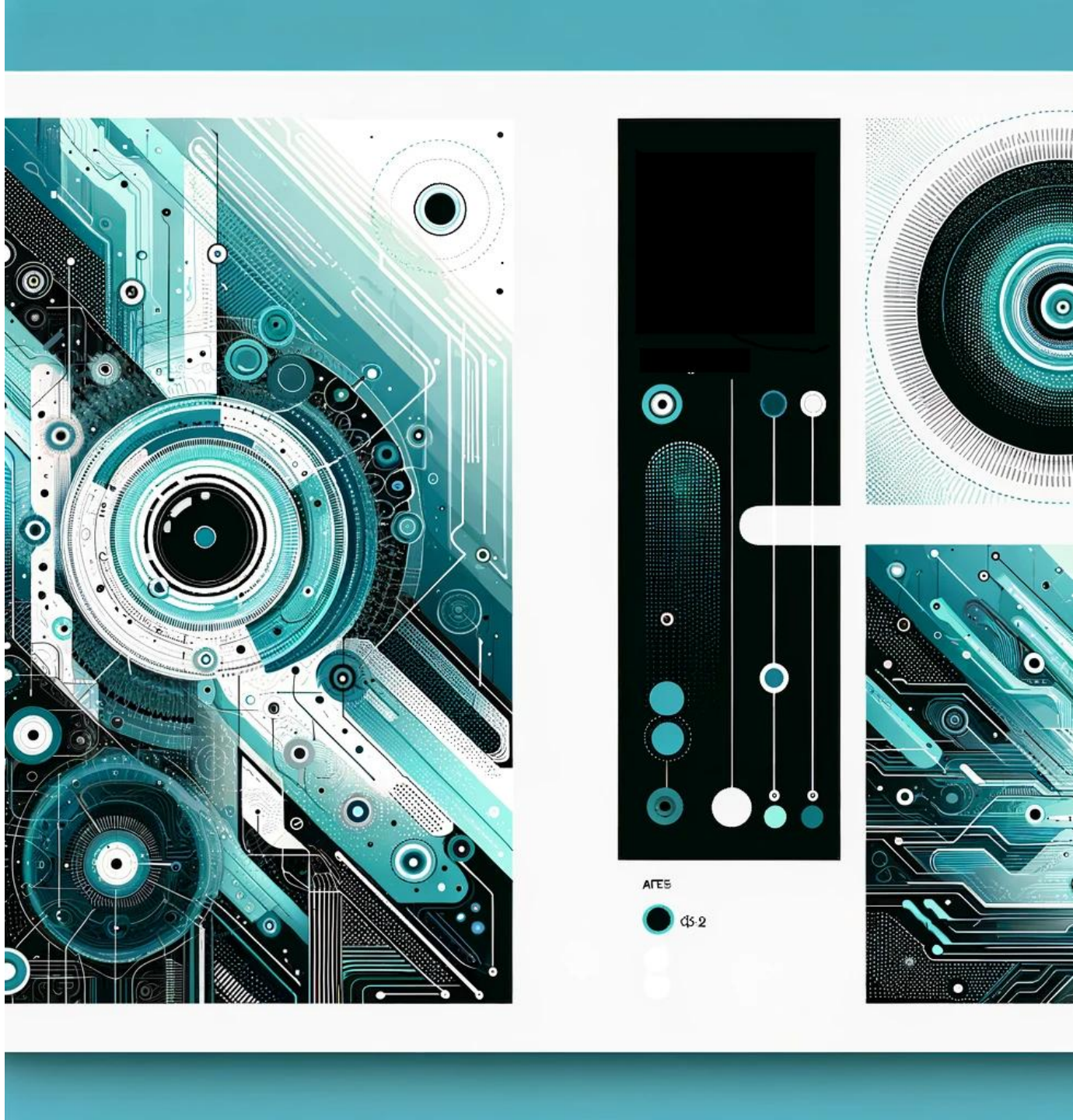
OpenAI APIs 101

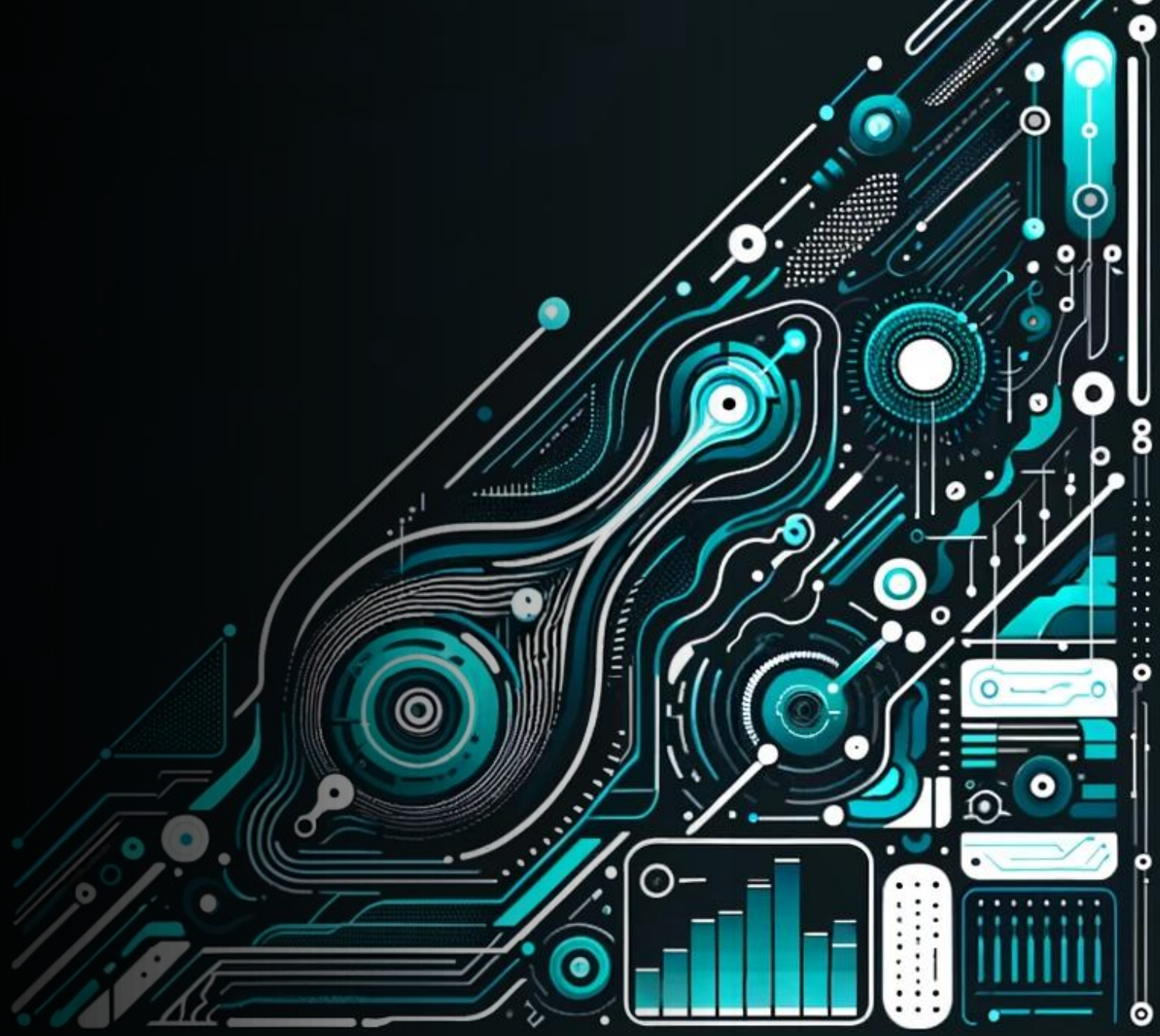
Yogev Shani

Agenda

- OpenAI API and Its Ecosystem
- Responses API
  - Quick Start
  - Text Generation
  - Conversation State
  - Function Calling
- Hand On Demo



