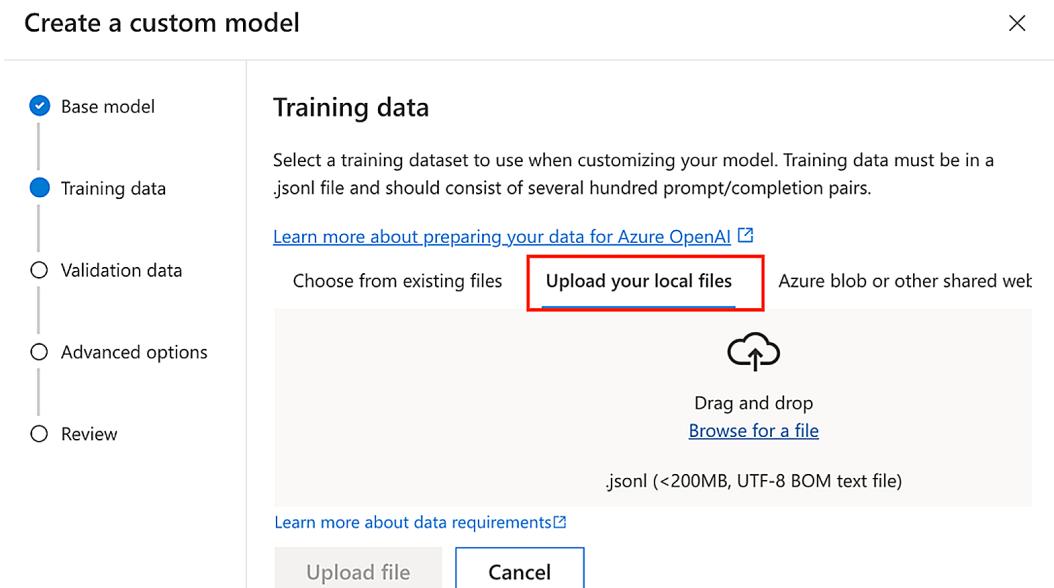


Figure 5.7: Upload your local files

7. After the training dataset file is uploaded, you can see it in the **Choose from existing files** tab. Select this file and click **Next**:

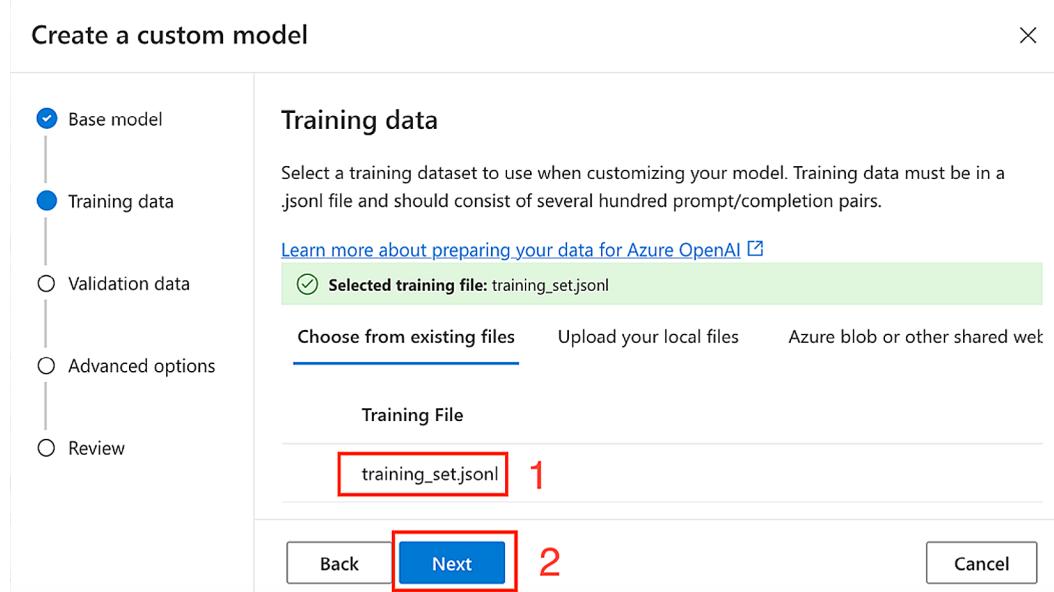


Figure 5.8: Select the training set

8. Repeat *Steps 6 and 7* in the **Validation data** section to upload the validation dataset and select it as the validation file on the **Choose from existing files** tab:

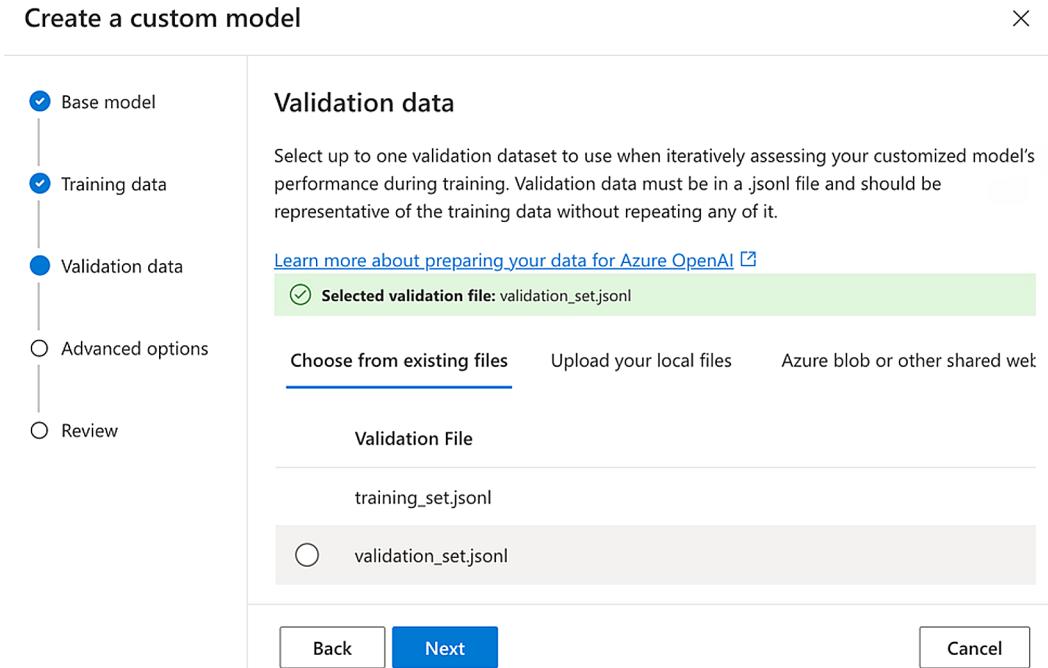


Figure 5.9: Choose the validation data

9. In the **Advanced options** section, choose **Default** for standard fine-tuning settings, or **Custom** to manually adjust the hyperparameters:

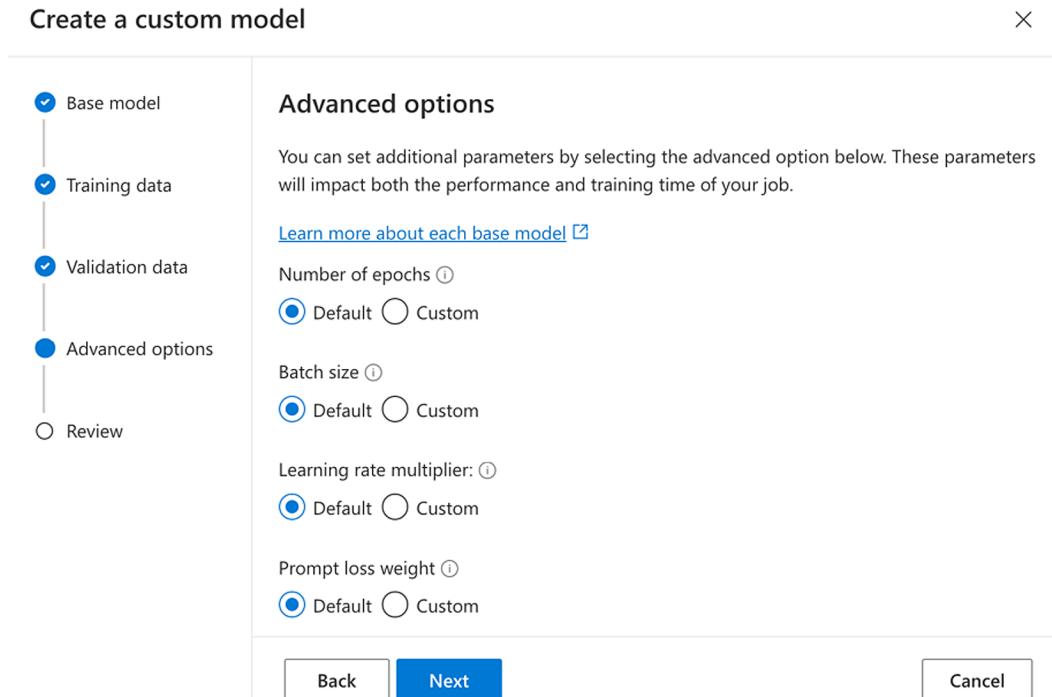


Figure 5.10: Advanced options

10. In the **Review** section, you can review your selections. If you're ready, click on **Start Training job**:

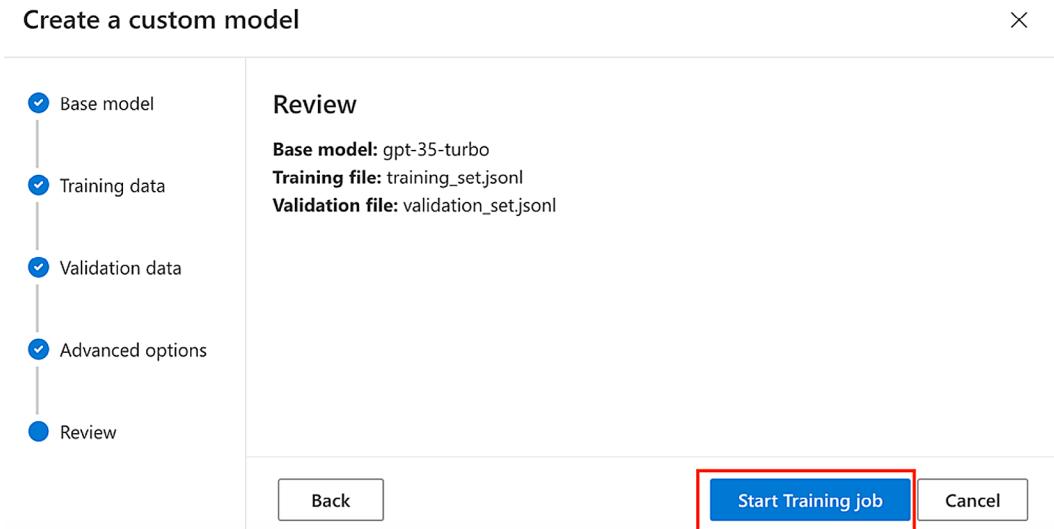


Figure 5.11: Review and Start Training job

11. Return to the **Models** page. Click on the **Custom models** tab. You can see the status of your model, as shown in *Figure 5.12*. Please note that training may take anywhere from a few minutes to a few hours:

Azure OpenAI Studio > Models

## Models

Azure OpenAI is powered by models with different capabilities and price points. Deploy one of the provided base models to ease and data for better performance and more accurate results.

[Learn more about the different types of base models](#)

The screenshot shows the 'Custom models' tab selected in the Azure OpenAI Studio 'Models' page. The table lists a single model entry:

Model name	M... ▾	C.. ↓ ▾	Base model	Status
<a href="#">ftjob-e26b1898a9d44aae9ae38e00780feec9</a>	1	3/31/2024	gpt-35-turbo-012	Running

At the top of the table, there are buttons for 'Deploy', 'Create a custom model', 'Delete', 'Column options', and 'Refresh'. The 'Custom models' tab is highlighted with a red box.

Figure 5.12: Custom models status

After the training process is completed, you can deploy your fine-tuned model.

