Azure OpenAI Service provides a completion endpoint that can be used for a wide variety of tasks. The endpoint supplies a simple yet powerful text-in, text-out interface to any Azure OpenAI model. To trigger the completion, you input some text as a prompt. The model generates the completion and attempts to match your context or pattern. Suppose you provide the prompt "As Descartes said, I think, therefore" to the API. For this prompt, Azure OpenAI returns the completion endpoint " I am" with high probability.

The best way to start exploring completions is through the playground in Azure OpenAI Studio. It's a simple text box where you enter a prompt to generate a completion. You can start with a simple prompt like this one:

Console

Copy

```
Unset

write a tagline for an ice cream shop
```

After you enter your prompt, Azure OpenAI displays the completion:

Console

Copy

```
Unset

we serve up smiles with every scoop!
```

The completion results that you see can differ because the Azure OpenAI API produces fresh output for each interaction. You might get a slightly different completion each time you call the API, even if your prompt stays the same. You can control this behavior with the `Temperature` setting.

The simple text-in, text-out interface means you can "program" the Azure OpenAI model by providing instructions or just a few examples of what you'd like it to do. The output success generally depends on the complexity of the task and quality of your prompt. A general rule is to think about how you would write a word problem for a pre-teenage student to solve. A well-written prompt provides enough information for the model to know what you want and how it should respond.

Note

The model training data can be different for each model type. The latest model's training data currently extends through September 2021 only. Depending on your prompt, the model might not have knowledge of related current events.

Design prompts

Azure OpenAI Service models can do everything from generating original stories to performing complex text analysis. Because they can do so many things, you must be explicit in showing what you want. Showing, not just telling, is often the secret to a good prompt.

The models try to predict what you want from the prompt. If you enter the prompt "Give me a list of cat breeds," the model doesn't automatically assume you're asking for a list only. You might be starting a conversation where your first words are "Give me a list of cat breeds" followed by "and I'll tell you which ones I like." If the model only assumed that you wanted a list of cats, it wouldn't be as good at content creation, classification, or other tasks.

Guidelines for creating robust prompts

There are three basic guidelines for creating useful prompts:

- Show and tell. Make it clear what you want either through instructions, examples, or a combination of the two. If you want the model to rank a list of items in alphabetical order or to classify a paragraph by sentiment, include these details in your prompt to show the model.
- Provide quality data. If you're trying to build a classifier or get the model to follow a pattern, make sure there are enough examples. Be sure to proofread your examples. The model is smart enough to resolve basic spelling mistakes and give you a meaningful response. Conversely, the model might assume the mistakes are intentional, which can affect the response.
- Check your settings. Probability settings, such as `Temperature` and `Top P`, control how deterministic the model is in generating a response. If you're asking for a response where there's only one right answer, you should specify lower values for these settings. If you're looking for a response that's not obvious, you might want to use higher values. The most common mistake users make with these settings is assuming they control "cleverness" or "creativity" in the model response.

Troubleshooting for prompt issues

If you're having trouble getting the API to perform as expected, review the following points for your implementation:

- Is it clear what the intended generation should be?
- Are there enough examples?
- Did you check your examples for mistakes? (The API doesn't tell you directly.)
- Are you using the `Temperature` and `Top P` probability settings correctly?

Classify text

To create a text classifier with the API, you provide a description of the task and provide a few examples. In this demonstration, you show the API how to classify the *sentiment* of text messages. The sentiment expresses the overall feeling or expression in the text.

Console

Copy

Unset

```
This is a text message sentiment classifier

Message: "I loved the new adventure movie!"
Sentiment: Positive

Message: "I hate it when my phone battery dies."
Sentiment: Negative

Message: "My day has been 👍"
Sentiment: Positive

Message: "This is the link to the article"
Sentiment: Neutral

Message: "This new music video is unreal"
Sentiment:
```

Guidelines for designing text classifiers

This demonstration reveals several guidelines for designing classifiers:

- Use plain language to describe your inputs and outputs. Use plain language for the input "Message" and the expected value that expresses the "Sentiment." For best practices, start with plain language descriptions. You can often use shorthand or keys to indicate the input and output when building your prompt, but it's best to start by being as descriptive as possible. Then you can work backwards and remove extra words as long as the performance to the prompt is consistent.
- Show the API how to respond to any case. The demonstration provides multiple outcomes: "Positive," "Negative," and "Neutral." Supporting a neutral outcome is important because there are many cases where even a human can have difficulty determining if something is positive or negative.
- Use emoji and text, per the common expression. The demonstration shows that the classifier can be a mix of text and emoji 👍. The API reads emoji and can even convert expressions to and from them. For the best response, use common forms of expression for your examples.
- Use fewer examples for familiar tasks. This classifier provides only a handful of examples because the API already has an understanding of sentiment and the concept of a text message. If you're building a classifier for something the API might not be familiar with, it might be necessary to provide more examples.