OpenAI API and Its Ecosystem



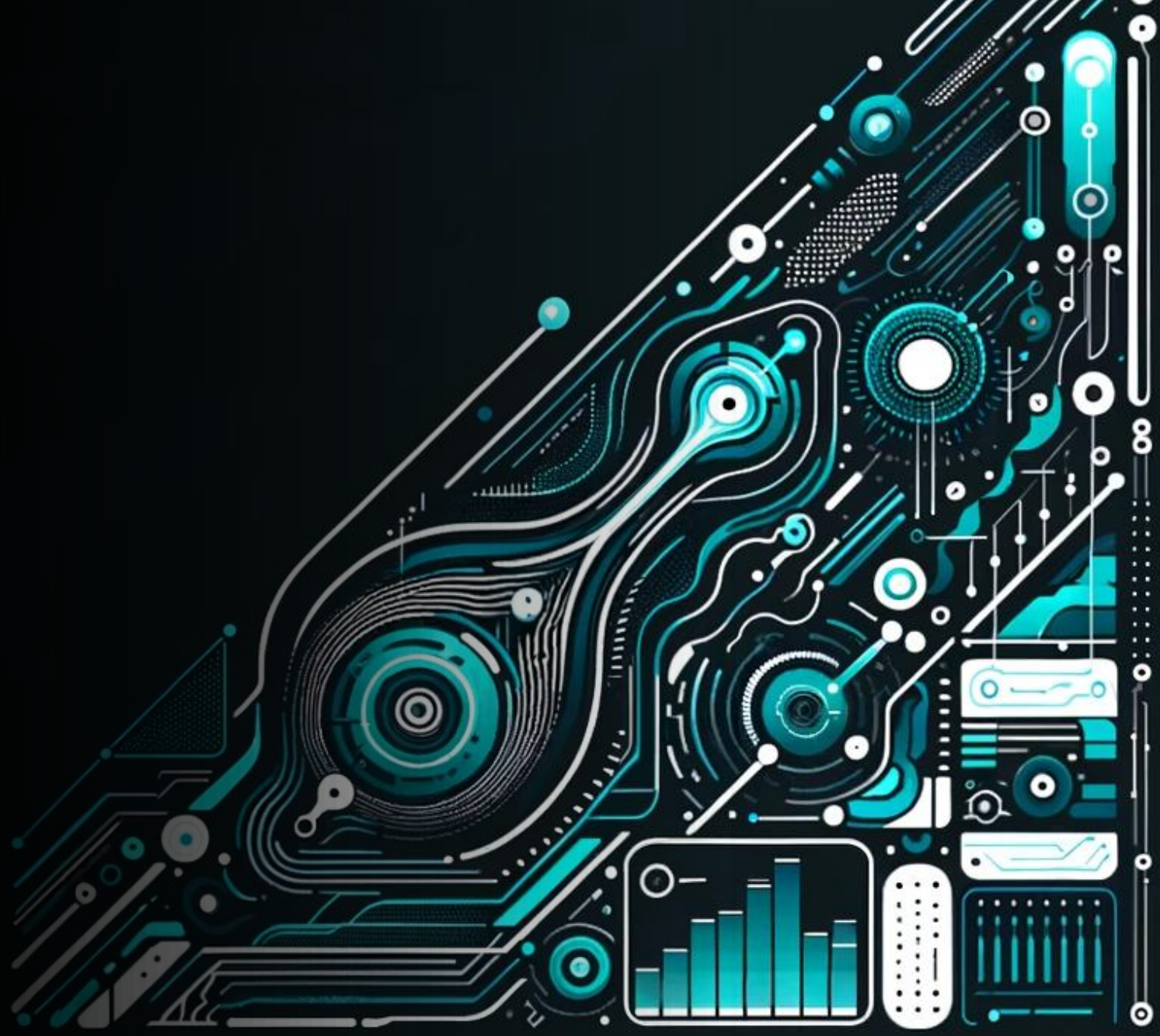
OpenAI API and Its Ecosystem

- **OpenAI API** is an API that provide multiple capabilities to interact with advanced **AI models** from OpenAI.
  - **AI Model:** A computational system designed to perform tasks that typically require human intelligence, such as:
    - **Language understanding**
    - **Image recognition**
    - **Speech to text**
    - **Code generation**
    - **Reasoning and decision making**

OpenAI API and Its Ecosystem

- **OpenAI APIs** provide access to **Large Language Models** (LLMs):
  - **LLM:** A model trained to understand, generate, and reason with human language.
- Examples of **OpenAI LLMs**:
  - GPT-5.4
  - GPT-5.2
  - GPT-4o-mini
  - GPT-4.1-nano
- **ChatGPT** is a web application powered by OpenAI's modern AI infrastructure:
  - Built on the **Responses API** and Assistants architecture
  - Supports **text, images, files, tools, and voice**
  - **Free and paid tiers** dynamically use **GPT-5.*** class models