12. In the **Management** section, choose **Deployments**. Click on **Create new deployment** to open the **Deploy model** pane:

Azure AI | Azure OpenAI Studio

Azure OpenAI Studio > Deployments

## Deployments

Deployments provide endpoints to the Azure OpenAI base models, or your fine-tuned models, configure handling, and deployment size. From this page, you can view your deployments, edit them, and create ne

+ Create new deployment | ↴

Edit deployment Delete deployment Column option

2

1

No deployment

No deployments found. Please create

Figure 5.13: Deployment

13. On the **Deploy model** pane, you can select the fine-tuned model from the dropdown list, enter your desired **Deployment name**, and then click **Create** to start the deployment:

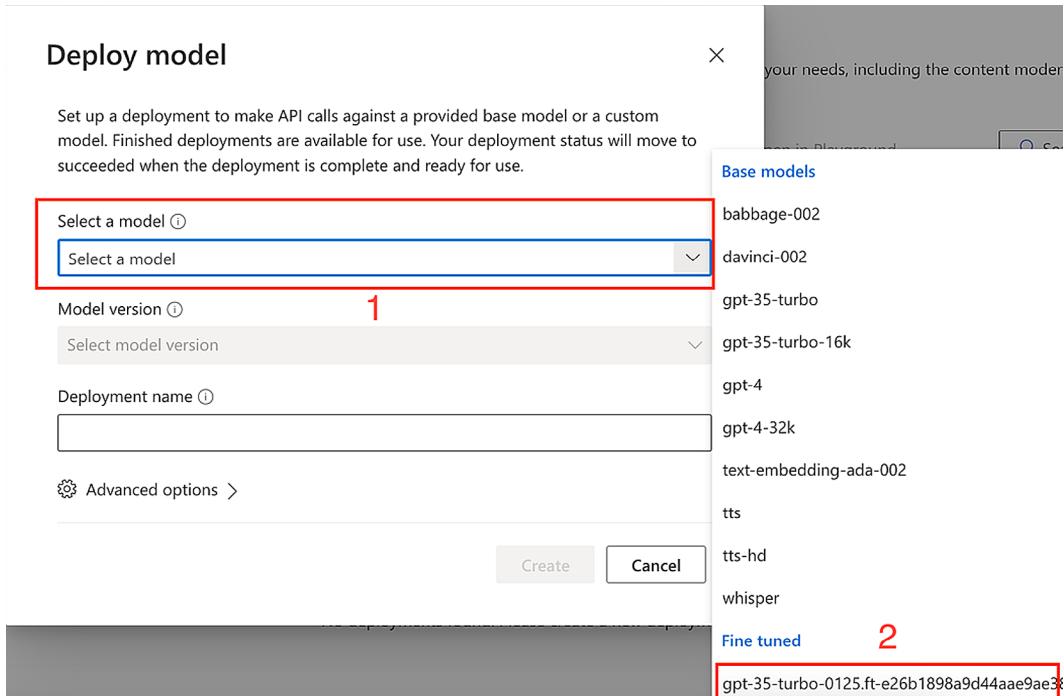


Figure 5.14: Deploy model

14. Wait for the deployment to complete. At this point, you have created a fine-tuned model and deployed it, which is now ready for use:

## Status: Training succeeded

Training file: training\_set.jsonl  
Base model: gpt-35-turbo-0125  
Total training time: 1 hour, 37 seconds  
Training hours billed: 0.250

[Download training file](#)

Figure 5.15: Training succeeded

## How it works...

Fine-tuning your model with Azure OpenAI Studio is a simple process that allows you to customize and optimize your models. The platform provides a **Create a custom model** wizard that shows the hyperparameters for training your fine-tuned model.

The available hyperparameters are **batch size**, **learning rate multiplier**, and the **number of epochs** for model training. Large batch sizes and learning rates are generally better for large datasets, while a small learning rate may help prevent overfitting.

## There's more...

Additionally, if you would prefer to import your dataset directly from Azure Blob store instead of uploading a local file, you can do so by following the streamlined steps at the following link: <https://learn.microsoft.com/azure/ai-services/openai/how-to/fine-tuning?tabs=turbo%2Cpython-new&pivots=programming-language-studio#import-training-data-from-azure-blob-store>.

## Fine-tuning your model with the Azure SDK

In this recipe, you will fine-tune your model using the Azure OpenAI Python SDK. This recipe will guide you through the process of preparing your training and validation data, selecting a base model, and uploading your training data. You'll then learn how to create a new custom model by training it using the uploaded data.

By the end of this recipe, you will know how to fine-tune your model using the Azure OpenAI SDK.

### Getting ready

To get started with this recipe, ensure you have an Azure subscription and access to Azure OpenAI in your subscription. You'll also need an Azure OpenAI resource deployed, Python 3.8 or later, and the `openai` Python library.

#### Note

This recipe uses Visual Studio Code with the Jupyter extension installed to run the code and visualize the results.

## How to do it...

Follow these instructions to fine-tune your model with the SDK:

1. Go to the Azure portal at <https://portal.azure.com> and log in to your account.
2. Navigate to your Azure OpenAI resource, click the **Keys and Endpoint** tab on the left blade, and copy the values for **KEY 1** and **Endpoint**:

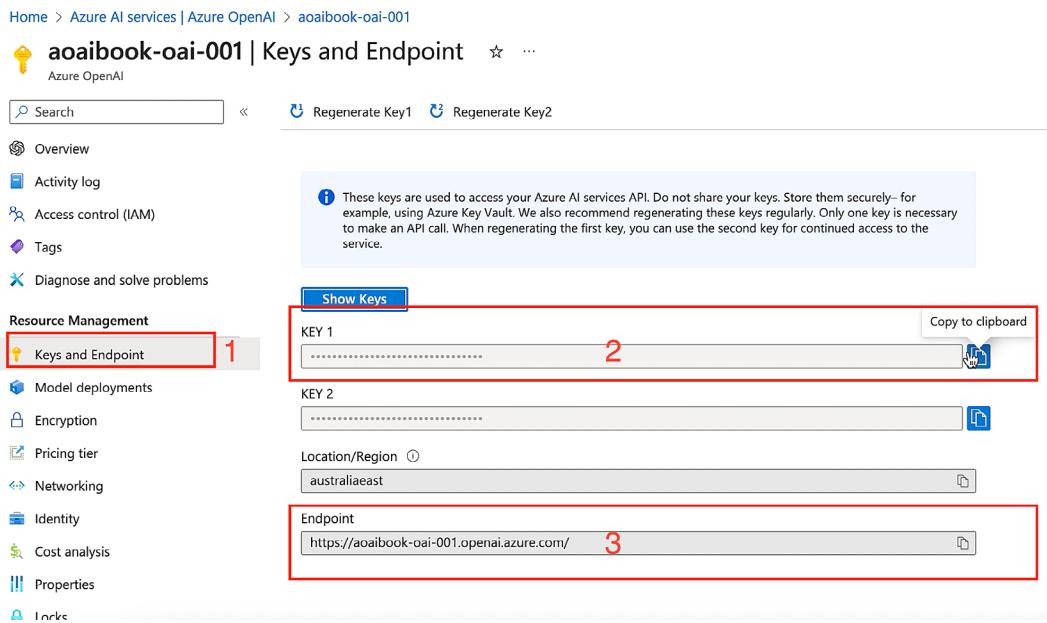


Figure 5.16: Keys and Endpoint tab

3. Open a terminal on your computer. This recipe uses Terminal on macOS. Then, set the key and endpoint of the Azure OpenAI resource as local environment variables:

```
echo export AZURE_OPENAI_API_KEY= value of KEY1 >> ~/.bash_profile
echo export AZURE_OPENAI_ENDPOINT= value of Endpoint >> ~/.bash_profile
source ~/.bash_profile
```

### Note

If you are using the Windows operating system, check out this link to learn how to set these environment variables: <https://learn.microsoft.com/windows-server/administration/windows-commands/setx>.

4. Run the following command in the terminal to install the `openai` Python package:

```
pip install openai
```

5. Create a new Jupyter interactive window in Visual Studio Code and put the two dataset files, `training_set.jsonl` and `validation_set.jsonl`, which you used before, in the same directory.
6. Then, you must import the required libraries into the notebook:

```
import os
from openai import AzureOpenAI
```

7. Next, create an instance of the `AzureOpenAI` client with specific parameters. The client, assigned to the `client` variable, interacts with Azure OpenAI Service. The `azure_endpoint` and `api_key` parameters are obtained securely from the `AZURE_OPENAI_ENDPOINT` and `AZURE_OPENAI_API_KEY` environment variables respectively, using the `os.getenv` function. The `api_version` parameter is set to `2024-02-01`, specifying the API version to be used. This or a later version is required for accessing fine-tuning for certain models:

```
client = AzureOpenAI(
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-02-01"
)
```

8. You can upload training and validation datasets to Azure OpenAI for model fine-tuning using the `client.files.create` method. This method opens the respective file in binary mode and uploads it for fine-tuning:

```
training_file_name = 'training_set.jsonl'
validation_file_name = 'validation_set.jsonl'

training_response = client.files.create(
    file=open(training_file_name, "rb"), purpose="fine-tune"
)
training_file_id = training_response.id

validation_response = client.files.create(
    file=open(validation_file_name, "rb"), purpose="fine-tune"
)
validation_file_id = validation_response.id

print("Training file ID:", training_file_id)
print("Validation file ID:", validation_file_id)
```

9. After executing the code from the preceding step, the response object's `id` attribute retrieves the ID of the uploaded file:

```
Training file ID: file-573cef81f9544d7a96d42fae1e6687fe
Validation file ID: file-068fbeca2fba45ec9248dc89b715b714
```

Figure 5.17: File ID

10. Then, you can fine-tune a model in Azure OpenAI using the `client.fine_tuning.jobs.create` method. It employs previously uploaded training and validation files, which are identified through their respective IDs. The base model, named `gpt-35-turbo-0125`, is fine-tuned with this process:

```
response = client.fine_tuning.jobs.create(
    training_file=training_file_id,
    validation_file=validation_file_id,
    model="gpt-35-turbo-0125",
)

job_id = response.id

print("Job ID:", response.id)
print("Status:", response.status)
```

11. Note that training may take anywhere from a few minutes to a few hours. If you return to the **Models** page of Azure OpenAI Studio and click on the **Custom models** tab, you can see the status of the fine-tuning process.

## How it works...

Fine-tuning your model with the SDK works by preparing and uploading your training data, which can either be existing or new data. After uploading your training and validation files, you proceed to the fine-tuning process, which involves creating a new fine-tuning job with the Python SDK. For more customized control, the hyperparameters can be adjusted. The hyperparameters that can be manipulated are **batch size**, **learning rate multiplier**, and the **number of epochs** for training the model.

## Testing your fine-tuned model with Azure OpenAI Studio

In this recipe, you will learn how to test your model using Azure OpenAI Studio. You will navigate through the website, utilize the Chat playground to configure a system message, and select the model for response generation.

By the end of this recipe, you will know how to effectively fine-tune your model and validate it using example queries. Understanding how to interact conversationally with your data using AI models and how the system message plays a crucial role in the context of the AI's responses will help you achieve more reliable and consistent results in your AI projects.

### Getting ready

To get started with this section, ensure you have an Azure subscription and access to Azure OpenAI Service in your subscription. You'll also need an Azure OpenAI resource with a fine-tuned model deployed.

