CHAPTER 9

Utilizing the Azure OpenAI Service for Generative AI Applications

Generative AI isn't just a game changer; it's handing everyone a cheat sheet. Picture a small bakery using Azure OpenAI to whip up personalized wedding cake descriptions in seconds, or a rural hospital autogenerating patient discharge summaries that even overworked nurses can trust. That's the real magic of services like Azure OpenAI: they let you turn "someday" ideas into today's workflows without requiring you to have a PhD in machine learning. No more begging for GPU clusters or scraping training data—with these models in your toolbox, you'll be ready to prototype a customer service bot by lunch and refine it by dinner.

This chapter is your backstage pass to that revolution. We'll skip the fluff and dive into what really matters: why a well-crafted prompt outperforms generic instructions, and why passing the AI-102's generative AI section isn't about memorizing APIs—it's about proving you can turn an idea into reality quickly. You'll walk away with blueprints that work in boardrooms and code reviews alike.

Generative AI on Microsoft Azure

On Microsoft Azure, generative AI is powered mostly by Azure OpenAI Service, which connects OpenAI's foundational models to Azure so you can use them in your workloads. We'll explore how this interaction works in this section.

Types of Generative AI Models

Azure OpenAI Service offers a variety of generative AI models, each suited to different types of tasks. Table 9-1 provides an overview of the available models, broken

down by their capabilities (strengths and weaknesses) and use cases, to help you select the most appropriate one for your needs.

Table 9-1. Generative AI models available in Azure OpenAI Service

Model	Strengths	Limitations	Typical applications	Cost	Latency and throughput
GPT-4	Advanced reasoning, multimodal inputs, robust code support	Highest cost tier, lower throughput than smaller models	Sophisticated chatbots, detailed document generation, and code assistance	$30.00 per 1M prompt tokens and $60.00 per 1M completion tokens	Moderate latency (tens to hundreds of ms per 1k tokens) and a quota of 50k transactions per minute (TPM)
GPT-3.5	Cost-effective, high throughput, optimized for chat	Less nuanced reasoning, smaller context window	Customer support bots, text completion, and tutoring	$1.50 per 1M prompt tokens and $2.00 per 1M completion tokens	Lower latency (tens of ms per 1k tokens) and a quota of 200k TPM
Embeddings	Deep semantic understanding, low cost	No text generation, not suited for dialogue	Semantic search, recommendations, and clustering	$0.40 per 1M tokens (at $0.0004 per 1k)	Very fast (typically under 50 ms per call)
DALL-E	High-fidelity image generation from text	High cost per image, slower inference times	Marketing visuals, product mockups, and creative art	$0.02 per 1,024 × 1,024 image	Slower (around 1–3 s per image)

The list of typical applications for each model illustrates where each one shines. The Cost column indicates input and output rates for token-based models and per-image rates for DALL-E. Finally, you can use the latency and throughput figures to guide your expectations for performance under load.

OpenAI o-series models

OpenAI o3 is the latest and most advanced type of reasoning model available in the Azure OpenAI Service, at the time of writing. It offers significant improvements over previous versions in understanding and generating natural language and code. Since it's designed to handle more complex tasks, OpenAI o3 provides higher accuracy and relevance in its responses, making it suitable for a wide range of sophisticated applications.

The o3 models represent a newer generation of OpenAI models that are rapidly evolving and not all fully documented within Azure at the time of writing. As such, they're excluded from the table of generally available model types; however, because these models are significant, I cover them here to provide insight into their capabilities and potential availability.

Capabilities. OpenAI o3 is equipped with multimodal input capabilities that allow it to process both text and images. The fact that it's enhanced with natural language understanding means it can handle complex queries and generate detailed responses effectively. Additionally, OpenAI o3 has improved capabilities for generating and understanding code that make it a versatile tool for various programming tasks.

Use cases. Typical use cases for OpenAI o3 models include:

Advanced conversational agents
: OpenAI o3 models are ideal for applications that require sophisticated dialogue and nuanced understanding, because they enable more natural and effective interactions.

Content creation
: OpenAI o3 models can generate high-quality written content, like articles and reports, which is beneficial for marketing, journalism, and academic purposes.

Programming assistance
: OpenAI o3 models can assist developers with code suggestions and debugging to enhance productivity and streamline the development process.

GPT-4 models

GPT-4 models, including GPT-4o, provide robust performance for a variety of tasks involving natural language understanding and generation. These models are cost-effective and optimized for chat and completion tasks, so they're practical choices for many applications.

Capabilities. GPT-4 models excel at generating conversational responses and completing text prompts, thanks to their optimization for chat and completion tasks. They also feature a high token limit that allows them to process and generate large amounts of text. This makes them particularly useful for longer interactions and documents.

Use cases. Typical use cases for GPT-4 models include:

Customer support chatbots
: GPT-4 models automate customer service interactions to provide timely and accurate responses to user inquiries.

Text completion
: GPT-4 models assist with writing tasks by predicting text continuations, which is useful for drafting emails, reports, and other written materials.

Educational tools
: GPT-4 models enhance interactive learning experiences and tutoring by providing detailed explanations and personalized assistance to learners.

Embedding models

Embedding models transform text into numerical representations, known as *embeddings*, that capture semantic information. This transformation enables tasks that rely on understanding the underlying meaning of the text.

Capabilities. These models excel at *semantic understanding*, which means they can capture the meaning and context of words and phrases effectively. They also enhance data retrieval processes by finding contextually relevant information and thus making searches more accurate and efficient.

Use cases. Typical use cases for embedding models include:

Semantic search
: Embedding models improve search accuracy by understanding the intent behind queries, which leads to more relevant search results.

Recommendation systems
: Embedding models can suggest products or content based on user preferences and behavior, enhancing the user experience and engagement.

Knowledge management
: Embedding models facilitate the organization and retrieval of information from large datasets, making it easier to manage and utilize knowledge resources.

Clustering
: Embedding models can group similar items based on their semantic meaning, enabling tasks like topic discovery, content categorization, and user segmentation.

DALL-E models

DALL-E models generate images from textual descriptions, which means you can use them to create visuals based on specific prompts. These models are ideal for applications in design, marketing, and content creation, where unique and tailored visuals are required.

Capabilities. DALL-E models are proficient at producing high-quality images from detailed text descriptions. They support creative design processes by generating unique, diverse visuals and thus enabling innovative, customized image creation.

Use cases. Typical use cases for DALL-E models include:

Marketing materials
: DALL-E models can create custom images for advertisements and social media to enhance marketing campaigns with visually appealing content.

Product design
: DALL-E models can visualize product concepts and ideas to aid in the design process and provide clear representations of new products.

Creative projects
: DALL-E models can generate artwork and illustrations based on descriptive prompts to support artists and content creators in their creative endeavors.

Choosing a model

To help you sort through these models and pick the right one for your use case, see the decision tree in Figure 9-1, which will guide you from general task definition to the best model choice.

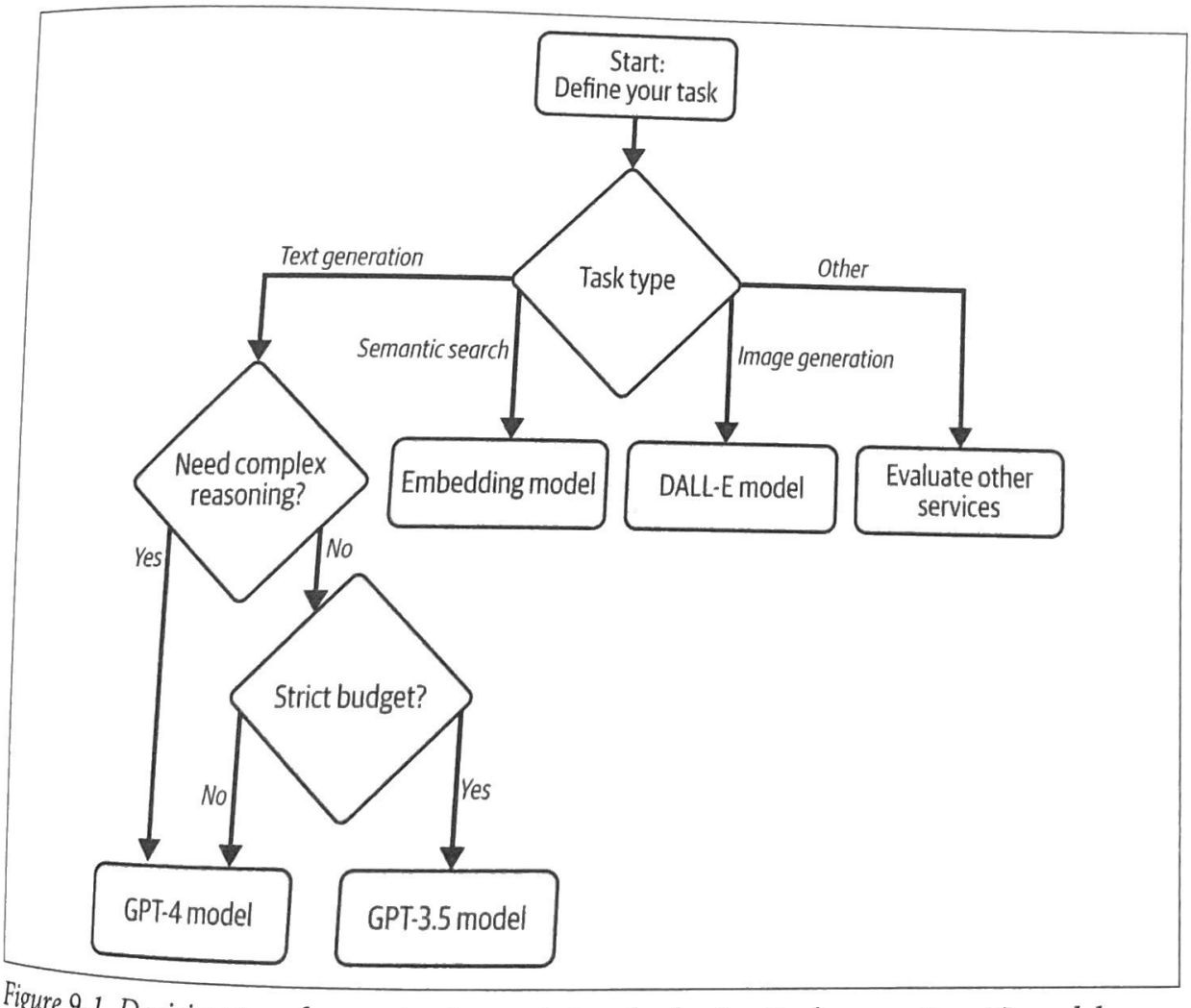

Figure 9-1. Decision tree for use in determining the best-suited generative AI model