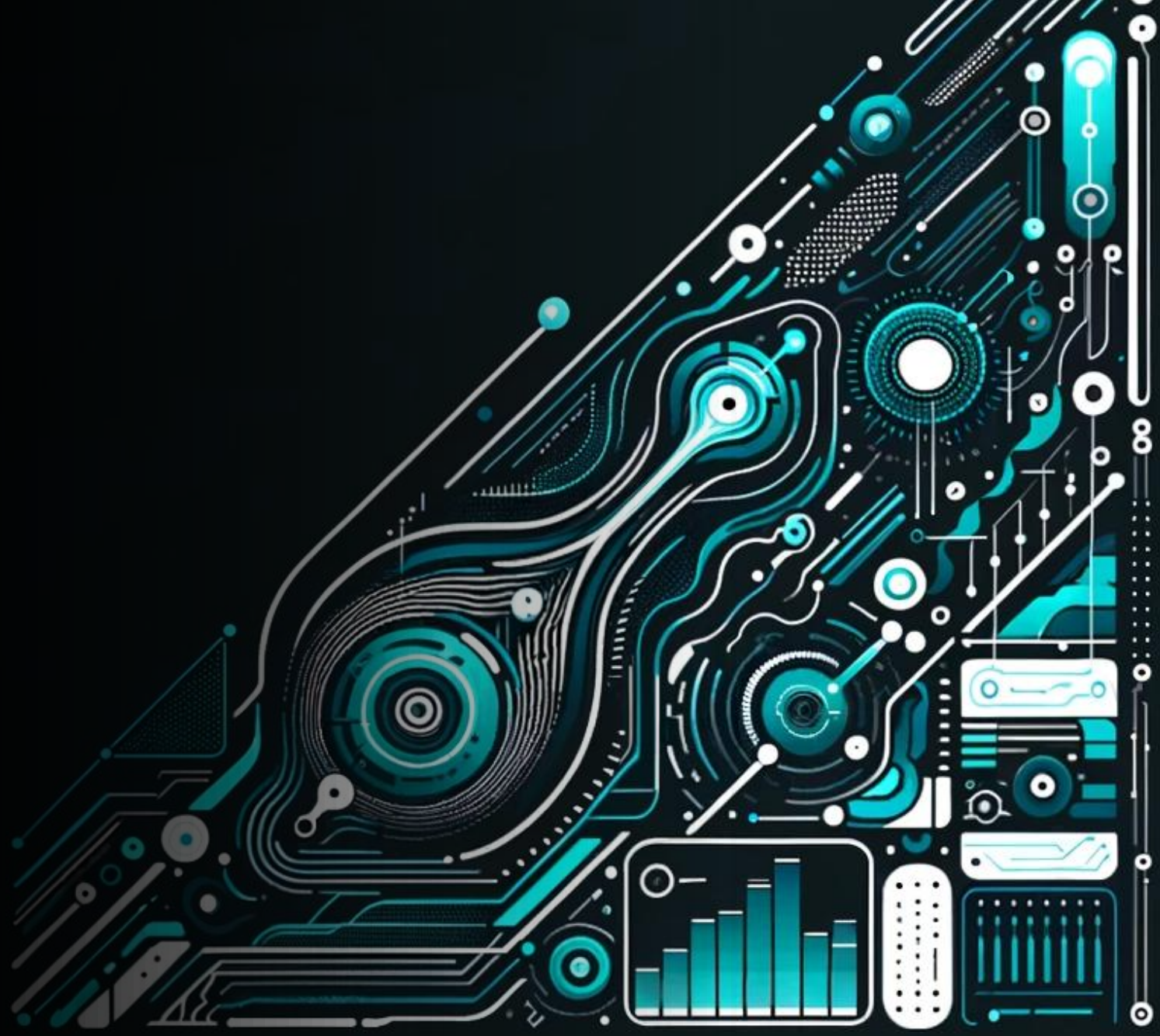
Responses API



OpenAI Responses API

OpenAI’s most advanced interface for generating model responses. Supports text and image inputs, and text outputs. Create stateful interactions with the model, using the output of previous responses as input. Extend the model’s capabilities with built-in tools for file search, web search, computer use, and more. Allow the model access to external systems and data using function calling.

Responses API Quick Start



Getting started with OpenAI API & 

- **Create** and **export** an API key
- **Install** the OpenAI SDK
- **Run** a basic API request
- Analyze **images** and **files**
- Extend the model with **tools**

Create and export an API key

- **Create an API key in the OpenAI dashboard**
  - Login at OpenAI developer platform
  - Go to API keys
  - Create a new secret key
- **Use this key to securely access the API**
- **Export the key to the OPENAI_API_KEY environment variable**
  - **macOS** or **Linux** systems: export OPENAI_API_KEY="your_api_key_here"
  - **Windows (PowerShell, temporary)**: $env:OPENAI_API_KEY="your_api_key_here"
  - **Windows (cmd, permanent)**: setx OPENAI_API_KEY "your_api_key_here"
    - Note: Changes apply to new terminals when using setx
- **OpenAI SDKs automatically read the API key from the system environment**

Install the OpenAI SDK

- **The OpenAI SDK can be installed in different languages using the relevant package manager.**
- **Install Python library**
  - *pip install openai*
- **Verify installation**

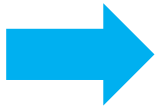
```
import openai

print( openai.__version__)
```

  - **And get:**
    - *2.11.0*

Run a basic API request

- With the OpenAI SDK installed, create a file called example.py and run a basic API Call code:

```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-4o-mini",
    input="Write a one-sentence bedtime story about a
unicorn."
)

print(response.output_text)
```



Once upon a time, a gentle unicorn named Luna danced among the stars each night, weaving dreams of magic and wonder for all the children in the world.