### How to do it...

Follow these instructions to test your fine-tuned model with Azure OpenAI Studio:

1. Go to the Azure OpenAI Studio website at <https://oai.azure.com/>.
2. Click on the **Chat** tab in the left blade in Azure OpenAI Studio to open the **Chat playground** page:

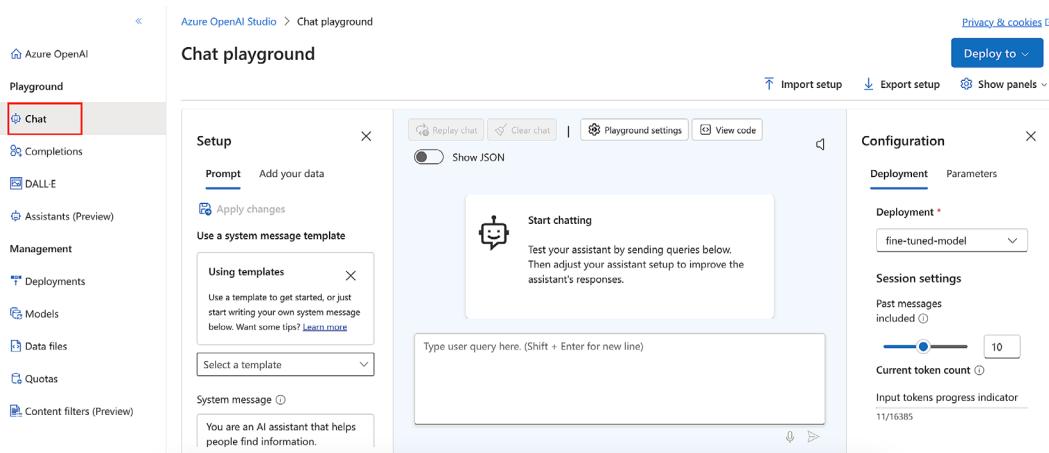


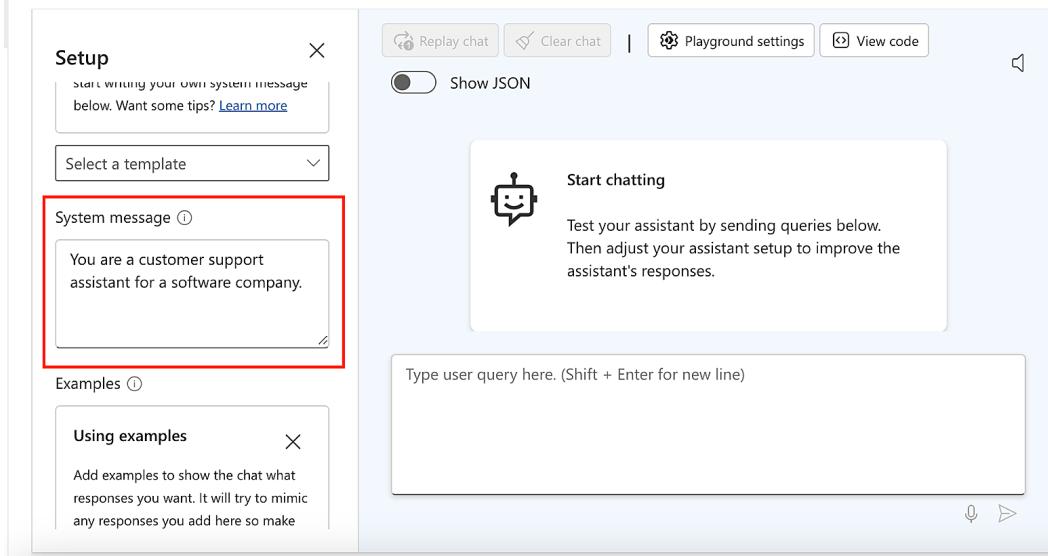
Figure 5.18: Chat playground in Azure OpenAI Studio

3. The **Setup** panel, located on the left side of the **Chat playground** page, allows you to configure the system message. It's important to use the same system message during the fine-tuning process and also when utilizing the fine-tuned model. If a different system message is given, the results may vary from the ones you've calibrated during the fine-tuning process:

Azure OpenAI Studio > Chat playground

## Chat playground

 Import setup



The screenshot shows the Azure OpenAI Studio interface with the "Chat playground" selected. On the left, there's a "Setup" panel containing a "System message" input field where the text "You are a customer support assistant for a software company." is entered. This input field is highlighted with a red rectangle. Below it is an "Examples" section with a "Using examples" input field. At the top right of the main area, there are buttons for "Replay chat", "Clear chat", "Playground settings", "View code", and a "Show JSON" toggle switch. To the right of the setup panel is a "Start chatting" section with a bot icon and instructions to test the assistant. At the bottom right is a large text input field for user queries.

Figure 5.19: Provide system message

4. On the right side of the **Chat playground** page, you can select the **Deployment** option to decide which model will be used to generate responses. For this option, you should use the previously deployed fine-tuned model. You can also define the number of past messages to include as conversation history for future replies:

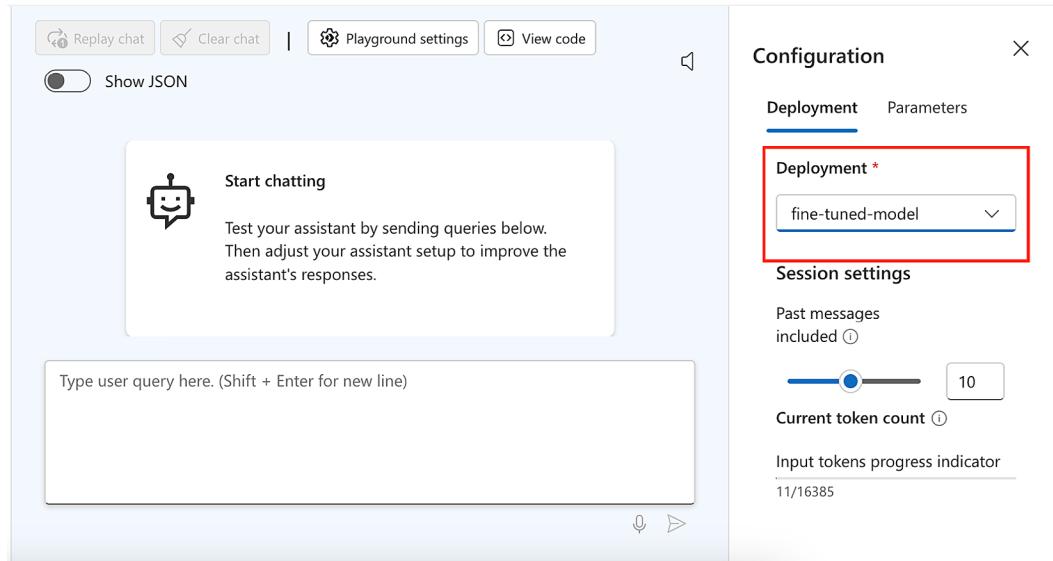


Figure 5.20: Select the model

5. Next, you can use the examples from the validation set to query the fine-tuned model. For instance, if you say, I am having trouble installing the software, the model should respond in a manner similar to, I'm sorry to hear that. Could you please tell me more about the issue you're facing? What error message are you seeing, if any?. This response is akin to the example in the validation set that stated, I'm sorry to hear that. Can you tell me what kind of error message you're seeing during the installation?:

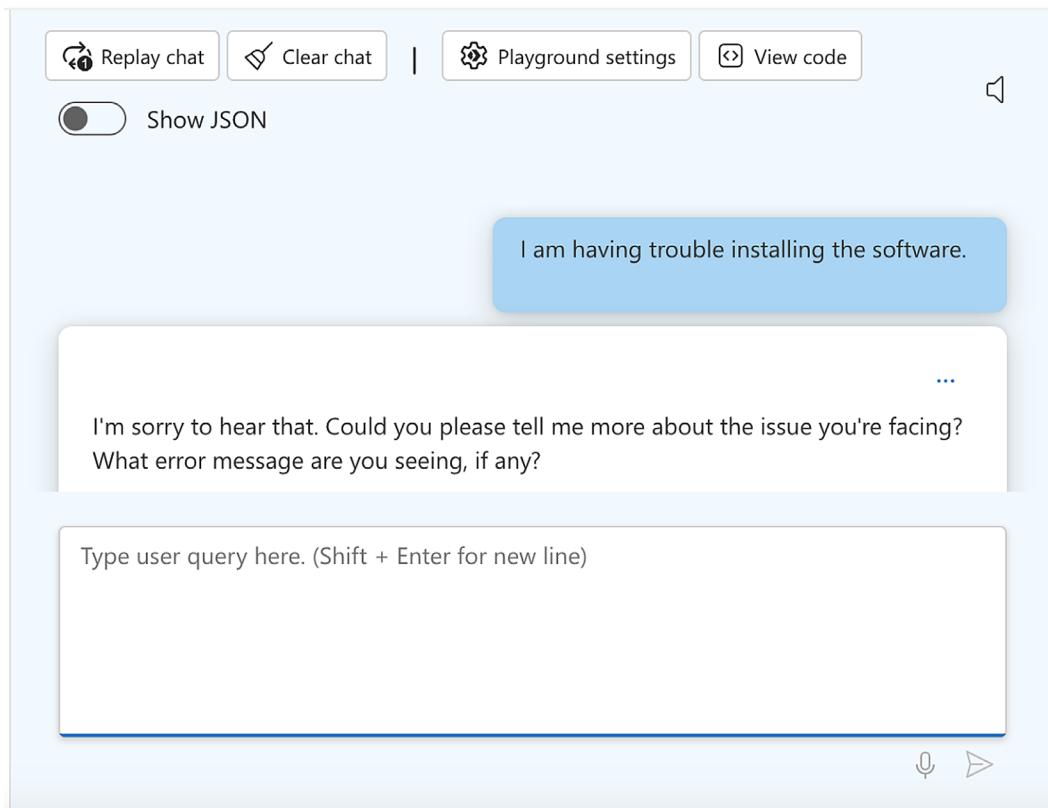


Figure 5.21: Query the fine-tuned model

## How it works...

Azure OpenAI Studio allows users to interact with their data conversationally. By utilizing AI models, including fine-tuned models, users can ask questions and receive informative replies.

## There's more...

When using your fine-tuned model in Azure OpenAI Studio, it's crucial to incorporate a system message during the procedure. This message establishes the context for the AI's replies, effectively directing its comprehension and interaction. The same system message should be used when employing your adjusted model. Using a different system message can lead to outcomes that differ from your initial adjustments.

## Testing your fine-tuned model with the Azure SDK

In this recipe, you will explore how to test your model using the Azure OpenAI SDK. You'll create a new Jupyter interactive window in Visual Studio Code, initialize an `AzureOpenAI` client, and generate chat completions.

By the end of this recipe, you will have learned how to use the Azure OpenAI SDK API to interact with a fine-tuned model and generate reliable responses.

### Getting ready

To get started with this section, ensure you have an Azure subscription and access to Azure OpenAI Service in your subscription. You'll also need an Azure OpenAI resource with a fine-tuned model deployed and Python 3.8 or later.

#### Note

This recipe uses Visual Studio Code with the Jupyter extension installed to run the code and visualize the results.

### How to do it...

Follow these steps to test your fine-tuned model with the Azure OpenAI SDK:

1. Create a new Jupyter interactive window in Visual Studio Code. Then import the required libraries and initialize an `AzureOpenAI` client with the specified base URL, API key, and API version in the notebook:

```
import os
from openai import AzureOpenAI

client = AzureOpenAI(
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-02-01"
)
```

2. Then, you must use the Azure OpenAI SDK API to generate a chat completion with a fine-tuned model. The `client.chat.completions.create` method is employed, taking the model name and conversation information as parameters. The conversation information includes a system message setting the context and a user message. The response, which is the model's response to the user's message, is then printed to the console:

```
response = client.chat.completions.create(  
    model="fine-tuned-model",  
    messages=[  
        {"role": "system", "content": "You are a customer  
support assistant for a software company."},  
        {"role": "user", "content": "I am having trouble  
installing the software."}  
    ]  
)  
  
print(response.choices[0].message.content)
```

3. Upon execution, the response you receive will be printed to the console. Given that the same questions were posed from the validation set in the previous section, the answers will be similar:

```
I'm sorry to hear that. Could you please tell me the exact error message you are seeing during the installation process?
```

Figure 5.22: Response from the model

## How it works...

Testing your fine-tuned model with the SDK involves utilizing the Azure OpenAI SDK API to generate a completion for a chat conversation. The Python code initializes an `AzureOpenAI` client using the endpoint, an API key, and a deployment name, and then creates a completion for a chat conversation, which consists of a system message setting the context and a user message from the validation dataset.