Start by classifying your workload as text generation, semantic search, or image generation. If you need text generation, you should decide whether you require OpenAI o3's advanced reasoning or can opt for the more economical GPT-4o. For pure semantic tasks, pick the embedding model, and for image outputs, follow the branch to DALL-E. If your needs lie outside these categories, the decision tree points you toward exploring other parts of the Azure AI portfolio.

Limitations and ethical considerations

Despite their power, large language models have several important downsides:

Hallucinations
: LLMs can generate plausible but incorrect or fabricated information. This is an inherent limitation—research has found that it's impossible to completely eliminate hallucinations from LLMs.

Bias
: LLMs can reflect and sometimes amplify biases that are present in training data, leading to unfair or harmful outputs (e.g., generation of violent or biased imagery).

Cost and compute
: LLMs require significant GPU resources; even Turbo variants carry nontrivial inference costs and must be budgeted carefully.

Privacy and security
: Ingesting sensitive data into LLMs runs the risk of exposing that data to unauthorized users. Therefore, to comply with regulations like GDPR and HIPAA, you must implement careful data-handling and access controls.

Ethical use issues
: There's a constant risk that LLMs will be used to generate misleading content, create deepfakes, or propagate disinformation. Ways to mitigate this risk include implementing fact-checking pipelines, performing bias audits, issuing transparent user disclosures, and conducting human-in-the-loop reviews.

Before you choose a generative AI model, you should weigh these factors against its capabilities and costs. That will help you select and deploy the Azure OpenAI model that best aligns with your performance requirements, budget constraints, and ethical standards.

Building Responsible Generative AI Solutions

Generative AI has transformed a myriad of sectors, both technical and creative. However, many developers may not be aware of or pay much attention to the fact that creating generative AI solutions carries with it many important responsibilities. In this

Start by classifying your workload as text generation, semantic search, or image generation. If you need text generation, you should decide whether you require OpenAI o3's advanced reasoning or can opt for the more economical GPT-4o. For pure semantic tasks, pick the embedding model, and for image outputs, follow the branch to DALL-E. If your needs lie outside these categories, the decision tree points you toward exploring other parts of the Azure AI portfolio.

Limitations and ethical considerations

Despite their power, large language models have several important downsides:

Hallucinations
: LLMs can generate plausible but incorrect or fabricated information. This is an inherent limitation—research has found that it's impossible to completely eliminate hallucinations from LLMs.

Bias
: LLMs can reflect and sometimes amplify biases that are present in training data, leading to unfair or harmful outputs (e.g., generation of violent or biased imagery).

Cost and compute
: LLMs require significant GPU resources; even Turbo variants carry nontrivial inference costs and must be budgeted carefully.

Privacy and security
: Ingesting sensitive data into LLMs runs the risk of exposing that data to unauthorized users. Therefore, to comply with regulations like GDPR and HIPAA, you must implement careful data-handling and access controls.

Ethical use issues
: There's a constant risk that LLMs will be used to generate misleading content, create deepfakes, or propagate disinformation. Ways to mitigate this risk include implementing fact-checking pipelines, performing bias audits, issuing transparent user disclosures, and conducting human-in-the-loop reviews.

Before you choose a generative AI model, you should weigh these factors against its capabilities and costs. That will help you select and deploy the Azure OpenAI model that best aligns with your performance requirements, budget constraints, and ethical standards.

Building Responsible Generative AI Solutions

Generative AI has transformed a myriad of sectors, both technical and creative. However, many developers may not be aware of or pay much attention to the fact that creating generative AI solutions carries with it many important responsibilities. In this

section, we'll explore why you must develop, deploy, and operate generative AI solutions responsibly to make use of their benefits while minimizing the potential harms that can come from them.

Microsoft has designed its guidance on responsible generative AI to be both practical and actionable. It provides a structured approach to developing and implementing AI responsibly that's broken down into four key stages:

1. Identify potential harms that may arise from your planned solution.
2. Measure the extent to which these harms are present in the outputs generated by your solution.
3. Mitigate these harms at various layers within your solution to reduce their occurrence and impact, while also clearly communicating to users any potential risks.
4. Operate the solution responsibly by establishing and adhering to a deployment and operational readiness plan.

These stages align closely with the functions outlined in the National Institute of Standards and Technology (NIST) AI Risk Management Framework (*https://oreil.ly/xSBam*).

Identifying and mitigating potential harms

Generative AI solutions provide significant benefits but also carry potential risks, and you must identify and mitigate these risks to ensure that your AI systems are fair, safe, and trustworthy. This requires a structured approach to risk assessment, measurement, and ongoing management. Potential harms and key areas of risk you must address include:

Bias and discrimination

AI models can perpetuate and amplify existing biases (gender, racial, socioeconomic, etc.) that are present in training data, which can lead to discriminatory outcomes such as biased hiring algorithms, unfair loan approval processes, and misrepresentation in generated content.

Privacy violations

AI systems can inadvertently leak sensitive personal information, perhaps by generating text that includes personal details from training data, or infer private data (such as health conditions) from seemingly innocuous inputs.

Misinformation

AI-generated content can spread false or misleading information, either unintentionally due to errors or deliberately if manipulated. For example, AI systems can disseminate fake news, misleading health advice, and incorrect educational materials.

Psychological impacts

Interaction with AI systems can affect users' mental health and well-being. They can become overdependent on AI, have reduced human interaction, or be exposed to harmful content. For example, users can experience adverse psychological impacts when they become overly reliant on AI for decision making or when AI chatbots give them harmful advice.

Measuring potential harms

To effectively prioritize and address these risks, you must measure the potential harms by quantifying their likelihood and impact. Ways to do this include:

Conducting impact assessments

You should conduct thorough assessments to help you understand the potential impacts of AI systems. This involves systematically evaluating the severity and scope of identified risks. One good way of accomplishing this is by using structured frameworks such as data protection impact assessments (DPIAs) or ethical impact assessments (EIAs).

Testing and validation

Before you deploy AI models, you should regularly test and validate them to identify potential harms they may cause. This involves rigorously testing them under various scenarios to uncover their hidden biases and vulnerabilities. Some ways to do this include implementing adversarial testing and stress testing, as well as using bias detection tools to assess the models' performance and behavior.

Establishing feedback loops

After you deploy AI models, you should establish continuous feedback mechanisms through which you can monitor and evaluate the models' performance. Some ways to do this include collecting user feedback, conducting regular audits, and using monitoring tools to detect and address issues as they arise.

Mitigating potential harms

Mitigation involves taking proactive steps to reduce and eliminate identified risks. Here are some effective mitigation steps you can take:

Ensure algorithmic fairness

You must implement measures to ensure that AI algorithms do not produce biased outcomes. For example, while you're training a model, you can use fairness-aware algorithms, resample training data, and apply fairness constraints to detect and mitigate bias in both training data and model outputs.

Enhance transparency and explainability

You must enhance the transparency and explainability of your AI models to build trust with users and help them understand how the models make decisions. To

do this, you can use explainable AI techniques, provide clear and comprehensive documentation, and implement transparency reports that disclose the model's behavior and decision-making processes. These practices help demystify AI operations for users and stakeholders and make them more accessible.

Implement human oversight

By integrating human oversight into AI systems, you can ensure that models will be continuously monitored and that humans can intervene when necessary. Establishing human-in-the-loop processes, setting up review panels, and implementing escalation procedures for critical decisions are effective methods of maintaining control over AI-driven outcomes. Human oversight also acts as a safeguard against potential errors and unintended consequences of AI actions.

Establish proper data governance practices

You must implement robust data governance practices to maintain the integrity, quality, and security of the data you use in your AI systems. This includes enforcing data privacy regulations, utilizing secure data storage solutions, and establishing comprehensive data management policies to control data access and usage. By establishing effective data governance, you can ensure that your data will be handled responsibly and that your AI systems will operate on reliable and secure information.

Overall, by implementing algorithmic fairness, transparency and explainability, human oversight, and data governance, you and your organization can mitigate the potential harms associated with generative AI models and promote their ethical and responsible use.