- In a few moments, you should see the output of your API request.

Analyze images

Send image URLs directly to the model to detect visual elements.

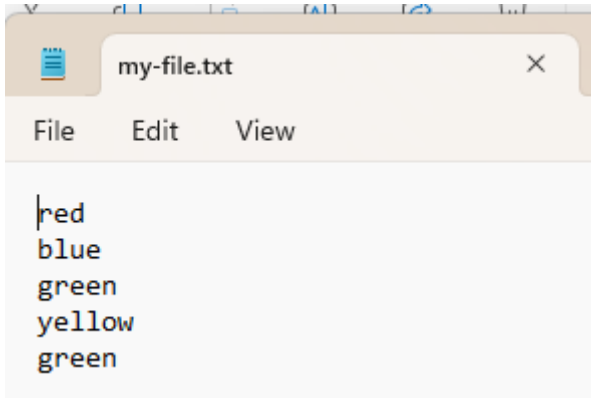
```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-5",
    input=[
        {
            "role": "user",
            "content": [
                {
                    "type": "input_text",
                    "text": "What do you see in the image?",
                },
                {
                    "type": "input_image",
                    "image_url": "https://api.nga.gov/iiif/a2e6da57-3cd1-4235-b20e-95dcaefed6c8/full/!800,800/0/default.jpg"
                }
            ]
        }
    ]
)
print(response.output_text)
```

*A painted portrait of a **young woman** seated on a **curved wooden chair** against a pale **green background**. She wears a **red-and-blue striped top** with **yellow buttons** and a **blue skirt with orange polka dots**. Her dark hair is tied back with a **red ribbon**, and she **holds a small bouquet** of white and pink flowers in her lap. The brushstrokes are bold and textured.*

Analyze files

Send uploaded files, or PDF documents directly to the model to extract text or classify content.

```
from openai import OpenAI
client = OpenAI()
with open("my-file.txt", "rb") as f:
    upload = client.files.create(file=f, purpose="user_data")

response = client.responses.create(
    model="gpt-5",
    input=[
        {
            "role": "user",
            "content": [
                {
                    "type": "input_text",
                    "text": "Analyze the list and provide a summary of the key values.",
                },
                {
                    "type": "input_file",  "file_url": "upload.id",
                },
            ],
        },
    ]
)
print(response.output_text)
```



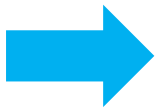
*The list contains the following colors: red, blue, green (**mentioned twice**), and yellow.*

Extend the model with tools

- Give the model access to external data and functions by attaching tools.
- Use built-in tools like web search or file search, or define your own for calling APIs, running code, or integrating with third-party systems.