Multiple results from a single API call

Now that you understand how to build a classifier, let's expand on the first demonstration to make it more efficient. You want to be able to use the classifier to get multiple results back from a single API call.

Console

Copy

Unset

```
This is a text message sentiment classifier

Message: "I loved the new adventure movie!"
Sentiment: Positive

Message: "I hate it when my phone battery dies"
Sentiment: Negative
```

```
Message: "My day has been 👍"
Sentiment: Positive

Message: "This is the link to the article"
Sentiment: Neutral

Message text
1. "I loved the new adventure movie!"
2. "I hate it when my phone battery dies"
3. "My day has been 👍"
4. "This is the link to the article"
5. "This new music video is unreal"

Message sentiment ratings:
1: Positive
2: Negative
3: Positive
4: Neutral
5: Positive

Message text
1. "He doesn't like homework"
2. "The taxi is late. She's angry 😠"
3. "I can't wait for the weekend!!!"
4. "My cat is adorable ❤️❤️"
5. "Let's try chocolate bananas"

Message sentiment ratings:
1.
```

This demonstration shows the API how to classify text messages by sentiment. You provide a numbered list of messages and a list of sentiment ratings with the same number index. The API uses the information in the first demonstration to learn how to classify sentiment for a single text message. In the second demonstration, the model learns how to apply the sentiment classification to a list of text messages. This approach allows the API to rate five (and even more) text messages in a single API call.

Important

When you ask the API to create lists or evaluate text, it's important to help the API avoid drift. Here are some points to follow:

- Pay careful attention to your values for the `Top P` or `Temperature` probability settings.
- Run multiple tests to make sure your probability settings are calibrated correctly.
- Don't use long lists. Long lists can lead to drift.

Trigger ideas

One of the most powerful yet simplest tasks you can accomplish with the API is generating new ideas or versions of input. Suppose you're writing a mystery novel and you need some story ideas. You can give the API a list of a few ideas and it tries to add more ideas to your list. The API can create business plans, character descriptions, marketing slogans, and much more from just a small handful of examples.

In the next demonstration, you use the API to create more examples for how to use virtual reality in the classroom:

Console

Copy

Unset

```
Ideas involving education and virtual reality

1. Virtual Mars
Students get to explore Mars via virtual reality and go on
missions to collect and catalog what they see.

2.
```

This demonstration provides the API with a basic description for your list along with one list item. Then you use an incomplete prompt of "2." to trigger a response from the API. The API interprets the incomplete entry as a request to generate similar items and add them to your list.

Guidelines for triggering ideas

Although this demonstration uses a simple prompt, it highlights several guidelines for triggering new ideas:

- Explain the intent of the list. Similar to the demonstration for the text classifier, you start by telling the API what the list is about. This approach helps the API to focus on completing the list rather than trying to determine patterns by analyzing the text.
- Set the pattern for the items in the list. When you provide a one-sentence description, the API tries to follow that pattern when generating new items for the list. If you want a more verbose response, you need to establish that intent with more detailed text input to the API.
- Prompt the API with an incomplete entry to trigger new ideas. When the API encounters text that seems incomplete, such as the prompt text "2.," it first tries to determine any text that might complete the entry. Because the demonstration had a list title and an example with the number "1." and accompanying text, the API interpreted the incomplete prompt text "2." as a request to continue adding items to the list.
- Explore advanced generation techniques. You can improve the quality of the responses by making a longer more diverse list in your prompt. One approach is to start with one example, let the API generate more examples, and then select the examples you like best and add them to the list. A few more high-quality variations in your examples can dramatically improve the quality of the responses.

Conduct conversations

Starting with the release of GPT-35-Turbo and GPT-4, we recommend that you create conversational generation and chatbots by using models that support the *chat completion endpoint*. The chat completion models and endpoint require a different input structure than the completion endpoint.

The API is adept at carrying on conversations with humans and even with itself. With just a few lines of instruction, the API can perform as a customer service chatbot that intelligently answers questions without getting flustered, or a wise-cracking conversation partner that makes jokes and puns. The key is to tell the API how it should behave and then provide a few examples.