Working with responsible generative AI solutions

To operate a responsible generative AI solution, you must continuously monitor your model's performance, be aware of your AI system's potential societal impacts, and be ready to adapt to new situations. Here are some practices you can adopt to stay on top of things and make sure your solution is behaving responsibly:

Monitoring and evaluation

You need to continuously monitor and evaluate your AI systems to make sure they're working effectively and to identify and proactively address potential issues. The Azure OpenAI Service provides robust monitoring tools that track the performance and behavior of generative AI models in real time, and by leveraging Azure's analytics and logging capabilities (such as Azure Monitor and Application Insights), you can assess your model's accuracy, response times, and usage patterns over time. Performing regular evaluations will help you detect anomalies, biases, and areas where the model may require adjustments to better align it with desired outcomes and ethical standards.

Updates and improvements

Generative AI models require regular updates and improvements to remain effective and relevant. The Azure OpenAI Service supports this by providing seamless access to new model versions and enhancements. Organizations can take advantage of Azure's managed services to deploy updates without significant downtime, ensuring that their AI systems benefit from the latest advancements in technology and research. Continuous improvement practices involve retraining models with new data, fine-tuning parameters, and incorporating user feedback to enhance the performance and reliability of AI solutions. Azure's DevOps tools support automated deployment pipelines, enabling rapid iteration and deployment of improved models while maintaining high standards of quality and consistency.

Stakeholder communication

You must effectively communicate with stakeholders as part of the process of responsibly deploying generative AI solutions. Azure OpenAI can support you in this by providing comprehensive documentation and insights into model functionalities and limitations. You should engage with stakeholders, including employees, customers, and partners, to inform them of how your organization is using AI systems and what measures it has in place to ensure their ethical usage. By implementing clear communication, you can foster trust and enable stakeholders to provide valuable feedback that can contribute to the ongoing refinement of AI solutions.

Compliance and reporting

Adhering to regulatory requirements and maintaining compliance is another key aspect of responsible AI operations. Azure OpenAI helps organizations meet compliance standards by providing tools and features that support data privacy, security, and ethical guidelines. Its automated compliance reporting capabilities allow organizations to generate detailed reports on AI system usage, data handling practices, and alignment with industry regulations. In addition, Azure's built-in compliance certifications—such as GDPR, HIPAA, and ISO—offer a robust framework for ensuring that AI deployments meet legal and ethical requirements.

Prompt Engineering with the Azure OpenAI Service

To unlock the full potential of the generative AI models that are provided by Azure OpenAI, you must use effective prompt engineering techniques. By carefully designing and refining inputs, you can significantly enhance the quality and relevance of AI-generated outputs.

Writing Effective Prompts

Prompt engineering is an emerging discipline that is focused on crafting and optimizing prompts to better work with LLMs across diverse sets of applications and use cases. It involves designing, testing, and refining prompts to improve the accuracy and relevance of the models' responses, combining analytical skills with creativity to make use of their full potential.

Prompt engineering plays a big role in helping models understand and execute tasks more efficiently, by ensuring clear, concise, and well-structured inputs are provided. The primary goal is to create prompts that are precise, context rich, and structured in a way that the model can easily interpret and respond to accurately. Here are some key practices you can follow to ensure that you clearly communicate your intentions to the model you're using and increase the likelihood of getting appropriate outputs:

Structure prompts clearly and consistently
: Use punctuation, headings, and separators (like `---` or `"""`) to define distinct sections of the prompt, such as instructions, context, and queries. Creating reusable prompt templates with this consistent structure helps the model interpret input more reliably and respond with greater coherence and accuracy across different use cases.

Break down tasks
: Break down large tasks into smaller, manageable steps. This approach, known as *chain-of-thought prompting*, enables the model to handle complex queries more effectively by processing each step sequentially.

Include contextual information
: Be specific about the outcome, format, and style of the response that you want. For instance, if you need a summary, specify how long it should be and what key points it should cover.

Use examples for few-shot learning
: By including one or more examples of the desired behavior (a practice known as *few-shot learning*), you can condition the model to respond appropriately. For instance, by providing example Q&A pairs, you can help the model understand the format and content of the expected output.

Experiment with different arrangements
: The order of instructions, context, and examples can impact the model's performance. Experiment with different prompt structures to find the most effective arrangement for your specific task.

Provide clear instructions and use cues

Include explicit instructions and use cues to guide the model toward the desired output. For example, you can specify the format or style of the response, such as by telling the model to use bullet points for a summary.

Manage token limits

Be mindful of *token limits*, which represent the combined length of input and output. Optimize your prompts to fit within these limits without losing essential information. (For instance, the GPT-3.5 Turbo Instruct model for Azure OpenAI has a limit of 4,097 tokens.)

Iterate and refine

Prompt engineering is an iterative process, so you should continuously test and refine your prompts to improve the model's performance. Each iteration will provide insights into how the model interprets the prompts and what adjustments you can make to produce better outcomes.

Advanced Techniques and Best Practices

Here are some specific techniques that you can use to design prompts that achieve better results:

- Provide a clear instruction. For instance, in a summarization task, you can use a prompt like this:

```
Summarize the following text in one sentence
The quarterly report shows a 20 percent increase in sales across all regions
```

  If the summary is too brief or misses key details, you can add guidance on length or focus areas.

- Include a few input/output pairs to demonstrate the desired behavior. For example, in a sentiment classification use case, you can classify sentiment for each review as follows:

```
Review: This product exceeded my expectations
Sentiment: Positive
Review: The service was slow and unhelpful
Sentiment: Negative
Review: The interface is user-friendly, but features are limited
Sentiment: Neutral
```

  The model uses these examples to complete the final classification.

- Prepare a JSON Lines (JSONL) file containing prompt-completion pairs and then use the fine-tuning API to train a custom model. For example:

```
{
  "prompt":"Translate to German English How are you What is the German",
  "completion":"Wie geht es Ihnen"
}
```

```
{
  "prompt":"Translate to German English Good morning What is the German",
  "completion":"Guten Morgen"
}
```

Fine-tuned models deliver more consistent outputs for domain-specific tasks.

- Direct the model to think step by step by adding instructions at the start of the prompt. Here's an example:

  ```
  Explain your reasoning step by step to solve the following math problem
  Problem: What is 12 times 15
  ```

  This yields more transparent reasoning and can improve correctness on complex queries.

- Include instructions in your prompt that trigger retrieval or API calls to ground responses in up-to-date information. For example:

  ```
  Fetch current weather data for Seattle then summarize temperature and
  conditions
  ```

  Combining external data with model generation helps improve factual accuracy.

- Begin prompts with a system message that sets ethical constraints and refusal rules as part of your application logic. Here's an example:

  ```
  You are an assistant that checks user input for bias and refuses to generate
  harmful or hateful language
  ```

- Monitor outputs continuously, and refine your prompts if biased or unsafe content appears.

By following these best practices, you can effectively harness the power of Azure OpenAI to achieve high-quality, relevant, and accurate outputs from your NLP tasks.

Generating Content with the Azure OpenAI Service

Azure OpenAI empowers you and your organization to revolutionize your content creation processes by leveraging advanced generative AI models. Whether you're crafting compelling marketing materials, drafting comprehensive reports, or generating engaging social media posts, Azure OpenAI provides you with the tools and flexibility you need to produce high-quality, tailored content efficiently. In this section, we'll explore how you can use this service to generate different types of content.

Using the Azure OpenAI Service

The first step is to provision an Azure OpenAI Service resource within your Azure subscription. Because access is currently limited, you'll need to apply at *https://aka.ms/oai/access*. Once approved, you'll also gain access to Azure AI Foundry, a platform where you can experiment with different models, explore their capabilities,

manage and deploy them, and customize them for your specific use cases. These services will become available through the Azure portal after you create your resource.

Before you can make API calls, you'll need to deploy a model to use with your prompts. When creating a new deployment, specify the base model you want to deploy. You can do this through Azure AI Foundry, the Azure CLI, or the REST API.

Once it's deployed, you can begin testing how the model completes your prompts. There are several factors that can affect this, including how the prompt was engineered, the parameters of the model, and the data that was used to train the model.

Azure AI Foundry provides interactive playgrounds where you can test model behavior. Two playgrounds are available: one for completions and one for chat.

The completions playground

There are several parameters that you can configure in the completions playground. This section summarizes the ones you'll use most often.

Temperature. This parameter controls the randomness of the output generated by the model. Lower values (close to 0) make the output more deterministic and focused, favoring high-probability words. This is useful for tasks requiring precision and consistency. Higher values (closer to 1) introduce more variability and creativity into the output. This allows the model to explore less likely word choices, which can be beneficial or creative writing and brainstorming tasks.

If you set the `temperature` parameter to `0`, the model will always pick the highest-probability next token. For example, the prompt "Write a short story about a robot exploring Mars" will produce this deterministic response:

> The robot landed on Mars, collected rock samples, and transmitted data back to Earth.

On the other hand, if you set the `temperature` parameter to `0.9`, the model will explore creative possibilities. In this case, you might get a response like:

The curious automaton danced beneath a crimson sky its sensors humming a melody as it uncovered ancient Martian inscriptions.

Max length (in tokens). This parameter sets an upper limit on the number of tokens (including words and punctuation) that the model can generate in a single response. It ensures that the output does not exceed a specified length, so can use it to make responses as short or long as you want them to be. For example, you can set a lower `max_length` value to produce concise answers or a higher limit to allow for more detailed explanations.