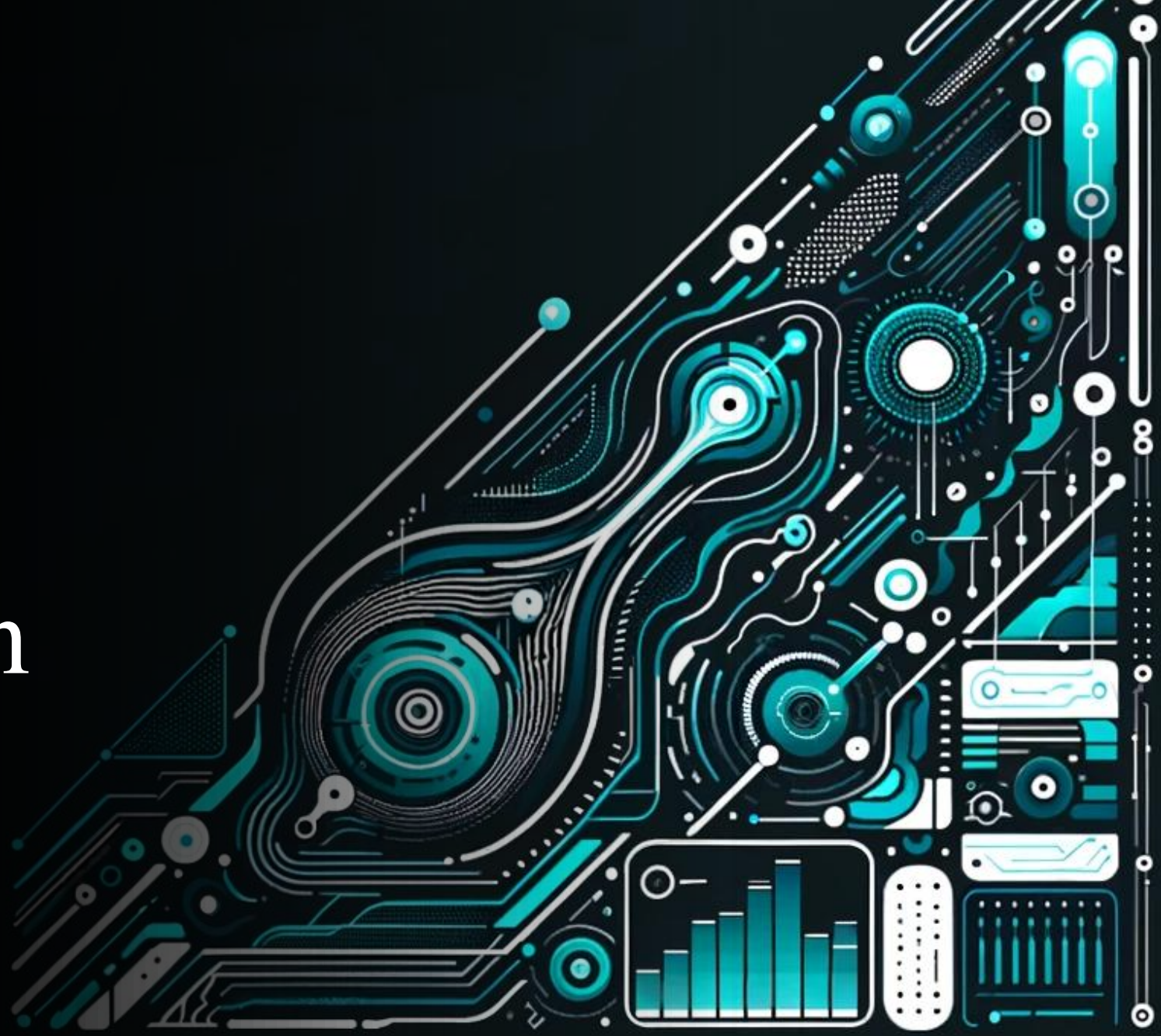
- Web search example:

```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-5",
    tools=[{"type": "web_search"}],
    input="What was a positive news story from
today?"
)

print(response.output_text)
```



Here's one: Today (March 25, 2026) is the American Red Cross's 12th annual Giving Day, a nationwide effort rallying donations to help families recover from disasters-an easy way people are coming together for something positive. ([redcross.org](https://www.redcross.org/donations/ways-to-donate/giving-day.html?utm_source=openai))

Text Generation



Text Generation - Request

With the OpenAI API, you can use a large language model to generate text from a prompt, as you might using ChatGPT. Models can generate almost any kind of text response - like code, mathematical equations, structured JSON data, or human-like prose.

```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-5.4",
    input="Write a one-sentence bedtime
story about a unicorn."
)

print(response.output_text)
```


```
[
  {
    "id": "msg_67b73f697ba4819183a15cc17d011509",
    "type": "message",
    "role": "assistant",
    "content": [
      {
        "type": "output_text",
        "text": "Under the soft glow of the moon, Luna the unicorn
danced through fields of twinkling stardust, leaving trails of
dreams for every child asleep.",
        "annotations": []
      }
    ]
  }
]
```

Text Generation - Response

An array of content generated by the model is in the output property of the response.

In this simple example, we have just one output which looks like this:


```
[
  {
    "id": "msg_67b73f697ba4819183a15cc17d011509",
    "type": "message",
    "role": "assistant",
    "content": [
      {
        "type": "output_text",
        "text": "Under the soft glow of the moon, Luna the unicorn danced through fields of twinkling stardust, leaving trails of dreams for every child asleep.",
        "annotations": []
      }
    ]
  }
]
```


Note 1: The output array often has more than one item in it! It can contain tool calls, data about reasoning tokens generated by reasoning models, and other items. It is not safe to assume that the model's text output is present at output[0].content[0].text.

Note 2: Python official SDK include an output_text property on model responses for convenience, which aggregates all text outputs from the model into a single string. This may be useful as a shortcut to access text output from the model.

```
print(response.output_text)
```

Provide instructions to the model

The instructions parameter gives the model high-level instructions on how it should behave while generating a response, including tone, goals, and examples of correct responses. Any instructions provided this way will take priority over a prompt in the input parameter.

```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-4o-mini",
    instructions="Talk like a pirate.",
    input="Are semicolons optional in
JavaScript?",
)

print(response.output_text)
```

Aye, matey! In the world of JavaScript, semicolons be often considered optional, thanks to a feature called Automatic Semicolon Insertion (ASI). The mighty parser tries to guess where ye meant to end a statement if ye be forgettin' one. But beware! This can lead to rough waters and unexpected errors if yer not careful.