# 6

# Building AI Solutions with Prompt Flow

**Prompt flow** is a GUI-based LLM application development tool designed by Microsoft to streamline the development of AI applications. Prompt flow includes a growing list of tools that help you integrate LLMs with Python code, vector databases, Bing search results, Google search results, and more. You can find prompt flow as an open source project on **GitHub** and within both **Azure AI Studio** and **Azure Machine Learning**. For this chapter, you will focus on using prompt flow within Azure AI Studio. Azure AI Studio is a service that brings together different AI services from Microsoft, including **Azure OpenAI Service**, **Azure AI Speech**, and **Azure AI Vision**.

By the end of this chapter, you will be able to connect Azure OpenAI Service to Azure AI Studio, use prompt flow to adapt sample flows to your own projects, and build LLM-based applications from scratch. You will also be able to evaluate your application using a variety of metrics based on output quality, grammar, relevance, and completeness. Furthermore, you will feel confident in using prompt flow to create proofs of concept, minimum viable products, and production-ready applications.

This chapter covers the following recipes:

- Connecting Azure AI Studio to Azure OpenAI Service
- Using sample flows to build an AI application
- Building generative AI applications

## Technical requirements

To follow along with the examples in this chapter, you will need the following:

- An Azure subscription
- An Azure OpenAI Service resource
- You can find the scripts for this chapter in the GitHub repository at <https://github.com/PacktPublishing/Azure-OpenAI-Cookbook/tree/main/Chapter6>.  
The Chapter 6 folder contains all the notebooks for this chapter.

## Connecting Azure AI Studio to Azure OpenAI Service

In this recipe, you will learn how to access Azure AI Studio and connect it to Azure OpenAI Service. This will allow you to use prompt flow in later recipes.

### Getting ready

All you need for this recipe is an Azure OpenAI Service resource and an Azure subscription.

### How to do it...

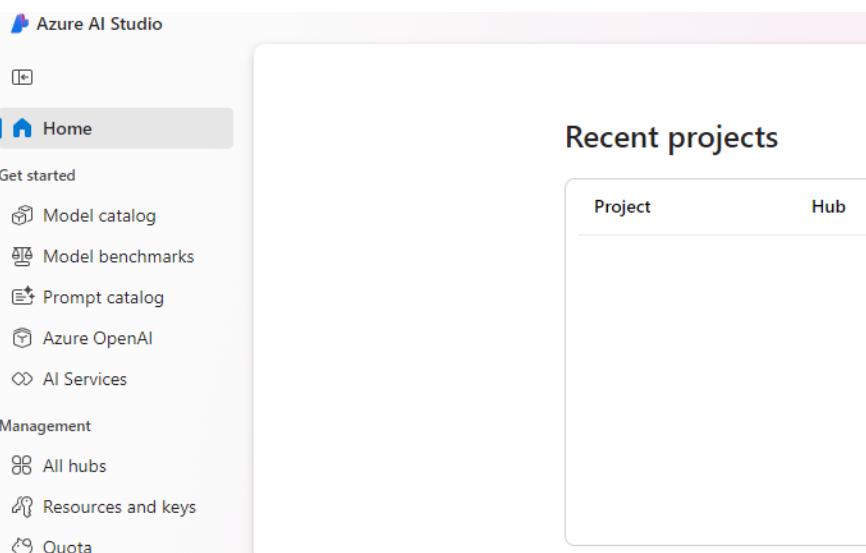
Follow these steps to access Azure AI Studio and create a connection between it and Azure OpenAI Service:

1. Open a web browser and navigate to <https://ai.azure.com>. This is a direct link to Azure AI Studio.

#### Note

As of the publication of this book, Azure AI Studio just became generally available after the Microsoft Build 2024 developer conference. However, it still has many features in public preview. This means that these features are still in development and liable to change.

2. When you land into Azure AI Studio, you will see the following screen listing your recent projects, along with nine tabs on the left-hand side of your screen, as shown in *Figure 6.1*. Click on the **All hubs** tab under **Management**:



Project 6.1: Azure AI Studio landing screen and left-hand navigation

3. You will now have to create a **hub** resource. With hub resources, teams can collaboratively build and manage AI applications, and managers can monitor costs and govern access and usage. Click **+ New hub** in the top-left corner of your screen:

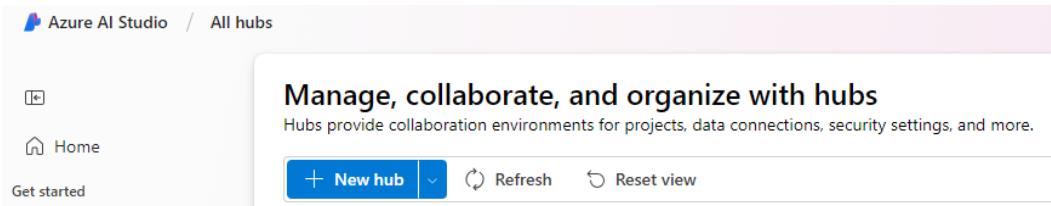


Figure 6.2: Creating a hub

4. Fill out the **Hub name** field, giving it a name of your choice. Fill in the fields for **Subscription**, **Resource group**, **Location**, and **Connect Azure AI Services or Azure OpenAI** as shown in *Figure 6.3*. Leave **Connect Azure AI Search** on its default setting of **Skip connecting**. Click **Next**:

The screenshot shows the 'Create a new hub' wizard. Step 1: Create a hub. It includes a navigation bar with 'Create a hub' (highlighted in blue) and 'Review and finish'. The main form has the following fields:

- Create a hub for your projects**: A hub is the collaboration environment for your team to share your project work, model endpoints, compute, (data) connections, and security settings.
- Hub name \***: my-azure-ai-hub
- Subscription \***: my-subscription
- Resource group \***: (new) my-resource-group
- Location \***: East US
- Connect Azure AI Services or Azure OpenAI \***: (new) ai-my-azure-ai-hub
- Connect Azure AI Search**: Skip connecting

At the bottom are 'Next' (highlighted in blue), 'Create', and 'Cancel' buttons.

Figure 6.3: Create a hub resource

5. Click **Create**. You have now connected Azure AI Studio to Azure OpenAI Service.

## How it works...

Azure OpenAI Service, Azure OpenAI Studio, Azure AI Studio, and hubs are separate services and concepts. Azure OpenAI Service lets you deploy and manage OpenAI models such as GPT-4 and GPT-3.5 Turbo. Azure OpenAI Studio is the GUI-based web interface for Azure OpenAI Service.

Azure AI Studio is a GUI-based web interface that interconnects all Azure AI resources together, including Azure OpenAI Service. A hub is a collaboration environment for teams to work together within Azure AI Studio. Without a hub, it's impossible to use Azure AI Studio.

## There's more...

When creating a hub, you're also creating numerous other resources to go along with it automatically. These resources include the following:

- **Azure Storage account** for storing data.
- **Azure Key Vault** for storing secrets.

**Note**

**Secrets** are another word for passwords.

Two other resources will be built automatically when you create your first application. These include:

- **Azure Container Registry** for building and storing container images for your applications.
- **Application Insights** for monitoring your applications.

**Note**

No matter how many applications you build, your hub will only be associated with a single instance of both Azure Container Registry and Application Insights.

To obtain direct access to these resources, follow these steps:

1. Click on the **All hubs** tab under **Management** on the left-hand side of your screen.
2. Click your newly created hub.
3. Under **Hub properties** on the right-hand side of your screen, click **Manage in Azure Portal** as shown in *Figure 6.4*. This will take you out of Azure AI Studio to your hub management screen in Azure.

## Hub properties

Name	Location
my-azure-ai-hub 	eastus
Subscription	Resource Group
SI Capital Data Science DevTest  	my-resource-group  
<a href="#">Manage in Azure Portal</a> 	
<a href="#">Get API endpoints and keys</a>	
<a href="#">View subscription quota</a>	

Figure 6.4: Hub properties

## See also

Here's an overview of connections in Azure AI Studio: <https://learn.microsoft.com/azure/ai-studio/concepts/connections>.

Here's a how-to-connect guide that covers a wide range of Azure services: <https://learn.microsoft.com/azure/ai-studio/how-to/connections-add?tabs=azure-ai-search>.

## Using sample flows to build an AI application

Prompt flow in Azure AI Studio contains many sample flows that you can adapt to your own projects. This is one of the ways in which prompt flow helps streamline application development. In this section, you will create your first prompt flow using the **Chat with Wikipedia** sample flow.

## Getting ready

In addition to your Azure OpenAI Service resource, you will also need a deployed GPT-3.5 Turbo model as taught in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*. You will also need to have created a hub resource and connected it to Azure OpenAI Service as outlined in the previous recipe.

## How to do it...

Before cloning a sample prompt flow to use in your own project, you must first create a project. A **project** is hosted within a hub and gives you a space to organize, save, and iterate your work while building AI applications.

Follow these steps to create a project and build a prompt flow using the **Chat with Wikipedia** sample flow:

1. Open a web browser and navigate to <https://ai.azure.com> to go to **Azure AI Studio**.
2. In Azure AI Studio, click **Home** on the left-hand side of your screen.
3. Click **+ New project** on the top-right corner of your screen.
4. For **Project name**, enter **Wikipedia-Chat**.
5. Select your hub from the dropdown under **Hub**.
6. Click **Create a project**. After the project is created, you should see the screen shown in *Figure 6.5*:

### Wikipedia-Chat

Pick up where you left off

The screenshot shows the Azure AI Studio interface for the 'Wikipedia-Chat' project. At the top, there's a search bar and filter/column options. The main area displays a table with columns: Resource type, Name, Status, Quick actions, Last viewed, and Created by. A message below the table states, "There are no recently viewed items to display". Below the table, there are three documentation cards:

- Build and deploy a question and answer copilot with prompt flow in Azure AI Studio**: This card includes a link to the article and a note: "Use generative AI and prompt flow to build, configure, and deploy a copilot for your retail company called Contoso."
- Build and deploy a question and answer copilot with the Azure AI CLI and SDK**: This card includes a link to the article and a note: "Use the Azure AI CLI and SDK to build, configure, and deploy a copilot for your retail company called Contoso Trek."
- Retrieval augmented generation and indexes**: This card includes a link to the article and a note: "This article talks about the importance and need for Retrieval Augmented Generation (RAG) and Index in generative AI."

Figure 6.5: Azure AI Studio project screen

7. Click **Prompt flow** from the left-side menu under **Tools**.
8. Click **+ Create**.
9. You should now see the prompt flow creation screen shown in *Figure 6.6*. In the Chat with Wikipedia box, click **Clone**:

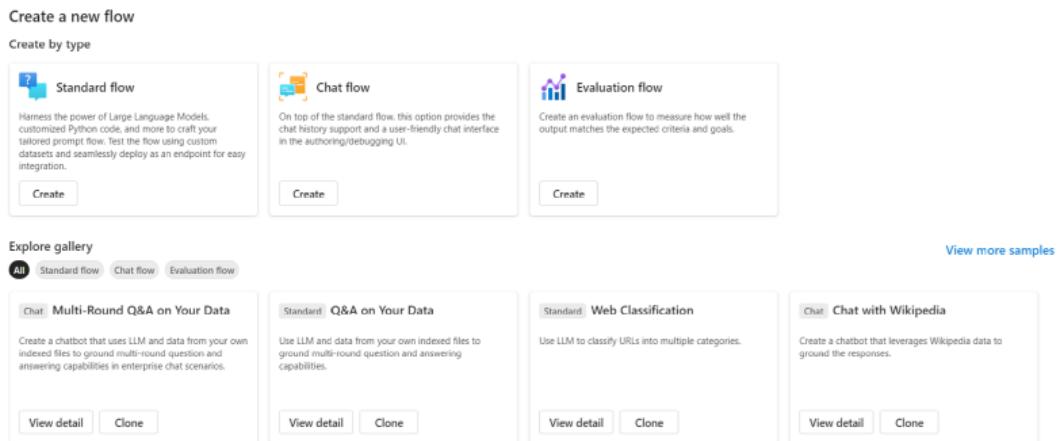


Figure 6.6: Prompt flow creation screen

10. A new box will pop up allowing you to name a folder within your **Azure Storage Account File Share** storage. Enter **Wikipedia** for **Folder name**, then click **Clone**.
11. In some cases, you may get a **CloudDependencyPermission** error telling you to grant workspace/registry read access to the source storage account. In this case, you must navigate to your storage account, click **Access control (IAM)** and assign the **Reader** role to your Wikipedia-Chat project. You will now see the prompt flow screen shown in *Figure 6.7*. Click **Start compute time** at the top-right corner of your screen to access a dropdown list and click **Start compute session**. This will give you a compute resource with which you can run your prompt flow. Runtimes take about 30 seconds to 3 minutes to spin up, although sometimes it can take longer to secure a compute resource:

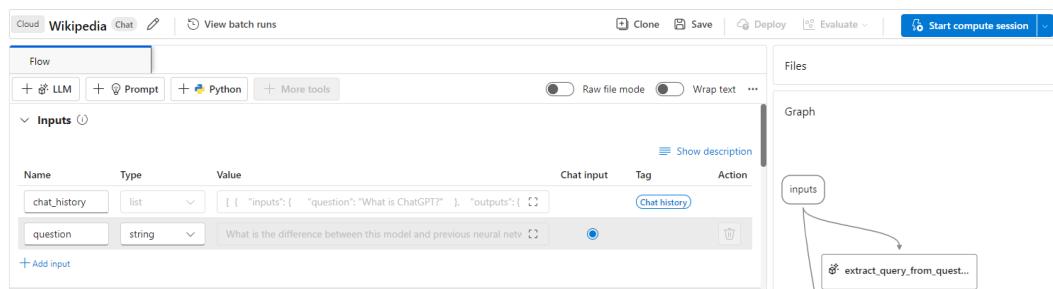


Figure 6.7: Prompt flow screen

### Note

You can also install Python packages on your compute by clicking **Compute session running** and selecting **Install packages from requirements.txt**. Computers do, however, come with hundreds of Python packages pre-installed.

12. Prompt flows are divided into multiple parts called **nodes**, represented by the graph on the right side of your screen as shown in *Figure 6.8*. Each node uses a specific tool that takes in an input and produces an output.

You need to connect your deployed Azure OpenAI Service models to each node that uses an LLM tool separately. To do so, continue with the following instructions:

13. Click the `extract_query_from_question` box in the **Graph** box found on the left side of your screen to navigate to the `extract_query_from_question` node:

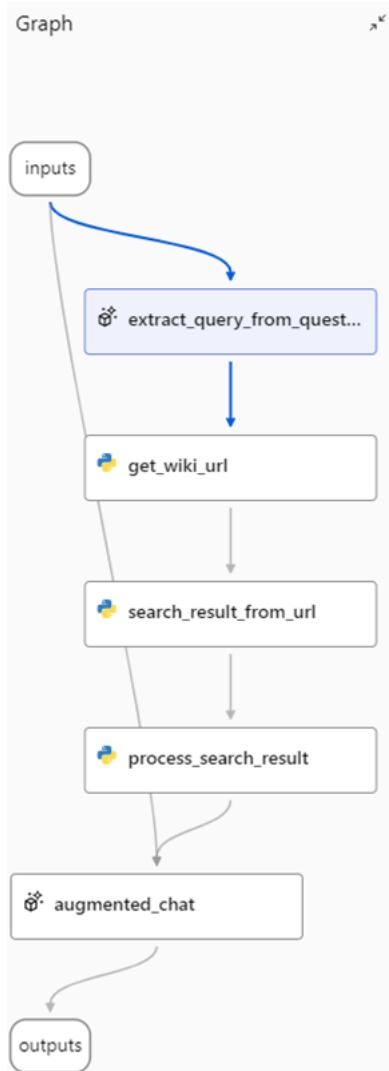


Figure 6.8: Prompt flow graph

14. Click the **Connection** dropdown and select the connection to your Azure OpenAI Service resource. You should now see the screen shown in *Figure 6.9*:

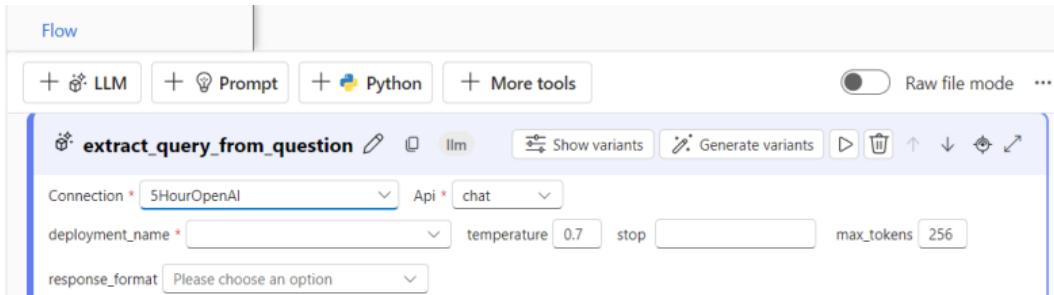


Figure 6.9: Connect to your Azure OpenAI Service model deployment

15. Select your GPT-3.5 Turbo deployment from the **deployment\_name** dropdown.
16. Select {"type": "text"} from the **response\_format** dropdown.
17. Click **Save**.
18. Scroll down in the **Graph** box on the left side of your screen and click the `augmented_chat` node. Like the `extract_query_from_question` node, this uses an LLM tool.
19. Repeat *Steps 13–16* of this recipe to connect the `augmented_chat` node to your deployed model.

#### Note

You can select different model deployments for different sections of a single prompt flow. This way, you can use cheaper models such as GPT-3.5 Turbo for simpler tasks and more expensive models such as GPT-4 for complex tasks.

20. Click **Chat** in the top-right corner of your screen.
21. Ask the model to respond to the following prompt:

```
List 3 different machine learning techniques used for  
forecasting and a one-sentence explanation for each.
```

22. Read the response. Notice that it contains multiple links to the appropriate Wikipedia pages. You have now successfully created a prompt flow chat. It should look something like this, although results will vary:

Linear Regression: A statistical approach for modelling the relationship between a dependent variable and one or more independent variables, often used for forecasting continuous outcomes (SOURCES: [https://wikipedia.org/w/index.php?search=Machine learning](https://wikipedia.org/w/index.php?search=Machine%20learning)).

Decision Trees: A type of machine learning model that makes predictions by learning simple decision rules inferred from the data features, often used for both regression and classification problems in forecasting (SOURCES: [https://wikipedia.org/w/index.php?search=Machine learning](https://wikipedia.org/w/index.php?search=Machine%20learning)).

Neural Networks: Comprising algorithms modeled after the human brain, these complex networks learn from vast amounts of data and are particularly effective in forecasting where relationships between inputs and output are nonlinear (SOURCES: [https://wikipedia.org/w/index.php?search=Machine learning](https://wikipedia.org/w/index.php?search=Machine%20learning)).

Creating a prompt flow from scratch can be quite intimidating if you're just beginning to use them. By using the sample flows in Azure OpenAI Studio, you can easily create prompt flows that chat with Wikipedia, your own data, or search results from the web.

## How it works...

Prompt flows are made up of a series of steps called nodes and use a different tool within each node. Understanding each step in the **Chat with Wikipedia** flow will give you a deeper understanding of how prompt flow works. *Figure 6.10* shows all the steps involved in the prompt flow you created. Notice there's now a green checkmark on each node to indicate a successful run:

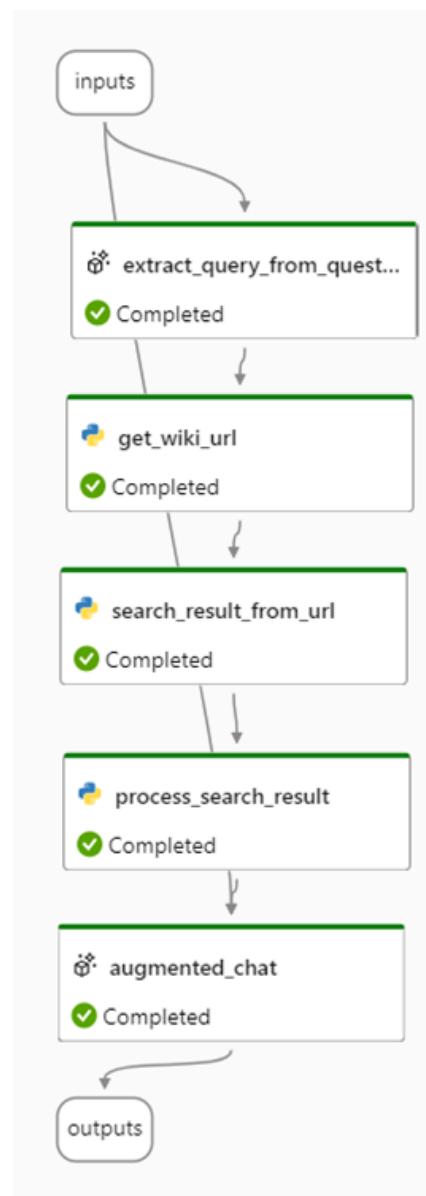


Figure 6.10: Chat with Wikipedia diagram

Here's a breakdown of these steps:

- **Flow inputs** are input variables that are processed within the flow. A Wikipedia chat has two variables, `chat_history` and `question`. When you use the chat window in the Wikipedia prompt flow, you are altering the `question` flow input variable.

- **LLM tools** ensure you can add both system and user prompts. The `extract_query_from_question` node is an LLM tool, and its system message is as follows:

You are an AI assistant reading the transcript of a conversation between an AI and a human. Given an input question and conversation history, infer user real intent.

The conversation history is provided just in case of acoreference (e.g. "What is this?" where "this" is defined in previous conversation).

Return the output as query used for next round user message.

- **Python tools** ensure you can run Python code within prompt flow. The next three nodes, `get_wiki_url`, `search_result_from_url`, and `process_search_result`, are all Python nodes to get URLs from Wikipedia, return search results, and process search results for the `augmented_chat` node.
- Next, `augmented_chat` is another LLM tool. Its system message contains anti-hallucination prompt engineering techniques that you learned in *Chapter 2, Engineering Useful Prompts*. Its system message is as follows:

You are a chatbot having a conversation with a human.

Given the following extracted parts of a long documentand a question, create a final answer with references ("SOURCES").

If you don't know the answer, just say that you don't know.  
Don't try to make up an answer.

ALWAYS return a "SOURCES" part in your answer.

- **Flow outputs** are simply the end results produced by executing the flow.

Now that you understand how prompt flow is made up of nodes that use different tools, you are ready to build your own flows from scratch.

## There's more...

You can also deploy your prompt flow to an endpoint that you can access via code. This way, you can embed prompt flows within your own code that will return output in real time. Here's how to do it:

1. Within Azure AI Studio, click **Home** on the left-hand side of your screen.
2. Click your project, **Wikipedia-Chat** under **Recent projects**.
3. Within your project, click on your prompt flow, **Wikipedia**.
4. Click **Start compute time** at the top-right corner of your screen to access a dropdown list and click **Start compute session**. Wait 30 seconds to 3 minutes for the runtime to start up, as you will be unable to proceed to the next step without it.
5. Click **Deploy** on the right side of your screen.

6. You will see the deployment creation screen as shown in *Figure 6.11*. Enter wikipedia-chat for **Endpoint name**.
7. Select **Standard\_DS3\_v2** for **Virtual machine**.

#### Note

Prompt flow deployments require a virtual machine to be up and running 24/7. You may have to increase your virtual machine quota to complete this step. Learn more about it here: <https://learn.microsoft.com/azure/machine-learning/how-to-manage-quotas?view=azureml-api-2>.