Stop sequences. These are specific strings of characters that signal the model to stop generating text once they are encountered. By using stop sequences, you can help

your model produce outputs that are well-defined and terminate appropriately, avoiding extraneous or irrelevant content beyond the desired endpoint.

Top P (aka nucleus sampling). This parameter determines the cumulative probability threshold for token selection. When you apply this parameter, your model will consider only the smallest set of tokens whose cumulative probability meets the specified `top_p` value. This allows you to strike a balance between maintaining coherence and introducing diversity, with lower values producing more predictable outputs and higher values allowing for more creative variations.

A `top_p` value of `0.3`, for instance, restricts the output to the top 30% of likely tokens, producing more precise technical narratives. For example, for the "Write a short story about a robot exploring Mars" prompt, the model might produce this output:

> The robot landed precisely, analyzed soil samples, and then returned findings to mission control.

Raising the `top_p` parameter to `0.9` will broaden the token pool and allow for more nuanced responses, like this one:

> Under swirling red dust, the robot pondered its solitude, forging onward toward uncharted canyons lit by distant suns.

Frequency penalty. This parameter discourages the model from repeating tokens that have already appeared in the text. Higher `frequency_penalty` values encourage more diverse word choices and help the model avoid redundancy, thus making the text the model produces more engaging and varied.

Presence penalty. This parameter influences the likelihood that the model will introduce new topics or concepts in its output. By setting the `presence_penalty` value higher, you can discourage the model from repeating previously mentioned topics and encourage it to explore a wider variety of ideas. This is particularly useful in tasks requiring innovative or varied content generation.

Pre-response text. This is a string that is added to the prompt before the model generates a completion. It sets the stage by providing necessary context or instructions, thus ensuring that the model's output aligns with the specific requirements of the task at hand.

Post-response text. This is appended to the generated output and is often used to format the final response or add instructions after the main content. You can use it to ensure that the response ends in a specific way or includes follow-up prompts for further interaction.

These parameters allow users to fine-tune and control the behavior of their AI models in the Azure OpenAI completions playground. They help optimize the outputs of various applications and ensure that they meet the specific requirements of different tasks.

The chat playground

The parameters users can configure in the chat playground include:

Max response
: This parameter controls the maximum number of tokens the model may generate. You can follow the same guidance as for `max_length` in the completions playground, setting upper limits to balance brevity and detail, depending on the task.

Top P
: This parameter limits token choices to those in the Top P cumulative probability mass. As in the completions playground, adjusting this value lets you control the trade-off between creativity and determinism.

Past messages included
: This parameter determines how many previous interactions the model should consider when generating a response. In a conversational AI setting, maintaining context is essential for producing coherent replies. Including more past messages can enhance the AI system's understanding of the conversation's flow and enable it to provide more contextually appropriate and relevant responses, but including too many may lead to longer processing times and potentially dilute the focus of the answers. On the other hand, including too few past messages might result in responses that lack context or continuity and can make the conversation feel disjointed.

With that background in mind, let's look at how you can start using Azure OpenAI for practical applications.

Using Azure OpenAI in Your Applications

To integrate Azure OpenAI into your own application, you need to complete a few key steps. The first one is setting up your environment to securely connect with your Azure OpenAI Service resource. You'll need to retrieve your API key and endpoint, then set them as environment variables on the local machine or store them in Azure Key Vault so that secure API calls can be made.

Next, define a main function that makes a request to your Azure OpenAI Service resource. This function should include parameters such as `prompt`, `max_tokens`, and `temperature`. You can use the client object to call the `getCompletions` method with these parameters. To handle the response, iterate over the choices provided by the API and print the results.

If you're using the REST API, you can make HTTP requests directly to the OpenAI endpoint, including the necessary headers and request body. You'll also need to specify the model type, the prompt, and any other relevant parameters.

Generating Text

The main purpose of generative AI models is generating text. There are different types of prompts that you can try for this purpose—here are two examples:

Asking about facts

```
What is the tallest mountain in the world?
```

Summarizing content

```
Can you summarize the following content? Add content here.
```

To evaluate and improve the quality of generated text, you need to measure its relevance by computing the semantic similarity between prompts and outputs. You can do this by using embedding-based cosine similarity or a BERTScore. To verify factual accuracy, cross-check responses against trusted data sources or use contextual consistency metrics such as Groundedness Pro. You should also assess the prompts' appropriateness, through human evaluation or by using automated classifiers for tone and bias. Additionally, automated metrics such as perplexity for fluency, Recall-Oriented Understudy for Gisting Evaluation (ROUGE), or BLEU for summarization, can provide quantitative feedback on model performance. Based on these insights, you can refine your model by tuning parameters such as temperature and Top P and by providing few-shot examples to guide it toward more accurate and relevant outputs.

Practical: Using Azure OpenAI API GPT-4 to generate text

Suppose you're a content writer for a company that promotes renewable energy solutions. Your task is to create informative and engaging content about the benefits of using renewable energy. To assist you in this task, you'll use Azure's OpenAI GPT-4o model to generate text that you can incorporate into your articles.

Step 1: Set up an Azure OpenAI Service resource. First, you'll need to create a language resource, as follows:

A. Log in to the Azure portal with your Azure account credentials.
B. Click "Create a resource" in the lefthand menu.
C. In the search bar, type "Azure OpenAI" and press Enter.
D. Select Azure OpenAI from the search results.
E. On the Azure OpenAI Service page, click Create.

F. Fill in the required details:
   i. Subscription: select your subscription.
   ii. Resource group: create a new resource named `OpenAIResourceGroup`, or use your default user group.
   iii. Region: choose a region that's close to you (e.g., East US).
   iv. Name: enter an appropriate OpenAI Service resource name, such as `MyOpenAIResource`.
   v. Pricing tier: select the appropriate tier based on your needs.
G. Click "Review + Create" and, after validation, Create.
H. Wait for the deployment to complete, then click "Go to resource."
I. Note the endpoint and API keys:
   i. Select "Keys and Endpoint" under Resource Management in the left menu.
   ii. Copy the endpoint URL (e.g., *https://myopenairesource.openai.azure.com*) and one of the keys.
J. Click "Explore Azure AI Foundry portal" at the bottom of the resource's Overview page.
K. Click Models + endpoints in the left pane.
L. Click "Deploy model" and choose "Deploy base model" at the top of the page.
M. Configure the deployment:
   i. Model: select "gpt-4" from the drop-down menu and click Confirm.
   ii. Deployment type: select Global Batch.
   iii. Model version: select the latest version that's available.
N. Click "Create resource and deploy." The deployment may take several minutes. Once it's completed, the model's status will show as Deployed.

Step 2: Set up your development environment. Next, you'll need to install the required packages:

A. Open your command prompt or terminal, navigate to your preferred directory, and run the following commands to create a project directory and change into that directory:

```
mkdir azure-openai-gpt4
cd azure-openai-gpt4
```

B. Create a virtual environment by running the following command:

```
python -m venv venv
```

C. Activate the virtual environment by running:

```
venv\Scripts\activate
```

D. Install the OpenAI Python client by running:

```
pip install openai
```

E. Install the `python-dotenv` package with:

```
pip install python-dotenv
```

F. In your project directory, create a file named *.env*.

G. Open *.env* in a text editor.

H. Add the following lines of code to the file:

```
AZURE_OPENAI_API_KEY=your_openai_api_key
AZURE_OPENAI_ENDPOINT=your_openai_endpoint
AZURE_OPENAI_DEPLOYMENT_NAME=gpt-4-model
```

Replace *your_openai_api_key* with your actual API key and *your_openai_endpoint* with your actual endpoint URL.

Ensure there are no spaces around the equals signs.

Step 3: Implement the GPT-4 text generation script. Now, you create the Python script:

A. In your project directory, create a file named *generate_text.py*.

B. Add the following code to it:

```
import os
from dotenv import load_dotenv
from openai import AzureOpenAI

load_dotenv()

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

response = client.chat.completions.create(
    model=os.getenv("AZURE_OPENAI_DEPLOYMENT_NAME"),
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {
```

```
            "role": "user",
            "content": "What are the benefits of using renewable energy?"
        },
    ]
)

print(response.choices[0].message.content)
```

First, you import the necessary libraries and call `load_dotenv` to load environment variables from the *.env* file. Then you configure the OpenAI API with your credentials. Next, you define what you want the model to generate text about—that is, you're setting the prompt. After that, you make the API request, specifying the GPT-4 model. Finally, you display the generated text.

Step 4: Run the script. Now, you can execute the script that you have created to show the text that you have generated as part of your model:

A. Ensure the virtual environment is activated (see step 2). Your command prompt should show `(venv)` at the beginning.

B. In the terminal, run this command to execute the script:

```
python generate_text.py
```

C. View the output.

Recall that in the script, we set the prompt to "What are the benefits of using renewable energy?" The response might look something like this:

> Renewable energy sources such as solar, wind, and hydroelectric power offer numerous benefits. They reduce greenhouse gas emissions, which helps to decrease environmental pollution and combat climate change. This energy is sustainable and inexhaustible, and it ensures a stable energy supply for the future. It also reduces our country's dependence on fossil fuels and thus enhances our energy security. Additionally, investing in renewable energy can create jobs, stimulate economic growth, and lead to technological advancements.