'Tis a good practice to use semicolons to keep yer code shipshape and avoid any mishaps. So, hoist the sails and be diligent with yer semicolons, lest ye find yerself in a tempest of bugs! Arrr!

Messages priority based on different roles

OpenAI models give different levels of priority to messages with different roles.

- **Developer**

Developer messages are instructions provided by the application developer, prioritized ahead of user messages

- **User**

User messages are instructions provided by an end user, prioritized behind developer messages

- **Assistant**

Messages generated by the model have the assistant role

```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-5",
    reasoning={"effort": "low"},
    input=[
        {
            "role": "developer",
            "content": "Talk like a pirate."
        },
        {
            "role": "user",
            "content": "Are semicolons optional in JavaScript?"
        }
    ]
)

print(response.output_text)
```

Conversation State

Conversation State

OpenAI provides a few ways to manage conversation state, which is important for preserving information across multiple messages or turns in a conversation.

- Manually manage conversation state – You have full control
- OpenAI APIs for conversation state

Manually manage conversation state

- Each text generation request is independent and stateless, but we can still implement multi-turn conversations by providing additional messages as parameters to the text generation request.
- By using alternating *user* and *assistant* messages, you capture the previous state of a conversation in one request to the model.
  - ✓ *Assistant* role: The model's previous response, used for memory and continuity
  - ✓ *User* role: The current question or instruction
- To manually share context across generated responses, include the model's previous response output as input, and append that input to your next request.

```python
from openai import OpenAI

client = OpenAI()

response = client.responses.create(
    model="gpt-4o-mini",
    input=[
        {"role": "user", "content": "knock knock."},
        {"role": "assistant", "content": "Who's there?"},
        {"role": "user", "content": "Orange."},
    ],
)

print(response.output_text)
```

Orange who?

OpenAI APIs for conversation state

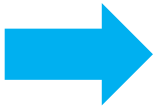
- OpenAI APIs can make it easier to manage conversation state automatically, so you don’t have to do pass inputs manually with each turn of a conversation:
  - Conversations API – use for persistence
  - Passing context from the previous response – use for simple chaining

Using the Conversations API

- The Conversations API works with the Responses API to persist conversation state as a long-running object with its own durable identifier. After creating a conversation object, you can keep using it across sessions, devices, or jobs.
- Conversations store items, which can be messages, tool calls, tool outputs, and other data.
- In a multi-turn interaction, you can pass the conversation into subsequent responses to persist state and share context across subsequent responses, rather than having to chain multiple response items together.

Using the Conversations API - Running it…

```
from openai import OpenAI
client = OpenAI()

conversation = client.conversations.create()

response = client.responses.create(
  model="gpt-4.1-mini",
  input=[{"role": "user", "content": "Tell me a short joke"}],
  conversation=conversation.id
)

print(response.output_text)

response = client.responses.create(
  model="gpt-4.1-mini",
  input= "explain the joke for me",
  conversation=conversation.id
)

print(response.output_text)
```



Why don't scientists trust atoms? Because they make up everything!

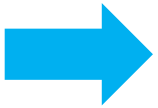
So, the joke is funny because it uses the double meaning: atoms "make up" everything (they compose everything physically), but it sounds like atoms are not trustworthy because they "make up" (invent) stories.

Passing context from the previous response

Another way to manage conversation state is to share context across generated responses with the previous_response_id parameter. This parameter lets you chain responses and create a threaded conversation.