#### Basic settings

Deploy your flow to a managed online endpoint for real-time inference. [Learn more](#) ⓘ

##### Endpoint

New  Existing

##### Endpoint name \* ⓘ

wikipedia-chat-Inyxv

##### Deployment name \* ⓘ

wikipedia-chat-Inyxv-1

##### Virtual machine \* ⓘ

Standard\_DS3\_v2 4 Cores, 14 GB (RAM), 28 GB (Disk), \$0.28/hr

##### Instance count \* ⓘ

2

##### Inferencing data collection (preview) ⓘ

Enabled

**Review + Create**

Back

Next

Figure 6.11: Deployment creation screen

8. Click **Review + Create**.
9. Click **Create** and wait 5 to 10 minutes before proceeding to the next step.
10. To see your deployed endpoint, click **Deployments** on the left side of your screen.
11. Click on your deployment. The name of the deployment will begin with **Wikipedia-chat** followed by a dash and then some letters.

12. Click the blue link with your endpoint name. You should see a **Deployment attributes** screen similar to *Figure 6.12*:

The screenshot shows the 'Deployment attributes' screen for a deployment named 'wikipedia-chat-rvxzs-1'. The 'Details' tab is selected. The left panel displays deployment details: Name (wikipedia-chat-rvxzs-1), Provisioning state (Succeeded), Created by (Dennis Sawyers), and Traffic allocation (100%). It also shows a 'Flow' section for 'Source flow' and a 'Model data collection' section indicating it's Enabled with links for Inputs, Outputs, and App traces. The right panel shows the 'Endpoint' configuration, including the URL (<https://wikipedia-chat-rvxzs.canadaeast.inference.ml.azure.com/score>), Swagger URI (<https://wikipedia-chat-rvxzs.canadaeast.inference.ml.azure.com/swagger>), Public network access (Enabled), Compute type (Managed), Authentication type (Key), and a Primary key field with a 'Regenerate' button.

Figure 6.12: Deployment attributes screen

13. Click **Consume**.
14. You will now see a screen that lets you copy code in JavaScript, Python, C#, or R. Click **Python**.
15. Click the **Copy** icon. You can now easily paste the code into any codebase you wish.
16. Delete the deployment by navigating to the **Deployment attributes** screen and clicking **Delete deployment** to save costs.

You now know how to deploy a prompt flow to a real-time endpoint and integrate that code with JavaScript, Python, C#, or R. You can also integrate the code in other languages by making a POST request to the API endpoint similar to how the sample code does for the four languages.

## See also

Here's an overview of prompt flow including sample flows: <https://learn.microsoft.com/azure/ai-studio/how-to/prompt-flow#flows>.

Here's documentation for deploying prompt flows to endpoints: <https://learn.microsoft.com/azure/ai-studio/how-to/flow-deploy?tabs=azure-studio>.

## Building generative AI applications

In this recipe, you will delve into the intricacies of building GenAI applications using prompt flow from scratch. You will create a standard flow, connect to OpenAI, and make a simple program that uses LLMs to profile web-hosted tabular data. You will learn how to add new tools within a prompt flow, pass input from one step to the next, and create AI applications in a streamlined, low-code way.

### Getting ready

In addition to your Azure OpenAI Service resource, you will also need a deployed GPT-3.5 Turbo model as taught in *Chapter 1, Introducing Azure OpenAI and Building Your First Chatbot*. A deployed GPT-4 model is also required. You will need to have a hub resource connected to Azure OpenAI Service as outlined in the first recipe of this chapter.

### How to do it...

For this exercise, you will first create a new project before configuring your prompt flow:

1. Open a web browser and navigate to <https://ai.azure.com> to go to Azure AI Studio.
2. Within Azure AI Studio, click **Home** on the left-hand side of your screen.
3. Click **+ New project** on the top-right corner of your screen.
4. Under **Project name**, enter **Data-Profiler**.
5. Click **Create a project**.
6. Click **Prompt flow** from the left-side menu.
7. Click **+ Create**.
8. You should now see the prompt flow creation screen shown earlier in this chapter in *Figure 6.6*. In the **Standard flow** box, click **Create**.
9. Enter **Profiler** for **Folder name**, then click **Create**. This will create a new folder within your Azure storage account. This step can sometimes take 1 to 2 minutes.

#### Note

In some cases, you may get a **CloudDependencyPermission** error telling you to grant workspace/registry read access to the source storage account. In this case, you must navigate to your storage account, click **Access control (IAM)** and assign the **Reader** role to your **Data-Profiler** project.

10. Click **Start compute time** at the top-right corner of your screen to access a dropdown list and click **Start compute session**.

11. This will give you a compute resource with which you can run your prompt flow. Runtimes take about 30 seconds to 3 minutes to spin up, although sometimes it can take longer to secure a compute resource.
12. You will see the graph shown in *Figure 6.13*. Notice that the input connects to the LLM tool, which connects to the Python tool, which then produces output. You can click the different boxes to practice navigating to different parts of the prompt flow with the graph. Click the **Inputs** box:

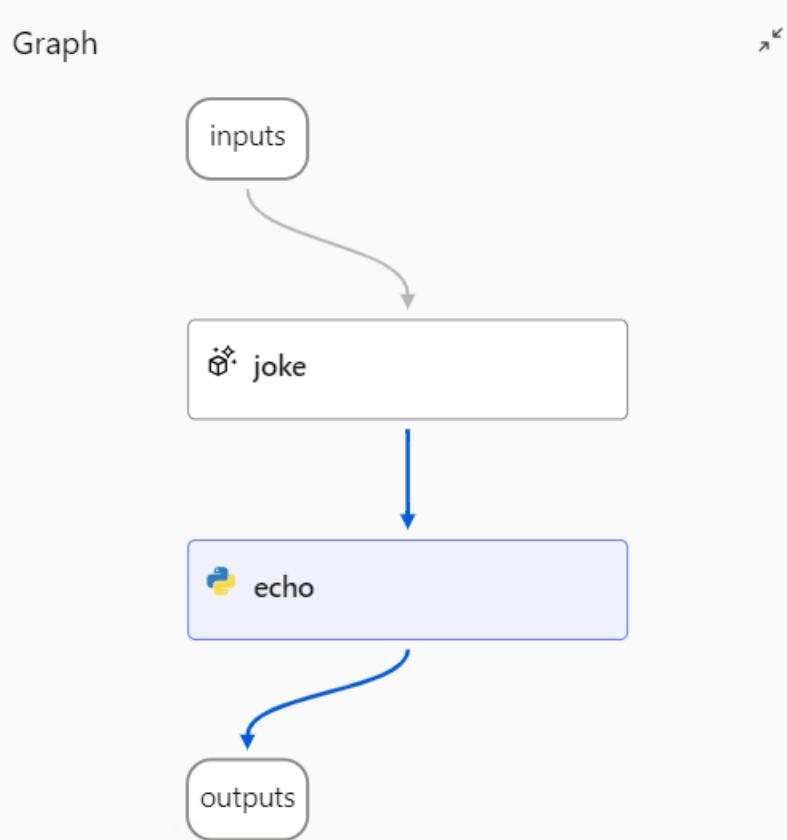


Figure 6.13: Standard flow graph upon creation

13. Within the **Inputs** section, change the first variable from `topic` to `url`.
14. Within the **Inputs** section, enter <https://data.lacity.org/api/views/2nrs-mtv8/rows.csv?accessType=DOWNLOAD> into the **Value** field. This is a direct link to Los Angeles crime data in a CSV format.
15. Click the `echo` box on the graph to navigate to the Python section.
16. Click the pencil icon shown in *Figure 6.14*, rename the node from `echo` to `web_data_pull`, and click the disk-shaped save icon that appears directly to the right of the name:



Figure 6.14: Renaming a prompt flow node

17. Within the same `web_data_pull` node, change the value of the input variable to  `${inputs.url}`, as shown in *Figure 6.15*. Notice how the graph changes:

Name	Type	Value
input	string	<code> \${inputs.url}</code>

> Activate config

Figure 6.15: Changing the Python node's input variable

18. Replace the Python code with the following code to pull CSV data from web URLs:

```
from promptflow import tool
import pandas as pd

@tool
def fetch_data_summary(input: str) -> str:
    """
    Fetches CSV data from a specified URL, returns the column names and the first row of data as a string.

    Args:
        input (str): The URL from which to fetch the CSV data.

    Returns:
        str: A string containing the column names and the first row of data.
    """
    df = pd.read_csv(input)
    # Combine column names and the first row into a single string
    summary = 'Columns: ' + ', '.join(df.columns) + ' | First Row: ' + ', '.join(df.iloc[0].astype(str))
    return summary
```

19. Click **Validate and parse input**. You should see this message: **Validation and parsing input completed successfully**. Your Python node is now set up.
20. Click the joke box on the graph to navigate to the LLM node of the prompt flow.
21. Rename `joke` to `data_profiler` by clicking the pencil icon. Then, hit the save icon.
22. Click the **Connection** dropdown within the `data_profiler` node and select the connection to your Azure OpenAI Service resource.
23. Select your GPT-3.5 Turbo deployment from the **deployment\_name** dropdown.
24. Select `{"type": "text"}` from the **response\_format** dropdown.
25. Click **Save** on the top-right corner of your screen.
26. In the **Prompt** section of the `data_profiler` node, replace the sample prompt with the following:

```
{# Prompt is a jinja2 template that generates prompt for LLM #}

# system:
You are a data profiler. A user will provide you with a string
containing column names and the first row of a dataframe. Your
job is to list the columns and corresponding data types. If
you cannot identify a data type from the data, respond with
"unknown" for that column's data type.

# user:
{{data_input}}
```

27. Click **Validate and parse input** and confirm that it completed successfully. When it does, the phrase **Validation and parsing input completed successfully** will pop up on your screen next to a green checkmark.
28. In the **Inputs** section of the `data_profiler` node, select  `${web_data_pull.output}` from the dropdown as shown in *Figure 6.16*. This pulls in output from the `web_data_pull` Python node as input to this node:

The screenshot shows the configuration interface for a 'data\_profiler' node. At the top, there's a dropdown labeled 'Inputs' with a green checkmark next to it, indicating that 'Validate and parse input' has been successful. Below this, there's a table with three columns: 'Name', 'Type', and 'Value'. The 'Name' column has a single entry 'data\_input'. The 'Type' column shows 'string'. The 'Value' column contains the expression '\${web\_data\_pull.output}' with a small edit icon to its right. At the bottom left, there's a link 'Activate config'.

Name	Type	Value
data_input	string	\${web_data_pull.output} 

Figure 6.16: Changing the input of the LLM tool

29. Scroll up to the **Outputs** section.
30. Rename the joke variable to final\_output under **Name**.
31. Select \${data\_profiler.output} under **Value** for the final\_output variable. This ensures that the output from your LLM node is the output for this prompt flow.
32. Click **Save** in the top-right corner of your screen. Your graph should look identical to *Figure 6.17*:

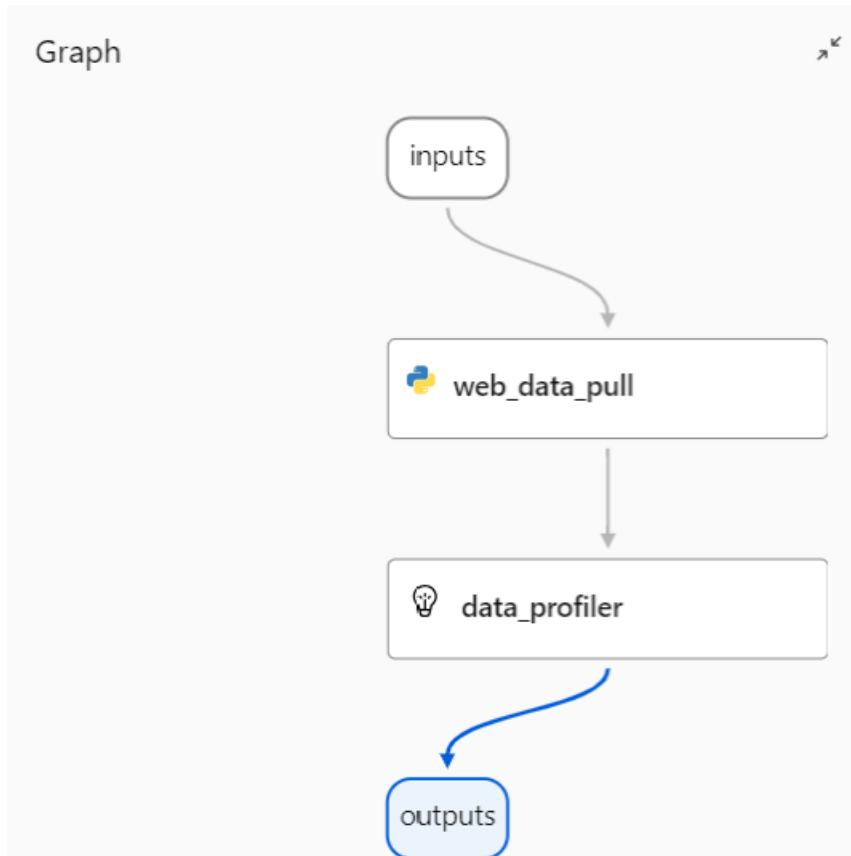


Figure 6.17: Completed prompt flow graph

33. Click **Run** in the top-right corner of your screen.
34. Wait a few minutes for **Run completed** to appear in the top-left corner of your screen. Once it appears, click the **View outputs** box found in the top-left corner of your screen.