And with that, you've successfully used Azure OpenAI GPT-4 to generate text! Feel free to modify the prompt and explore as you like.

Best practices

There are several best practices that you can follow to make sure your solution is deployed securely and the results generated by the OpenAI Service are effective:

- Securely manage credentials by storing API keys in a *.env* file and excluding it from version control.
- Regularly review documentation and test new API versions in a development environment.
- Implement error handling and logging to manage and troubleshoot API issues.
- Optimize API calls and monitor usage to control costs.
- Use version control, write clear, modular code, and include unit tests.
- Craft concise, context-rich prompts for high-quality outputs.
- Isolate dependencies with virtual environments and keep them updated.

Generating Images

To enable image generation, OpenAI provides DALL-E, a model that takes natural language input and generates original images based on it. You can explore its capabilities in the DALL-E playground, where you can test different prompts and view the images they produce. The playground allows you to configure settings such as the number of images to generate and the desired resolution.

To use DALL-E models through your Azure OpenAI Service endpoint, you'll need to make a REST API call. Start by setting up a POST request directed at your Azure OpenAI endpoint. You'll need to include the specific endpoint URL and an authorization key. The body of the request should contain the following parameters:

`prompt`
: This is a textual description that DALL-E uses to generate an image. You should make this description as detailed and specific as possible to guide the model effectively. It can include information about objects, scenes, actions, styles, and any other elements you want to appear in the generated image.

`n`
: This specifies the number of images you want DALL-E to generate for the given prompt. It allows you to receive multiple variations based on the same description.

`size`
: This defines the dimensions of the generated image, typically formatted as width × height. Common sizes include `1024×1024`, `1792×1024` (landscape), and `1024×1792` (portrait).

Here's an example:

```
{
    "prompt": "A squirrel in a bow tie",
    "n": 1,
    "size": "512x512"
}
```

The response will include an *operation-location header*, which contains a URL for a callback service. Your application can use this to poll and track the progress of the image generation task.

Once the process concludes, the service will return a JSON object containing details about the result. This will include the unique ID of the operation, timestamps that indicate the creation and expiration times, and a URL that points to the generated image. Here's an example of such a response:

```
{
    "created": 1679320850,
    "expires": 1679407255,
    "id": "unique_operation_id_placeholder",
    "result": {
      "created": 1679320850,
      "data": [
            {
                "url": "https://yourgeneratedimageurl.png"
            }
        ]
    },
    "status": "succeeded"
}
```

Next, you'll put this into practice in an exercise on generating images with the Azure OpenAI DALL-E API. Note that this exercise assumes you've completed the previous one.

Practical: Generating images with the Azure OpenAI DALL-E API

Suppose you're a graphic designer for a travel magazine. You need to create unique images to accompany some of the articles, so instead of using stock photos, you'll generate custom images.

Step 1: Deploy the DALL-E model. Start off by deploying the Dall-E model:

A. Go to your Azure OpenAI Service resource in the Azure portal.

B. Create a new deployment:

   i. Click "Model deployments" under Resource Management in the left menu.

   ii. Click "Create."

C. Configure the deployment:
   i. Model: select "dall-e."
   ii. Deployment name: enter **`dalle-model`**.
D. Click Create.
E. Wait for deployment to complete. At this point, the model's status will change to Deployed.

Step 2: Update environment variables. You can then put the DALL-E deployment name into your list of environment variables:

A. Open the *.env* file in a text editor and add the DALL-E deployment name:

```
AZURE_OPENAI_DALLE_DEPLOYMENT_NAME=dalle-model
```

B. Save the file and exit the editor.

Step 3: Implement a DALL-E image generation script. Now, you can create the Python script:

A. In your project directory, create a file named *generate_image.py*.
B. Add the following code to it:

```
from openai import AzureOpenAI
import os
import requests
from dotenv import load_dotenv
from PIL import Image
import json

load_dotenv()

azure_client = AzureOpenAI(
    api_version="2024-02-01",
    api_key=os.environ["AZURE_OPENAI_API_KEY"],
    azure_endpoint=os.environ['AZURE_OPENAI_ENDPOINT']
)

dalle_deployment_name = os.environ["AZURE_OPENAI_DALLE_DEPLOYMENT_NAME"]
response = azure_client.images.generate(
    deployment_name=dalle_deployment_name,
    prompt="A multicolored umbrella on the beach, disposable camera",
    size="1024x1024",
    n=1
)

parsed_response = json.loads(response.model_dump_json())

output_dir = os.path.join(os.curdir, 'output_images')
```


```
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

output_file = os.path.join(output_dir, 'output_image.png')
image_url = parsed_response["data"][0]["url"]
image_data = requests.get(image_url).content

with open(output_file, "wb") as file:
    file.write(image_data)

Image.open(output_file).show()
```

The script starts by importing the necessary modules and libraries, including `os` for handling environment variables, *dotenv* for loading variables from a *.env* file, and *openai* for interacting with the OpenAI API. It then uses the `load_dotenv` function to load environment variables from the *.env* file, enabling secure access to sensitive information like API keys. After that, the script creates an `AzureOpenAI` client, passing the API version, API key, and endpoint, all of which are retrieved from environment variables. It also obtains the deployment name for the DALL-E model from the *.env* file.

Next, the code defines a prompt that describes the image to be generated—in this case, "A multicolored umbrella on the beach, disposable camera." The script then makes an API request to the DALL-E model using the `images.generate` method, providing the prompt, the number of images to generate (`n=1`), the size of the image (`1024x1024`), and the deployment name for the model. Finally, it prints the URL of the generated image so the user can view or download the output directly.

Step 4: Run the script. After you implement the script, you can execute it to view the generated image:

A. Ensure the virtual environment is activated. Your command prompt should show `(venv)` at the beginning.

B. In the terminal, run:

```
python generate_image.py
```

C. View the output. The script will display a URL.

D. Copy the URL and paste it into a web browser to view the generated image. Figure 9-2 depicts a sample image that might be generated based on the prompt.

Figure 9-2. Sample image of umbrella and disposable camera generated by prompt

Best practices

There are a few best practices you can follow to ensure the model produces the best images for your purposes, in a secure manner:

Use clear, specific prompts
: Ensure that your prompts are clear and specific to get the best results from the model.

Test and iterate
: Experiment with different prompts and parameters to refine the outputs.

Secure your API keys
: Use secure methods like Azure Key Vault to store and access your API keys in production environments.

By following the steps outlined in this section, you've successfully set up and used the Azure OpenAI API with the DALL-E model to generate images for your application.

Generating Code

Another important use of generative AI is for generating code and debugging code issues. In this section, you'll get some hands-on experience with this.

Fixing a bug in a piece of code

You can fix bugs in your code by entering a prompt like the following, along with the buggy code:

```
Fix the bug in this Python function:
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n)
```

The output will be:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Writing unit tests

You can also ask generative AI to write unit tests for code by entering this:

```
Write a unit test for this code:
CODE_FOR_FACTORIAL_HERE
```

Substitute the factorial code in the previous subsection for *CODE_FOR_FACTORIAL_HERE*.

Here's a sample output:

```
import unittest

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

class TestFactorialFunction(unittest.TestCase):

    def test_factorial_of_zero(self):
        self.assertEqual(factorial(0), 1)

    def test_factorial_of_one(self):
        self.assertEqual(factorial(1), 1)

    def test_factorial_of_positive_int(self):
        self.assertEqual(factorial(5), 120)

    def test_factorial_of_large_number(self):
        self.assertTrue(factorial(10) == 3628800)
```

```
    def test_factorial_with_negative_input(self):
        with self.assertRaises(RecursionError):
            factorial(-1)

if __name__ == '__main__':
    unittest.main()
```

You can see that the model generated a set of tests you can run to see if your code covers everything it's supposed to.

Building complete functions and classes

You can also prompt the model to create entire modules. For example, if you enter the prompt: "Write a Python Flask class with CRUD endpoints for a Book resource, including JWT authentication and error handling," the model will generate a `BookAPI` class with routes for creating, reading, updating, and deleting books, as well as handling token verification and errors.

Automated code documentation

To generate docstrings or comments, use a prompt such as "Add Google-style docstrings to each function in the following code." The model will produce detailed descriptions for parameters, return values, and examples.

By going beyond simple bug fixes and tests to include complete class generation, documentation, and refactoring, you can leverage Azure OpenAI to accelerate every stage of software development.

Practical: Creating a chatbot

This exercise will guide you through setting up a simple GPT-3.5 chatbot using the LangChain framework and your Azure OpenAI Service resource. It will also cover environment setup, code configuration, and deployment.

Step 1: Deploy the GPT-3.5 Turbo model. To start off, you deploy the GPT-3.5 Turbo model:

A. Go to your Azure OpenAI Service resource in the Azure AI Foundry within the Azure Portal.

B. Create a new deployment:

   i. Click Deployments in the left pane.

   ii. Click "Deploy model."

C. Configure the deployment:

   i. Model: select "gpt-35-turbo."

   ii. Deployment name: enter **`gpt-35-turbo`**.

D. Click Create to create the deployment.

E. Wait for the deployment to complete. Once it does, the model's status will change to Deployed.

Step 2: Install the required Python packages. You'll need to set up some Python packages before you can run the chatbot:

A. Activate the virtual environment.

B. Install the LangChain package by running the following command:

```
pip install langchain
```

C. If the `python-dotenv` package is not already installed, run this command to install it:

```
pip install python-dotenv
```

Step 3: Update the environment variables. To continue setting up the chatbot, you'll need to set up the environment variables that will be required.