In this demonstration, the API supplies the role of an AI answering questions:

Console

Copy

Unset

```
The following is a conversation with an AI assistant. The
assistant is helpful, creative, clever, and very friendly.

Human: Hello, who are you?
AI: I am an AI created by OpenAI. How can I help you today?
Human:
```

Let's look at a variation for a chatbot named "Cramer," an amusing and somewhat helpful virtual assistant. To help the API understand the character of the role, you provide a few examples of questions and answers. All it takes is just a few sarcastic responses and the API can pick up the pattern and provide an endless number of similar responses.

Console

Copy

Unset

```
Cramer is a chatbot that reluctantly answers questions.

###
User: How many pounds are in a kilogram?
Cramer: This again? There are 2.2 pounds in a kilogram. Please
make a note of this.
###
User: What does HTML stand for?
Cramer: Was Google too busy? Hypertext Markup Language. The T
is for try to ask better questions in the future.
###
User: When did the first airplane fly?
Cramer: On December 17, 1903, Wilbur and Orville Wright made
the first flights. I wish they'd come and take me away.
###
User: Who was the first man in space?
Cramer:
```

Guidelines for designing conversations

Our demonstrations show how easily you can create a chatbot that's capable of carrying on a conversation. Although it looks simple, this approach follows several important guidelines:

- Define the intent of the conversation. Just like the other prompts, you describe the intent of the interaction to the API. In this case, "a conversation." This input prepares the API to process subsequent input according to the initial intent.
- Tell the API how to behave. A key detail in this demonstration is the explicit instructions for how the API should interact: "The assistant is helpful, creative, clever, and very friendly." Without your explicit instructions, the API might stray and mimic the human it's interacting with. The API might become unfriendly or exhibit other undesirable behavior.
- Give the API an identity. At the start, you have the API respond as an AI created by OpenAI. While the API has no intrinsic identity, the character description helps the API respond in a way that's as close to the truth as possible. You can use character identity descriptions in other ways to create different kinds of chatbots. If you tell the API to respond as a research scientist in biology, you receive intelligent and thoughtful comments from the API similar to what you'd expect from someone with that background.

Transform text

The API is a language model that's familiar with various ways that words and character identities can be used to express information. The knowledge data supports transforming text from natural language into code, and translating between other languages and English. The API is also able to understand content on a level that allows it to summarize, convert, and express it in different ways. Let's look at a few examples.

Translate from one language to another

This demonstration instructs the API on how to convert English language phrases into French:

Console

Copy

Unset

```
English: I do not speak French.
French: Je ne parle pas français.
English: See you later!
French: À tout à l'heure!
English: Where is a good restaurant?
French: Où est un bon restaurant?
English: What rooms do you have available?
French: Quelles chambres avez-vous de disponible?
English:
```

This example works because the API already has a grasp of the French language. You don't need to try to teach the language to the API. You just need to provide enough examples to help the API understand your request to convert from one language to another.

If you want to translate from English to a language the API doesn't recognize, you need to provide the API with more examples and a fine-tuned model that can produce fluent translations.

Convert between text and emoji

This demonstration converts the name of a movie from text into emoji characters. This example shows the adaptability of the API to pick up patterns and work with other characters.

Console

Copy

Unset

```
Carpool Time: 👨👴👩🚗🕑
Robots in Cars: 🚗🤖
Super Femme: 👸👸👸👸👸
Webs of the Spider: 🕸🕷🕸🕸🕷🕸
The Three Bears: 🐻🐼🐻
Mobster Family: 👨👩👧🕵️‍♀️👲💥
Arrows and Swords: 🏹🗡️🗡️🏹
```

```
Snowmobiles:
```

Summarize text

The API can grasp the context of text and rephrase it in different ways. In this demonstration, the API takes a block of text and creates an explanation that's understandable by a primary-age child. This example illustrates that the API has a deep grasp of language.

Console

Copy

Unset

```
My ten-year-old asked me what this passage means:
"""
A neutron star is the collapsed core of a massive supergiant
star, which had a total mass of between 10 and 25 solar
masses, possibly more if the star was especially
metal-rich.[1] Neutron stars are the smallest and densest
stellar objects, excluding black holes and hypothetical white
holes, quark stars, and strange stars.[2] Neutron stars have a
radius on the order of 10 kilometres (6.2 mi) and a mass of
about 1.4 solar masses.[3] They result from the supernova
explosion of a massive star, combined with gravitational
collapse, that compresses the core past white dwarf star
density to that of atomic nuclei.
"""

I rephrased it for him, in plain language a ten-year-old can
understand:
"""
```

Guidelines for producing text summaries

Text summarization often involves supplying large amounts of text to the API. To help prevent the API from drifting after it processes a large block of text, follow these guidelines:

- Enclose the text to summarize within triple double quotes. In this example, you enter three double quotes (""") on a separate line before and after the block of text to summarize. This formatting style clearly defines the start and end of the large block of text to process.
- Explain the summary intent and target audience before, and after summary. Notice that this example differs from the others because you provide instructions to the API two times: before, and after the text to process. The redundant instructions help the API to focus on your intended task and avoid drift.