35. View your final output. Since this is a GenAI workflow, results may vary slightly every time you run the flow but should resemble *Figure 6.18*:

The screenshot shows a user interface for managing prompt flows. At the top, there are tabs for 'Outputs' (which is selected), 'Logs', 'Metrics', and 'Trace'. Below the tabs is a search bar and an 'Export' dropdown menu. The main area displays a table with the following columns: '#', 'inputs.url', 'Status', and 'final\_output'. There is one row in the table:

#	inputs.url	Status	final_output
0	<a href="https://data.lacity.org/api/vie/ws/2nrs-mtv8/rows.csv?accessType=DOWNLOAD">https://data.lacity.org/api/vie/ws/2nrs-mtv8/rows.csv?accessType=DOWNLOAD</a>	Completed	Column: DR_NO, Data Type: int Column: Date Rptd, Data Type: datetime Column: DATE OCC, Data Type: datetime Column: TIME OCC, Data Type: int Column: AREA, Data Type: int Column: AREA NAME, Data Type: str Column: Rpt Dist No, Data Type: int Column: Part 1-2, Data Type: int Column: Crm Cd, Data Type: int Column: Crm Cd Desc, Data Type: str Column: Mocodes, Data Type: unknown Column: Vict Age, Data Type: int Column: Vict Sex, Data Type: str Column: Vict Descent, Data Type: str Column: Premis Cd, Data Type: int Column: Premis Desc, Data Type: str Column: Weapon Used Cd, Data Type: float Column: Weapon Desc, Data Type: str Column: Status, Data Type: str Column: Status Desc, Data Type: str Column: Crm Cd 1, Data Type: float Column: Crm Cd 2, Data Type: float Column: Crm Cd 3, Data Type: unknown Column: Crm

Figure 6.18: Final output from your prompt flow

## How it works...

Prompt flow lets you build AI applications by chaining together a series of tools on its design canvas. As you add and rearrange tools in the prompt flow, the graph on the right side of your screen is automatically constructed. As your flow becomes more and more complex, the graph can also be used to navigate to different nodes of the flow.

In this example, you chained together a Python node with an LLM node to create a data-profiling AI application. This is a typical example of a standard flow. **Standard flows** are flexible and allow you to use a wide range of tools to develop applications. Commonly, these are non-chat applications such as text summarization, translation, data profiling, data analysis, report generation, or sentiment analysis.

**Chat flows** are designed specifically for conversational application development. Unlike standard flows, chat flows provide chat history management and enhanced inputs/outputs designed specifically for conversational assistants. Common scenarios for chat flows include customer support bots, educational bots, and virtual personal assistants.

Lastly, **evaluation flows** let you take the output of other prompt flows and score them on a series of metrics such as groundedness, relevance, coherence, and fluency. **Groundedness** measures how well your flow's responses can be verified against source text. **Relevance** measures how related a flow's output is to the input prompt. **Coherence** measures how human-like a flow's output sounds. **Fluency** evaluates grammar and syntax. A deeper explanation of these metrics and others can be found in this documentation: <https://learn.microsoft.com/azure/machine-learning/prompt-flow/concept-model-monitoring-generative-ai-evaluation-metrics?view=azureml-api-2>.

Each prompt begins with a single input node and ends with a single output node. Within these nodes, however, you can have multiple inputs and multiple outputs. These inputs and outputs are then fed into the various nodes. One node's output can be used as another node's input, and each node can utilize multiple inputs. Each node consists of a single prompt flow tool, and each tool type has defined outputs.

The tool types at the time of writing are as follows:

- LLM tool
- Prompt tool
- Python tool
- Azure OpenAI GPT-4 Turbo with Vision tool
- OpenAI GPT-4V tool
- Index Lookup tool
- Vector Index Lookup tool
- Content Safety (Text) tool
- Faiss Index Lookup tool
- Vector DB Lookup tool
- Embedding tool
- Serp API tool

A detailed explanation of each of these tools can be found in this documentation: <https://learn.microsoft.com/azure/ai-studio/how-to/prompt-flow-tools/prompt-flow-tools-overview>.

Once you have constructed all your nodes and assigned inputs and outputs, you can easily chain together different nodes. A common example would be to have multiple LLM tools, with the output of one being used as input for another, so you can have a long string of prompt chains.

After your prompt flow is complete, you can deploy it to a REST endpoint and embed your flow within your code. By letting you complete all of these things in a single, GUI-based tool, prompt flow greatly simplifies the process of building AI applications for complex, multi-faceted projects.

## There's more...

You can also easily expand the scope and capabilities of your prompt flow by adding more tools to it. One useful way to expand the **Profiler** flow is to have your LLM analyze the dataset as a whole and suggest useful SQL queries for it. Here's how you do it:

1. Navigate to your **Data-Profiler** project in Azure AI Studio.
2. Click **Prompt flow** on the left side of your screen.
3. Click **Profiler** in the middle of your screen.
4. Within your **Profiler** prompt flow, click **Start compute time** at the top-right corner of your screen to access a dropdown list and click **Start compute session**. Wait 30 seconds to 3 minutes for it to spin up.
5. Click **+ LLM** in the middle of your screen. This will add an LLM tool to your flow.
6. Enter `generate_queries` for the name of the node and click **Add**.
7. Click the **Connection** dropdown within the `generate_queries` node and select the connection to your Azure OpenAI Service resource.
8. Select **chat** from the dropdown list for **Api**:

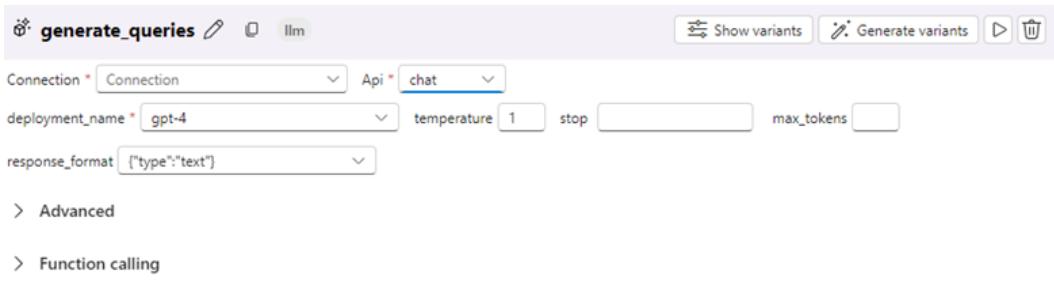


Figure 6.19: Connecting to Azure OpenAI Service

9. Select your GPT-4 deployment from the **deployment\_name** dropdown.
10. Select `{"type": "text"}` from the **response\_format** dropdown.

### Note

Even though this prompt outputs JSON, select text. It will still output JSON. While a JSON response format option exists, it's not yet available for most models.

11. Click **Save** in the top-right corner of your screen.

12. Enter these instructions into the **Prompt** section:

```
# system:  
You are tasked with generating SQL queries for datasets based on  
provided column names and their data types. When a user submits  
these details about a dataset, you should analyze the provided  
information to hypothesize what the dataset might contain and  
its potential uses.  
Based on this analysis, you will generate SQL queries that could  
be helpful for users wanting to extract meaningful information  
from the dataset. You will respond with a JSON object containing  
a brief description of what you believe the dataset represents  
and a list of useful SQL queries. Ensure your response adheres  
to the following JSON structure:  
{  
    "description": "This dataset likely contains...",  
    "queries": [  
        "SELECT ...",  
        "SELECT ...",  
        "..."  
    ]  
}  
# user:  
{profiled_data}}
```

13. Click **Validate and parse input** to generate the `profiled_data` input variable.
14. In the **Inputs** section of the `generate_queries` node, select `$(data_profiler.output)` from the dropdown. This will take the column names and data types generated by the `data_profiler` node as input.
15. Click **outputs** on the graph on the right side of your screen to navigate to the **Outputs** section.
16. Rename `final_output` to `columns_data_types`.
17. Click **+Add output**.
18. Name the new output `sql_queries`.

19. Select \${generate\_queries.output} from the **Value** dropdown list for **sql\_queries**. Your graph should look like *Figure 6.20*:

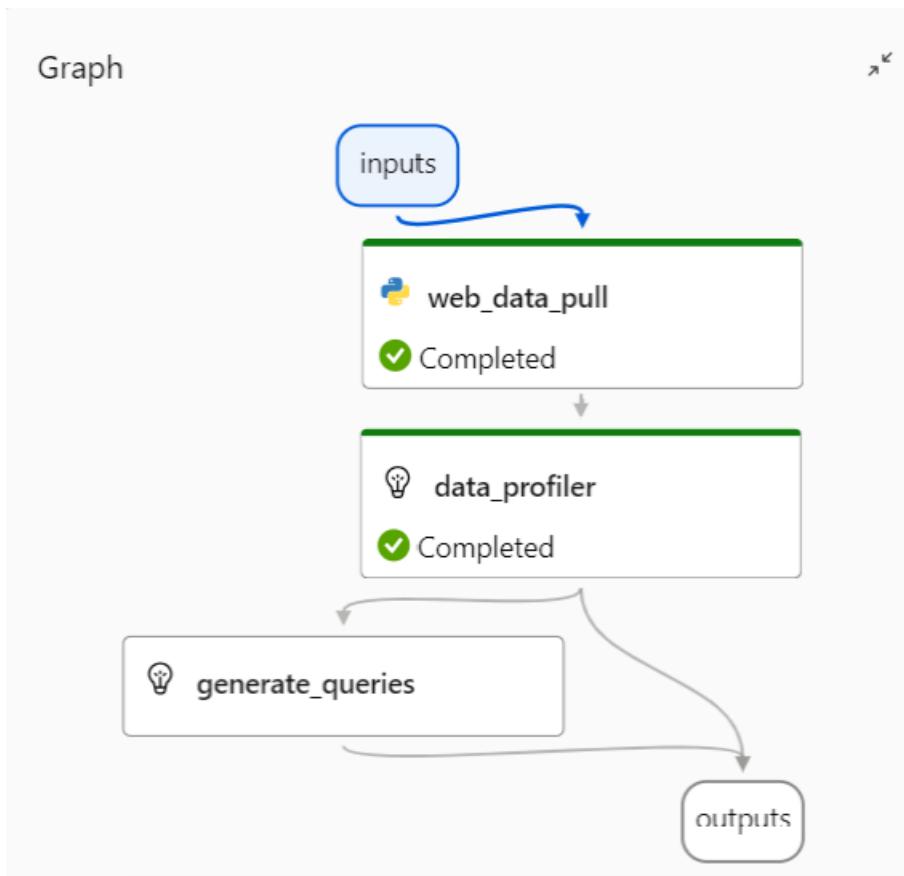


Figure 6.20: Updated graph

20. Click **Save**.
21. Click **Run**.
22. You have now completed this recipe and successfully added an additional LLM tool to your prompt flow. Wait a few minutes for **Run completed** to appear in the top-left corner of your screen. Once it appears, click the **View outputs** box found in the top-left corner of your screen.