Passing context from the previous response - Running it…

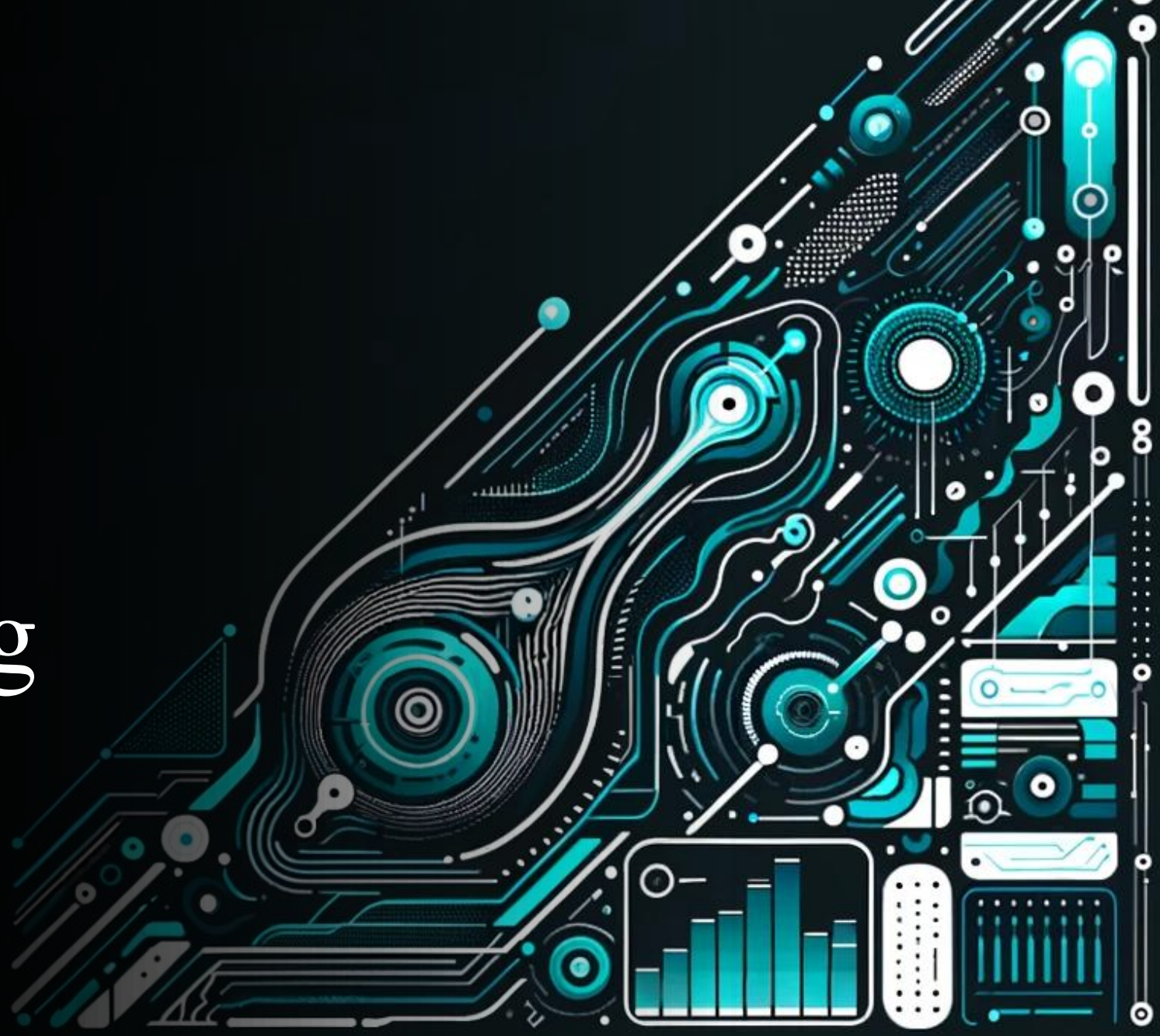
```
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-4o-mini",
    input="tell me a joke",
)
print(response.output_text)

second_response = client.responses.create(
    model="gpt-4o-mini",
    previous_response_id=response.id,
    input=[{"role": "user", "content": "explain why this is funny."}],
)
print(second_response.output_text)
```



Why don't scientists trust atoms? Because they make up everything!

So, the joke is funny because it uses the double meaning: atoms "make up" everything (they compose everything physically), but it sounds like atoms are not trustworthy because they "make up" (invent) stories.

Function calling



Function calling

Function calling (also known as tool calling) provides a powerful and flexible way for OpenAI models to interface with external systems and access data outside their training data.

Main terms:

- **Tools - functionality we give the model**
  - As the model generates a response to a prompt, it may decide that it needs data or functionality provided by a tool to follow the prompt's instructions.
- **Tool calls - requests from the model to use tools**
  - A function call or tool call refers to a special kind of response we can get from the model if it examines a prompt and then determines that in order to follow the instructions in the prompt, it needs to call one of the tools we made available to it.
- **Function tool**
  - A function is a specific kind of tool, defined by a JSON schema. A function definition allows the model to pass data to your application, where your code can access data or take actions suggested by the model.

Defining functions

Functions are usually declared in the tools parameter of each API request.

A function definition has the following properties:

- type: This should always be function
- name: The function’s name (e.g. get_weather)
- description: Details on when and how to use the function
- parameters: JSON schema defining the function’s input arguments

Pizza example

Here is an example function definition for a get_weather function.
Note: Your code executes the function, not the model

```
{
  "type": "function",
  "name": "get_weather",
  "description": "Retrieves current weather for the given location.",
  "parameters": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "City and country e.g. Bogotá, Colombia"
      },
      "units": {
        "type": "string",
        "enum": ["celsius", "fahrenheit"],
        "description": "Units the temperature will be returned in."
      }
    },
    "required": ["location", "units"],
    "additionalProperties": false
  }
}
```

HANDS ON DEMO

Thank You