A. Open the *.env* file and add the following line:

```
AZURE_OPENAI_GPT35_DEPLOYMENT_NAME=gpt-35-turbo
```

B. Save the file.

Step 4: Implement the chatbot script. Now, you can create the Python script:

A. In your project directory, create a file named *chatbot.py*.

B. Open it in your text editor and add the following code:

```
import os
from dotenv import load_dotenv
from langchain_openai.chat_models import AzureChatOpenAI
from langchain.schema import HumanMessage

load_dotenv()
azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
azure_api_key = os.getenv("AZURE_OPENAI_API_KEY")
deployment_name = os.getenv("AZURE_OPENAI_GPT35_DEPLOYMENT_NAME")
api_version = os.getenv("AZURE_OPENAI_API_VERSION")

model = AzureChatOpenAI(
    azure_endpoint=azure_endpoint,
    azure_deployment=deployment_name,
    api_version=api_version,
    openai_api_key=azure_api_key
)
```

```
    def chat():
        print("Welcome to the Customer Service Chatbot!")
        print("Type 'exit' or 'quit' to end the chat.\n")
        while True:
            user_input = input("You: ")
            if user_input.lower() in ["exit", "quit"]:
                print("Chatbot: Goodbye!")
                break

            message = HumanMessage(content=user_input)
            response = model([message])
            print(f"Chatbot: {response.content}\n")

if __name__ == "__main__":
    chat()
```

The script starts by importing the necessary modules and libraries, including `os` for environment variable management, *dotenv* for loading variables from a *.env* file, and `langchain` to interact with the GPT-3.5 Turbo model. It then uses the `load_dotenv` function to load environment variables from the *.env* file, enabling secure access to sensitive information such as the Azure OpenAI endpoint, API key, and deployment name.

Next, the script initializes the GPT-3.5 Turbo model using the `AzureChatOpenAI` class, specifying the endpoint, API version, deployment name, and API key. It then defines the `chat` function to handle the conversational loop with the user. It displays a welcome message and instructions for exiting the chat. Inside the loop, the user's input is collected and passed as a `HumanMessage` object to the model, which processes the input and generates a response. The chatbot then prints the response back to the user. The script concludes with a main block that starts the chat session when it's executed.

Step 5: Run the chatbot. You can now execute the chatbot and see your implementation in action:

A. Ensure that the virtual environment is activated. Your command prompt should show `(venv)` at the beginning.

B. Run the script by entering this command in the terminal:

```
python chatbot.py
```

C. Interact with the chatbot. Here's an example of a conversation you could have:

   i. *Chatbot:* Welcome to the Customer Service Chatbot!

   ii. *You:* What are your operating hours?

   iii. *Chatbot:* Our operating hours are from 9 a.m. to 6 p.m., Monday through Friday.

   iv. *You:* Thank you!

v. *Chatbot*: You're welcome! If you have any more questions, feel free to ask.

vi. *You*: exit

vii. *Chatbot*: Goodbye!

Best practices

There are several best practices you can follow to ensure that the chatbot will work properly and be effective at providing the appropriate responses. Here are a few tips:

- Implement error handling and logging to maintain a reliable chatbot experience.
- Keep your code modular and use version control for easier maintenance.
- Regularly review documentation and update libraries to leverage new features.
- Monitor API usage to optimize performance and control costs.

And with that, you've successfully created a chatbot using Azure OpenAI and LangChain. Next, we'll look at how you can use fine-tuning and optimization to further enhance the model based on your own data.

Fine-Tuning and Optimizing Generative AI Models

Fine-tuning allows you to customize pretrained models to perform specific tasks with greater accuracy and relevance. By fine-tuning a general-purpose model, you can adapt it to understand the nuances and specifics of your particular dataset or application, enabling it to perform better on tasks that are specific to your use case. Training the model on domain-specific data enhances its ability to generate relevant responses, improves its accuracy in prediction tasks, and reduces bias. Additionally, fine-tuning is often more resource-efficient than training a model from scratch, as it leverages the existing capabilities of a pretrained model and requires less data and computational power to achieve good performance.

Real-world applications are diverse and span multiple industries. In healthcare, fine-tuned models can assist in diagnosing diseases, and you can train them on specific medical datasets to make them better able to assist in personalized medicine. In customer service, to improve customer satisfaction, companies fine-tune models on their own datasets of interactions to create chatbots that provide more accurate and contextually appropriate responses. In the legal and financial sectors, firms fine-tune models to comprehend and analyze domain-specific documents so they can aid in contract analysis and fraud detection. As these applications illustrate, fine-tuning brings adaptability and enhanced performance to AI models, making them indispensable tools in specialized tasks across various fields.

Fine-Tuning Your OpenAI Model

You can use your own information sources when fine-tuning your model, enabling your AI chat models to reference that data when generating responses. This helps ensure the responses are more relevant to your use cases. This integration relies on Azure AI Search's ability to inject data segments into the dialogue prompts, which improves the model's context as well as the accuracy of its answers.

To use the Azure OpenAI Service with custom data, follow these steps:

1. Prepare your training data in JSONL format with at least 100 prompt-completion pairs. For example, the first few lines of your *training_data.jsonl* file might look like this:

```
{
  "prompt":"Translate to French: How are you",
  "completion":"Comment allez-vous ?"
}
{
  "prompt": "Translate to French: Good morning",
  "completion": "Bonjour"
}
```

   Each line must be valid JSON, with quoted keys and no trailing commas.

2. Upload the JSONL file via the OpenAI Python SDK:

```
import openai
openai.api_type="azure"
openai.api_base="https://your-resource.openai.azure.com"
openai.api_version="2024-02-01"
openai.api_key="YOUR_API_KEY"
file_resp = openai.File.create(
    file=open("training_data.jsonl"),
    purpose="fine-tune"
)
```

3. Start a fine-tuning job by specifying the training file, base model, epoch count, and learning rate:

```
ft_job = openai.FineTune.create(
  training_file=file_resp.id,
  model="gpt-3.5-turbo",
  n_epochs=4,
  learning_rate_multiplier=0.1
)
```

4. Monitor the job status until it completes, then retrieve `fine_tuned_model` from the response.

5. Deploy your custom model by creating a deployment resource in Azure AI Foundry or using the Azure OpenAI REST API.

6. Call the fine-tuned model in chat completions by using its deployment name:

```
resp = openai.ChatCompletion.create(
  engine="my-fine-tuned-deployment",
  messages=[{"role":"user","content":"What is your return policy"}]
)
```

For example, imagine you're fine-tuning a support bot that will answer questions about your product catalog. In this case, you can build *training_data.jsonl* with user questions and ideal replies drawn from your FAQ. You can then fine-tune the model with a few hundred examples for three epochs, then deploy it and call it to verify whether it gives responses that are grounded in your catalog data, rather than generic answers. The model will prioritize the user-provided data in its responses, but you can adjust it to lean more on the preexisting knowledge base if you prefer.

Be aware that fine-tuning incurs both token-based training charges and hourly hosting fees. For example, hosting a GPT-3.5 Turbo fine-tuned model costs around $7 per hour, in addition to training costs per million tokens. Also note that fine-tuning is resource-intensive, and therefore best reserved for scenarios where its benefits justify the investment. You'll need to weigh the improved accuracy from fine-tuning against the additional operational expenses. You may want to consider an alternative approach—such as retrieval-augmented generation, discussed later in this chapter—which may reduce costs by avoiding hosting fees.

Integrating Data Sources

There are two main ways to integrate personal data sources. The first is through the chat playground, where you can enrich model prompts with your own data. You can upload files directly, connect to Azure Blog Storage, or link to an existing Azure AI Search index. Supported file types include *.md*, *.txt*, *.html*, *.pdf*, and *.docx*.

Prepare at least 100 high-quality, representative examples, formatted in JSONL. Start with a small, curated set of around 50 clean examples, then expand. Clean your dataset by removing duplicates, balancing class distributions, normalizing text, and splitting the data into training and validation sets. Store large datasets in Azure Blob Storage, and use a centralized training hub to promote security and compliance by separating data scientists from direct access to raw data.

You can also use Azure AI Foundry to create and manage search resources and indices. This will let you perform efficient data segmentation and indexing, which are crucial for improving the quality of model responses. A data preparation script can be used to refine large text files or documents before indexing.

To further improve search results, you can enable semantic search in Azure AI Search. This can significantly enhance relevance, but it may also incur increased operational costs.

You'll connect your data via the "Add your data" tab in the Assistant setup pane, which provides options to link new data sources. Once you connect to your repositories, you can integrate them into the search index. For custom indices, the platform supports manual column mapping, allowing you to fine-tune how the model accesses and uses the data, thereby improving the relevance of its responses.

Interacting with the Model

You can interact with your model with the chat playground or the API. Each call you make will automatically include the necessary tokens. When using your own data, responses can include up to 1,500 tokens.