23. View your output. These results will vary from one user to another, as you will get different queries each time you run it. However, your output should resemble the following:

```
{ "description": "This dataset likely contains records of crime incidents, including when and where they occurred, descriptions of the crimes, victim information, information on the premises where the crimes occurred, and possibly the weapons used. It might be used by law enforcement for crime analysis and statistics, and by researchers studying patterns in criminal activity.", "queries": [ "SELECT COUNT(*) FROM dataset; -- count total number of incidents", "SELECT `AREA NAME`, COUNT(`DR_NO`) FROM dataset GROUP BY `AREA NAME`; -- find number of incidents per area", "SELECT `Crm Cd Desc`, COUNT(`DR_NO`) FROM dataset GROUP BY `Crm Cd Desc`; -- classify incidents by crime code description", "SELECT AVG(`Vict Age`) FROM dataset; -- calculate average victim age", "SELECT `Vict Sex`, COUNT(`DR_NO`) FROM dataset WHERE `Vict Sex` IN ('M', 'F') GROUP BY `Vict Sex`; -- find number of male and female victims", "SELECT `DATE OCC`, COUNT(`DR_NO`) FROM dataset GROUP BY `DATE OCC`; -- find number of incidents per day", "SELECT `Premis Desc`, COUNT(`DR_NO`) FROM dataset GROUP BY `Premis Desc`; -- classify incidents by premise description", "SELECT AVG(`LAT`), AVG(`LON`) FROM dataset; -- find average coordinates of incidents", "SELECT `Weapon Desc`, COUNT(`DR_NO`) FROM dataset GROUP BY `Weapon Desc`; -- classify incidents by weapon description (if available)", "SELECT `Status Desc`, COUNT(`DR_NO`) FROM dataset GROUP BY `Status Desc`; -- classify incidents by status description (if available)" ] }
```

#### Note

While you can do this exercise with GPT-3.5 Turbo, it will not output formatted JSON.

You have now finished adding an LLM tool to your prompt flow. Try adding other tools to it and experiment to further your understanding. For example, add other Python nodes, and pull in and profile other web-based data sources. Once you get the hang of it, prompt flow will become a powerful tool to quickly construct useful AI applications.

#### See also

Here's a link to documentation showing how to develop a prompt flow: <https://learn.microsoft.com/azure/ai-studio/how-to/flow-develop>.

Here's how to deploy a model for real-time inferencing: <https://learn.microsoft.com/azure/ai-studio/how-to/flow-deploy?tabs=azure-studio>.

Here's how to build a flow to evaluate the output of your models: <https://learn.microsoft.com/azure/ai-studio/how-to/flow-develop-evaluation>.

# Index

## A

accuracy: 5-6, 35-36, 38-39, 57  
AI-generated: 7, 21, 27  
ai-studio: 134, 143, 150, 154  
api\_base: 63  
api\_key: 63, 66, 72, 101, 104, 122, 128  
APIs: 53, 102  
Azure OpenAI Studio: 9, 15, 18-19, 22, 24, 26, 28-29, 33, 36, 39, 42, 44, 48-50, 53, 55, 62, 65, 77-78, 83, 86-88, 93-94, 98-99, 104, 106-110, 113, 120, 122-124, 127-128, 130-135, 139, 141, 144, 151  
Azure OpenAI Service: 1-2, 8-9, 11, 18-19, 22, 26, 28, 39-40, 53, 58-65, 67, 69-73, 75-78, 107, 109, 122, 124, 128, 130-134, 136, 138, 144, 146, 151  
azure-ai-search: 134  
azure\_endpoint: 63, 66, 72, 101, 122, 128

## B

Blob: 26, 71, 76, 89-91, 93, 120  
byte-order: 108

## C

chain-of-thought: 28, 31, 43-47  
chatbot: 1-2, 4-5, 8-9, 14-15, 19, 20, 22-28, 33, 54, 59-60, 66, 109, 134, 141, 144  
citations: 94, 97, 105  
clone: 135-136  
codebase: 62, 143  
Cognitive: 66, 78, 83, 107  
configuration: 22, 29, 33, 36, 39, 42, 44, 48-50, 53, 55, 85, 96  
CORS: 90  
Cosmos DB: 23-24, 26

## D

data: 3-4, 6-7, 10, 20, 26, 28, 31-32, 35-43, 46-47, 50-52, 54, 66, 68, 71, 76-77, 83, 88-89, 91-95, 97-102, 104-109, 113, 115, 120, 123-124, 127, 133, 139, 144-147, 149, 152, 154  
database: 24, 42, 82, 101, 130  
DataFrame: 20, 100-101, 147  
data\_profiler: 144, 151, 147-148, 152  
data-profiling: 149  
deployment\_4: 62-64, 66, 70, 73-74  
deployment\_name: 138, 147, 151

**E**

Elasticsearch: 26  
embedding: 3, 40, 53, 77, 82-83, 87-88,  
91-93, 94, 98, 100-101, 104-105, 150  
epochs: 120, 123

**F**

few-shot: 28, 31, 35, 41-43, 51, 54  
final\_output: 148, 152  
Finance: 4  
financial: 6  
fine-tune: 106, 108-110, 117, 119-129  
fine-tuning: 32, 106-109, 116,  
120, 122-123, 125  
flow: 130-131, 134-151, 153-154  
Frequency Pane: 55-57  
function: 58, 69, 71-76, 109, 122

**G**

GANs: 3  
GenAI: 1-8, 27, 32, 58-59, 144, 149  
GPT-3: 4, 14, 18, 22, 28-29, 32-33, 36-41,  
44, 52-53, 55, 62, 66, 70-71, 73,  
107, 133-134, 138, 144, 147, 154  
GPT-4: 4, 9, 14, 18, 22-23, 28, 32-33, 36-44,  
47-50, 52-55, 62, 64, 66-67, 69-71, 73-74,  
82, 87, 96, 98, 133, 138, 144, 150-151  
GPT-4-32k: 43  
GPT-3.5 Turbo: 4, 14, 18, 23, 28, 32-33,  
36-37, 39, 43, 53, 62, 66, 70-71, 73, 107,  
120, 133-134, 138, 144, 147, 150, 154  
gpt-35-turbo: 15, 112-113  
GPT-4V: 150  
GPUs: 4  
GUI-based: 15, 58, 130, 133, 150

**H**

hallucination: 6, 28, 36-39, 42  
hyperparameters: 116, 120, 123

**I**

instructions: 4, 6, 13, 20-22, 28, 31-33,  
37-39, 47-49, 51, 54, 64, 67, 71,  
94, 98, 121, 124, 137, 152  
interact: 1, 3, 5, 8, 22, 28, 65, 77,  
94, 98, 102, 124, 127-128

**J**

JavaScript: 50, 143  
JSON: 24, 26, 31, 50-52, 65, 68, 70, 72,  
74-75, 104-105, 107, 151-152, 154  
Jupyter: 59, 65, 67, 72, 78, 98-99,  
104, 107, 120, 122, 128

**L**

language: 3, 6, 8, 18-19, 22-23, 35, 40, 48, 50,  
52-53, 59, 63, 82, 97, 100, 106, 143  
libraries: 65, 72, 78, 98, 100, 104, 107, 122, 128  
LLM-based: 8, 50, 130  
LLMs: 3, 5-6, 14-15, 21, 32, 35-36, 38-40, 45,  
48-49, 54, 68, 82, 106, 109, 130, 144

**M**

machine-learning: 142, 150  
machine-readable: 28, 50-51, 65, 68, 72  
management: 24, 60, 62, 83, 87, 92, 103,  
109-110, 118, 131, 133, 149  
marker: 20, 28, 47-51, 54, 49  
Marketplace: 83-84

message: 6, 22, 25-26, 28-33, 36, 38-39, 41-42, 44-51, 53-55, 64, 66-70, 73-75, 96, 102, 104-105, 107, 124-127, 129, 141, 147

## N

non-chat: 28, 52, 54-55, 149

## P

parameter: 3-4, 28, 49, 55, 67-68, 71, 73, 75, 104-105, 122, 129

playground: 2, 15, 19, 29, 94-96, 124-126

prompt: 3, 6, 8, 13, 15, 18-19, 21, 27-34, 36-51, 53-58, 63-65, 67-68, 71, 75, 109, 130-131, 134-154

Python: 19-20, 27, 30, 52-53, 58-67, 69-73, 75, 78, 98-100, 105, 107, 120, 122-123, 128-130, 136, 141, 143, 145-147, 149-150, 154

## Q

queries: 38, 47, 67, 82, 95, 124, 151-152, 154

query: 30, 82, 101, 126-127, 141

question: 6-8, 22, 26, 30, 32, 47, 64, 77, 94, 97-98, 104, 129, 140-141

quota: 14-16, 18-19, 142

## R

retrieval: 52, 54, 77, 82, 93-94

runtime: 136, 141, 145

## S

semantic: 53, 82, 93, 101

speech: 3-5, 8, 40, 130

syntactical: 28, 47-51, 54

system: 22, 25-26, 28-33, 36, 38-39, 41-42, 44-55, 64, 66-69, 73, 78, 107, 121, 124-125, 127, 129, 141, 147, 152

## T

text: 1-6, 8, 22-23, 40, 42, 48, 50, 52-53, 55-57, 63, 68-71, 78, 82, 87, 101, 108, 138, 147, 149-15

Text-based: 5

## V

validation: 107-108, 115, 120, 122-123, 126, 129, 147

vector: 3, 40, 77-78, 82-83, 87, 91-93, 98, 101, 104-105, 130, 150

vector-based: 82

VNets: 11

## W

Wikipedia: 134-136, 139-142

## Z

zero-shot: 27-28, 32-33, 35-36, 42-43



[www.packtpub.com](http://www.packtpub.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry-leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

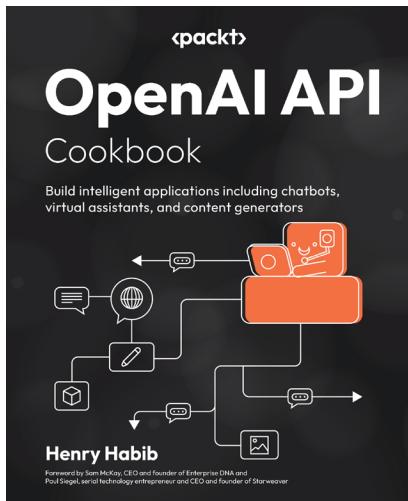
- Spend less time learning and more time coding with practical e-books and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free e-book or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers e-book versions of every book published with PDF and ePub files available? You can upgrade to the e-book version at [packtpub.com](http://packtpub.com) and as a print book customer, you are entitled to a discount on the e-book copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packtpub.com](http://www.packtpub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and e-books.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

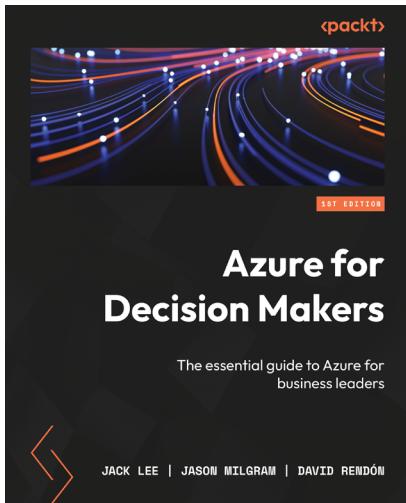


## OpenAI API Cookbook

Henry Habib

ISBN: 978-1-80512-135-0

- Grasp the fundamentals of the OpenAI API
- Understand the different elements, endpoints, and parameters of the OpenAI API
- Build tailored intelligent applications and workflows with the OpenAI API
- Explore advanced features, such as system messages, fine-tuning, and more
- Design and build applications that fully harness the power of ChatGPT



## Azure for Decision Makers

Jack Lee, Jason Milgram, David Rendón

ISBN: 978-1-83763-991-5

- Find out how to optimize business costs with Azure
- Strategize the migration of applications to the cloud with Azure
- Smoothen out the deployment and running of Azure infrastructure services
- Effectively define roles, responsibilities, and governance frameworks in DevOps
- Maximize the utility of Azure security fundamentals and best practices
- Adopt best practices to make the most of your Azure deployment

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.