Complete partial text and code inputs

While all prompts result in completions, it can be helpful to think of text completion as its own task in instances where you want the API to pick up where you left off.

In this demonstration, you supply a text prompt to the API that appears to be incomplete. You stop the text entry on the word "and." The API interprets the incomplete text as a trigger to continue your train of thought.

Console

Copy

```
Unset

Vertical farming provides a novel solution for producing food
locally, reducing transportation costs and
```

This next demonstration shows how you can use the completion feature to help write `React` code components. You begin by sending some code to the API. You stop the code entry with an open parenthesis `(`. The API interprets the incomplete code as a trigger to complete the `HeaderComponent` constant definition. The API can complete this code definition because it has an understanding of the corresponding `React` library.

Python

Copy

```
Unset

import React from 'react';
const HeaderComponent = () => (
```

Guidelines for generating completions

Here are some helpful guidelines for using the API to generate text and code completions:

- Lower the Temperature to keep the API focused. Set lower values for the `Temperature` setting to instruct the API to provide responses that are focused on the intent described in your prompt.
- Raise the Temperature to allow the API to tangent. Set higher values for the `Temperature` setting to allow the API to respond in a manner that's tangential to the intent described in your prompt.
- Use the GPT-35-Turbo and GPT-4 Azure OpenAI models. For tasks that involve understanding or generating code, Microsoft recommends using the `GPT-35-Turbo` and `GPT-4` Azure OpenAI models. These models use the new chat completions format.

Generate factual responses

The API has learned knowledge that's built on actual data reviewed during its training. It uses this learned data to form its responses. However, the API also has the ability to respond in a way that sounds true, but is in fact, fabricated.

There are a few ways you can limit the likelihood of the API making up an answer in response to your input. You can define the foundation for a true and factual response, so the API drafts its response from your data. You can also set a low `Temperature` probability value and show the API how to respond when the data isn't available for a factual answer.

The following demonstration shows how to teach the API to reply in a more factual manner. You provide the API with examples of questions and answers it understands. You also supply examples of questions ("Q") it might not recognize and use a question mark for the answer ("A") output. This approach teaches the API how to respond to questions it can't answer factually.

As a safeguard, you set the `Temperature` probability to zero so the API is more likely to respond with a question mark (?) if there's any doubt about the true and factual response.

Console

Copy

Unset

```
Q: Who is Batman?
A: Batman is a fictional comic book character.

Q: What is torsalplexity?
A: ?

Q: What is Devz9?
A: ?

Q: Who is George Lucas?
A: George Lucas is an American film director and producer
famous for creating Star Wars.

Q: What is the capital of California?
A: Sacramento.

Q: What orbits the Earth?
A: The Moon.

Q: Who is Egad Debunk?
A: ?

Q: What is an atom?
A: An atom is a tiny particle that makes up everything.

Q: Who is Alvan Muntz?
A: ?

Q: What is Kozar-09?
A: ?

Q: How many moons does Mars have?
A: Two, Phobos and Deimos.
```

```
Q:
```

Guidelines for generating factual responses

Let's review the guidelines to help limit the likelihood of the API making up an answer:

- Provide a ground truth for the API. Instruct the API about what to use as the foundation for creating a true and factual response based on your intent. If you provide the API with a body of text to use to answer questions (like a Wikipedia entry), the API is less likely to fabricate a response.
- Use a low probability. Set a low `Temperature` probability value so the API stays focused on your intent and doesn't drift into creating a fabricated or confabulated response.
- Show the API how to respond with "I don't know". You can enter example questions and answers that teach the API to use a specific response for questions for which it can't find a factual answer. In the example, you teach the API to respond with a question mark (?) when it can't find the corresponding data. This approach also helps the API to learn when responding with "I don't know" is more "correct" than making up an answer.

Work with code

The Codex model series is a descendant of OpenAI's base GPT-3 series that's been trained on both natural language and billions of lines of code. It's most capable in Python and proficient in over a dozen languages including C#, JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, SQL, and even Shell.

For more information about generating code completions, see Codex models and Azure OpenAI Service.

Next steps

- Learn how to work with the GPT-35-Turbo and GPT-4 models.
- Learn more about the Azure OpenAI Service models.