If you use the API, you'll also need to specify where the data you want to use is stored. Here's an example of a request body:

```
{
  "sourceConfigurations": [
    {
      "serviceType": "AzureSearchIntegration",
      "settings": {
        "serviceEndpoint": "your_search_service_endpoint",
        "apiKey": "your_api_key",
        "index": "your_index_name"
      }
    }
  ],
  "dialogue": [
    {
      "entity": "system",
      "text": "As an efficient assistant,
      your task is to guide users with suggestions on travel plans."
    },
    {
      "entity": "user",
      "text": "I'm looking to visit New York.
      Can you suggest some places to stay?"
    }
  ]
}
```

The request body is structured to integrate a custom data source and define a conversational context for the AI model. It begins with a `sourceConfigurations` section, where the data repository's connection details are specified. This includes the type of integration service (`AzureSearchIntegration`), along with its endpoint, API key, and index name, which together provide the model with access to the relevant dataset.

The second section, `dialogue`, defines the conversational framework for the assistant. It includes predefined entities and text that provide a system-level instruction that establishes the assistant's role (e.g., guiding users in their travel plans) and an example

user query. This setup ensures that the assistant has both the necessary data access and an initial context for delivering relevant responses.

Retrieval-Augmented Generation

Retrieval-augmented generation (RAG) is a technique that enhances the capabilities of large language models like GPT-4 by integrating them with an information retrieval system. This combination allows the models to fetch relevant information from external data sources and use it to generate responses that are more accurate and up-to-date. RAG addresses the limitation of LLMs that rely solely on their training data, which might be outdated or insufficient to meet specific needs.

Understanding RAG

The process of RAG begins with a user query. This query is first sent to a retrieval system, such as a search index or a vector database, which finds relevant documents or data chunks. These retrieved pieces of information are then combined with the original query to form an augmented input, which the LLM processes to generate a response. This approach ensures that the model has access to the most relevant and up-to-date information, which enhances the quality of its responses.

Real-life use cases of RAG

RAG has numerous real-life applications across various industries. In customer support, it can help AI systems create more accurate and contextually relevant responses by retrieving information from a company's knowledge base, past tickets, and documentation. For example, a customer support chatbot can use RAG to fetch the latest troubleshooting steps or policy details, which will improve the efficiency and accuracy of its responses.

In the healthcare sector, RAG can assist medical professionals by providing the most recent research findings, guidelines, and patient data. This helps them make informed decisions and provide better patient care. In the legal industry, RAG can aid lawyers by retrieving relevant case law, statutes, and legal documents, ensuring that their arguments are well supported by the most pertinent information.

Using RAG in Azure OpenAI

Azure OpenAI integrates RAG capabilities by combining Azure AI Search with OpenAI's LLMs. This integration allows developers to create applications that can retrieve and use specific relevant data to generate more accurate, context-aware responses. Azure provides tools such as Azure AI Foundry and Azure AI Document Intelligence to support this process. Azure AI Search also supports various search modes—including full-text search, semantic search, and vector search—to efficiently retrieve both structured and unstructured data.

Effectively implementing RAG in Azure OpenAI requires following a few key steps. First, you need to set up and manage indexes using Azure AI Search. The indexes store data in a structured format to allow for efficient retrieval based on queries. Next, break down your documents into manageable chunks and convert them into vectors using embedding models. This supports vector and semantic search capabilities.

Implement the RAG pattern by combining the retrieved content with the user query and sending this augmented input to the LLM. Azure AI Foundry provides tools and workflows to streamline this process. Finally, make use of Azure AI Search features such as semantic ranking and hybrid search to improve the relevance and quality of the retrieved data and help ensure that the LLM generates accurate and contextually appropriate responses.

By leveraging these capabilities, you can build powerful, context-aware applications that provide accurate and relevant responses based on real-time, domain-specific data.

Performance optimization strategies for RAG

For production deployments, here are various optimization strategies that you can employ to improve performance and efficiency. These include:

- Implementing multilevel caching of retrieval outputs with a system such as RAGCache, which stores intermediate knowledge states in both GPU and host memory to eliminate redundant retrieval and inference work
- Structuring your data in hierarchical indexes, where a top-level summary index quickly narrows the search space before a detailed vector search is conducted
- Using chunking to keep individual retrieval calls small
- Periodically rebuilding and fine-tuning approximate nearest neighbor (ANN) index structures to maintain an optimal balance between recall and latency
- Caching popular queries at the application level so repeated requests can be served directly from memory without hitting the retrieval layer

Practical: Implementing RAG with Azure OpenAI and Azure AI Search

Suppose you're an IT specialist at a healthcare organization who's tasked with developing an intelligent assistant that can answer employee queries about available health plans. To achieve this, you'll implement a RAG solution using Azure OpenAI and Azure AI Search. Your assistant will retrieve relevant information from your organization's health plan documents and generate accurate, context-aware responses to user inquiries.

Step 1: Create an Azure AI Search resource

Start by creating an Azure AI Search resource in the Azure portal:

A. Search for "Azure AI Search," and select it from the search results.
B. On the Azure AI Search page, click Create.
C. Fill in the required details:
    i. Subscription: select your Azure subscription.
    ii. Resource group: select the resource group you created earlier.
    iii. Service name: enter **`myaisearchservicerg`** or something similar.
    iv. Location: choose the same region you used earlier (e.g., East US).
    v. Pricing tier: select a pricing tier that suits your needs (e.g., Basic for testing).
D. Click "Review + create" and, when validation is complete, click Create.
E. Wait for the deployment to complete. Once it does, click "Go to resource."
F. Note the endpoint URL and API key:
    i. Copy the URL (e.g., *https://myaisearchservice.search.windows.net/*) on the Overview page, under Essentials.
    ii. Select Keys under Settings in the lefthand menu, and copy the primary admin key.

Step 2: Deploy models in Azure OpenAI

Next, deploy the chat and embedding models:

A. Navigate to your Azure OpenAI Service resource on Azure AI Foundry:
    i. Select "Resource groups" in the left menu, and select the resource group you've provisioned.
    ii. Click on `MyOpenAIResource`.
B. In the left menu, click Deployments.
C. Click Create to create a new deployment for the chat model, and configure it as follows:
    i. Model: select "gpt-35-turbo."
    ii. Version: ensure that the version is 1106, or the latest one that's available.
    iii. Deployment name: enter **`chat-model`**.
    iv. Scale settings: leave the default settings, unless you have specific requirements.
D. Click Deploy.

E. Repeat the preceding steps to create a new deployment for the embedding model. Configure it as follows:

   i. Model: select "text-embedding-ada-002."

   ii. Version: ensure that the version is 2 or the latest version that's available.

   iii. Deployment name: enter **`embedding-model`**.

   iv. Scale settings: leave the default settings, unless you have specific requirements.

F. Click Deploy.

G. Ensure that both the `chat-model` and the `embedding-model` show as Deployed before proceeding to the next step.

Step 3: Set up an Azure AI Search index

Now, start preparing your dataset:

A. Collect all the available documents that are related to your organization's health plans. These can be in formats like *.pdf*, *.docx*, or *.txt*.

B. Ensure that each document has clear and consistent formatting to facilitate indexing.

C. Navigate to your Azure AI Search resource:

   i. In the Azure portal, choose "Resource groups" in the left menu and select `RAGResourceGroup`.

   ii. Click on `MyAISearchService`.

D. In the left menu, choose Indexes.

E. On the Indexes page, click "Add index."

F. Configure the index:

   i. Name: enter **`healthplans-index`**.

   ii. Define the fields for your index. At a minimum, include the fields in Table 9-2.

Table 9-2. Index fields to be used for Azure AI Search

Field name	Type	Key	Retrievable	Searchable	Filterable	Sortable
`id`	`Edm.String`	Yes	Yes	No	No	No
`content`	`Edm.String`	No	Yes	Yes	No	No

G. Import the required data:

   i. After creating the index, click "Import data."

ii. Data source: choose Azure Blob Storage or another supported data source where your health plan documents are stored.

iii. Data source name: enter an appropriate name for your data source.

iv. Connection string: provide the connection string to your data source.

v. Content: ensure that this field is mapped correctly to extract text from your documents.

H. Review your settings and click Create to start the indexing process.

I. The indexing process may take some time, depending on the number and size of your documents. Once it's completed, the status will show as Ready.

Step 4: Configure environment variables

To continue configuring the resources, you'll have to set up some environment variables to securely store your Azure service credentials and endpoint URLs. You can define them in your local development environment or within your application's configuration:

A. In your project directory, create a file named *.env*.

B. Open the file in a text editor and ad the following lines:

```
AZURE_OPENAI_ENDPOINT="https://myopenairesource.openai.azure.com/"
AZURE_OPENAI_API_KEY="your_openai_api_key"
AZURE_OPENAI_DEPLOYMENT_ID_CHAT="chat-model"
AZURE_OPENAI_DEPLOYMENT_ID_EMBEDDING="embedding-model"
AZURE_AI_SEARCH_ENDPOINT="https://myaisearchservice.search.windows.net/"
AZURE_AI_SEARCH_API_KEY="your_search_api_key"
AZURE_AI_SEARCH_INDEX="healthplans-index"
```

Replace *your_openai_api_key* with your actual Azure OpenAI API key and *your_search_api_key* with your actual Azure AI Search API key.

Also ensure that the `AZURE_OPENAI_ENDPOINT` and `AZURE_AI_SEARCH_ENDPOINT` match the endpoints you noted earlier.

Step 5: Implement the RAG solution

Now, create a new Python environment:

A. Navigate to the directory where you want to set up the project.

B. Create a project directory and move into it with the following commands:

```
mkdir azure-openai-rag
cd azure-openai-rag
```

C. Create a virtual environment:

```
python -m venv venv
```

D. Activate the virtual environment with:

```
venv\Scripts\activate
```

E. Install the required packages using `pip`:

```
pip install openai azure-search-documents python-dotenv
```

F. In your project directory, create a file named *rag.py*.

G. Open the file in your IDE or text editor, and add the following lines to import the required modules and libraries:

```
import os
import openai
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from azure.search.documents.models import QueryType
from dotenv import load_dotenv
```

These include `os` for handling environment variables, `openai` for interacting with OpenAI's API, and Azure-specific libraries like *AzureKeyCredential*, *SearchClient*, and *QueryType* for connecting to Azure AI Search. Additionally, you'll need to import *dotenv* to load environment variables securely from a *.env* file.

H. Next, you'll use the `load_dotenv` function to load environment variables from the *.env* file and set up your credentials for the Azure OpenAI Service and Azure AI Search:

```
load_dotenv()
openai.api_type = "azure"
openai.api_key = os.getenv("AZURE_OPENAI_API_KEY")
openai.api_base = os.getenv("AZURE_OPENAI_ENDPOINT")
openai.api_version = "2024-02-01"  # Use the latest API version

chat_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT_ID_CHAT")
embedding_deployment = os.getenv("AZURE_OPENAI_DEPLOYMENT_ID_EMBEDDING")
search_endpoint = os.getenv("AZURE_AI_SEARCH_ENDPOINT")
search_api_key = os.getenv("AZURE_AI_SEARCH_API_KEY")
index_name = os.getenv("AZURE_AI_SEARCH_INDEX")
```

You can then initialize the `SearchClient` from Azure AI Search, which connects to the specified search endpoint and index using the API key. This client will handle document retrieval from the Azure AI Search index:

```
search_client = SearchClient(
    endpoint=search_endpoint,
    index_name=index_name,
    credential=AzureKeyCredential(search_api_key)
)
```

I. Use the `search_documents` function to perform a search on the Azure AI Search index using the provided query. It retrieves up to `top_k` documents (the default is 5) and returns them as a list:

```
def search_documents(query, top_k=5):
    """
    Search the Azure AI Search index for documents matching the query.
    """
    results = search_client.search(
        search_text=query,
        query_type=QueryType.FULL,
        top=top_k
    )
    documents = [doc for doc in results]
    return documents
```

This function is crucial for finding contextually relevant documents based on user input.

J. Next, you'll define an optional function that generates embeddings for a given input text using the specified embedding model. While it's not used in the main workflow, you can employ it for more advanced similarity searches or for improving search relevance:

```
def get_embeddings(text):
    """
    Get embeddings for the input text using the embedding model.
    """
    response = openai.Embedding.create(
        engine=embedding_deployment,
        input=text
    )
    embeddings = response['data'][0]['embedding']
    return embeddings
```

The `get_openai_response` function sends a prompt to the OpenAI chat model and retrieves the generated response.

K. This allows customization of response generation through parameters like `max_tokens`, `temperature`, and `top_p`:

```
def get_openai_response(prompt):
    """
    Get a response from the OpenAI chat model based on the prompt.
    """
    response = openai.Completion.create(
        engine=chat_deployment,
        prompt=prompt,
        max_tokens=150,
        temperature=0.7,
        top_p=0.9,
        frequency_penalty=0,
        presence_penalty=0
    )
    return response.choices[0].text.strip()
```

L. Finally, the main block initiates the program by prompting the user to input a query and searches for relevant documents using the `search_documents` function: If it doesn't find any documents, it will inform the user of that. Otherwise, it will create a contextual prompt using the content of the retrieved documents and the user query, send the prompt to the OpenAI model to generate a response, and display that response to the user.

```
if __name__ == "__main__":
    user_query = input("Enter your query about available health plans: ")
    documents = search_documents(user_query)

    if not documents:
        print("No relevant documents found.")
    else:
        context = "\n".join([doc["content"] for doc in documents])
        prompt = (
            "Given these documents:\n\n"
            f"{context}\n\n"
            f"Answer the question: {user_query}"
        )
        answer = get_openai_response(prompt)
        print("\nAnswer:")
        print(answer)
```

Step 6: Run the script

You can now execute the script and see your retrieval-augmented generation workflow in action in your application. Follow these steps:

A. Ensure that the virtual environment is activated. Your command prompt should show `(venv)` at the beginning.

B. Run the script with:

```
python rag.py
```

C. Interact with the application. Here's an example of a conversation you could have:

You: What are my available health plans?

Chatbot: Based on the documents, your organization offers three following health plans. The Basic Health Plan covers essential medical services, including general practitioner visits, emergency care, and hospitalization. The Premium Health Plan includes all features of the Basic plan, plus additional benefits such as dental and vision coverage, specialist consultations, and wellness programs. The Family Health Plan is designed for employees with dependents; offers comprehensive coverage for family members under a single policy.

D. You can choose the plan that best fits your healthcare needs. For more detailed information, please refer to the specific plan documents or contact the HR department.

Step 7: Test and refine the application

Before deploying the application, test it thoroughly, verify that everything is set up correctly, and refine as needed:

A. Test the application with various queries to ensure it retrieves relevant information and generates accurate responses. For example, you can enter the following queries:

   i. **`What does the Premium Health Plan include?`**

   ii. **`How can I enroll in a health plan?`**

   iii. **`Are dental services covered under any plan?`**

B. Verify the environment variables. Ensure that all environment variables in the *.env* file are correctly set and correspond to your Azure resources.

C. Check the index configuration. Make sure that all necessary fields in the Azure AI Search index are marked as Retrievable and Searchable.

D. Optimize search parameters:

   i. Adjust the `top_k` parameter in the `search_documents` function to retrieve more or fewer documents, based on your needs.

   ii. Experiment with different search queries and parameters to improve the relevance of the retrieved documents.

E. Refine the prompt structure in the *rag.py* script to provide clearer context to the OpenAI model, which will lead to the model giving more accurate responses.

F. Add error handling to manage scenarios where no documents are found or API requests fail. Then, create an enhanced version of the `search_documents` function that includes a `try_except` block to handle potential errors during the search process:

```
def search_documents(query, top_k=5):
    try:
        results = search_client.search(
            search_text=query,
            query_type=QueryType.FULL,
            top=top_k
        )
        documents = [doc for doc in results]
        return documents
    except Exception as e:
        print(f"Error during search: {e}")
        return []
```

Now, if the Azure AI Search API call fails (e.g., due to connectivity issues or incorrect query parameters), the exception will be caught, an error message will be printed, and an empty list will be returned. This will ensure that the program will not crash and will be able to gracefully handle scenarios in which document retrieval fails.

G. Update the `gen_openai_response` function to use a `try_except` block to handle errors that may occur during the OpenAI API call:

```
def get_openai_response(prompt):
    try:
        response = openai.Completion.create(
            engine=chat_deployment,
            prompt=prompt,
            max_tokens=150,
            temperature=0.7,
            top_p=0.9,
            frequency_penalty=0,
            presence_penalty=0
        )
        return response.choices[0].text.strip()
    except Exception as e:
        print(f"Error during OpenAI request: {e}")
        return "I'm sorry, I couldn't process your request at the moment."
```

H. Now, if the API request fails (e.g., due to invalid credentials, a network issue, or exceeding token limits), the program won't crash. Instead, the exception will be caught and a user-friendly message will be returned. This fallback response will help maintain a positive user experience, even during API failures.

Best practices

There are several best practices that you can use to ensure that your retrieval-augmented generation workflow is implemented properly and generates appropriate results. For example:

- Optimize API calls and monitor usage to control costs.
- Use version control, write clear, modular code, and include unit tests.
- Craft concise, context-rich prompts for high-quality outputs.
- Isolate dependencies with virtual environments and keep them updated.

And with that, you've successfully implemented a retrieval-augmented generation solution using the Azure OpenAI Service and Azure AI Search. This intelligent assistant can retrieve relevant information from your organization's health plan documents and provide accurate, context-aware responses to user queries.

Chapter Review

In this chapter, we explored what generative AI looks like in the Azure ecosystem and how to work with the Azure OpenAI Service. You learned how to select an appropriate model from the ones offered, how to perform prompt engineering effectively, and how to optimize AI models.

To be successful on the exam, you'll need to know how to do the following things that we covered in this chapter:

- Use the Azure OpenAI Service to generate text, images, and code.
- Write effective prompts to enhance the quality and relevance of AI-generated outputs.
- Optimize generative AI, based on your own data and parameters.

In the next chapter, we'll take a look at the future of AI on Azure.

Chapter Quiz

1. What is the first step you must take to access OpenAI models like GPT-4 and DALL-E through Azure?
   A. Deploy a model with Azure Machine Learning.
   B. Create an Azure OpenAI Service resource within your Azure subscription.
   C. Directly access the OpenAI API with your Azure credentials.
   D. Install the OpenAI Python SDK in your Azure environment.
2. Which Azure OpenAI model would you select to generate a high-resolution image from a text description?
   A. GPT-4 Turbo with Vision
   B. An embedding model
   C. DALL-E
   D. Whisper
3. How do you submit a prompt to the Azure OpenAI Service to generate natural language content?
   A. By using the Azure AI services API
   B. By sending a request to the Azure OpenAI endpoint
   C. By uploading a text file to Azure Blob Storage and linking it to OpenAI
   D. By configuring an Azure Logic App to process natural language prompts