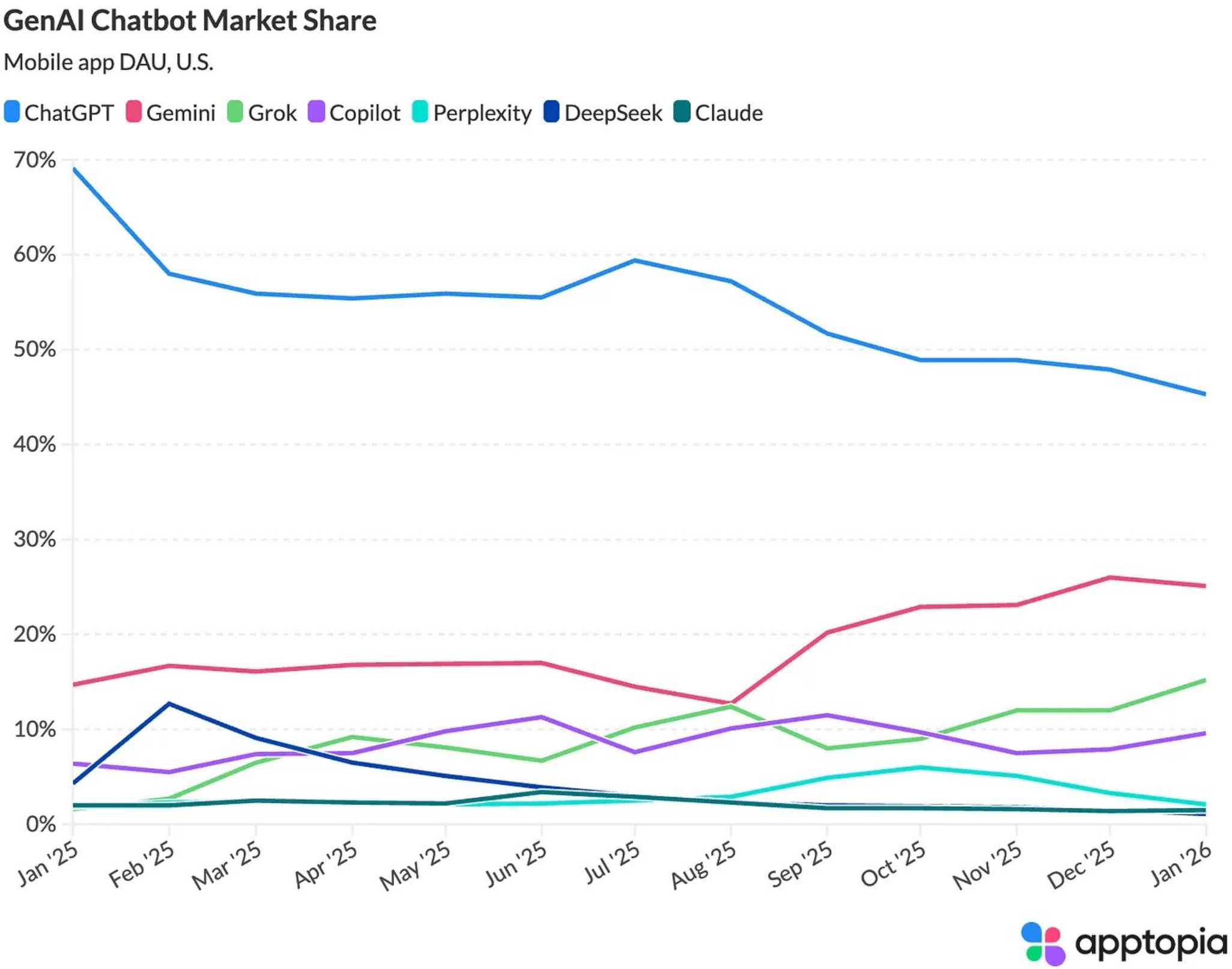
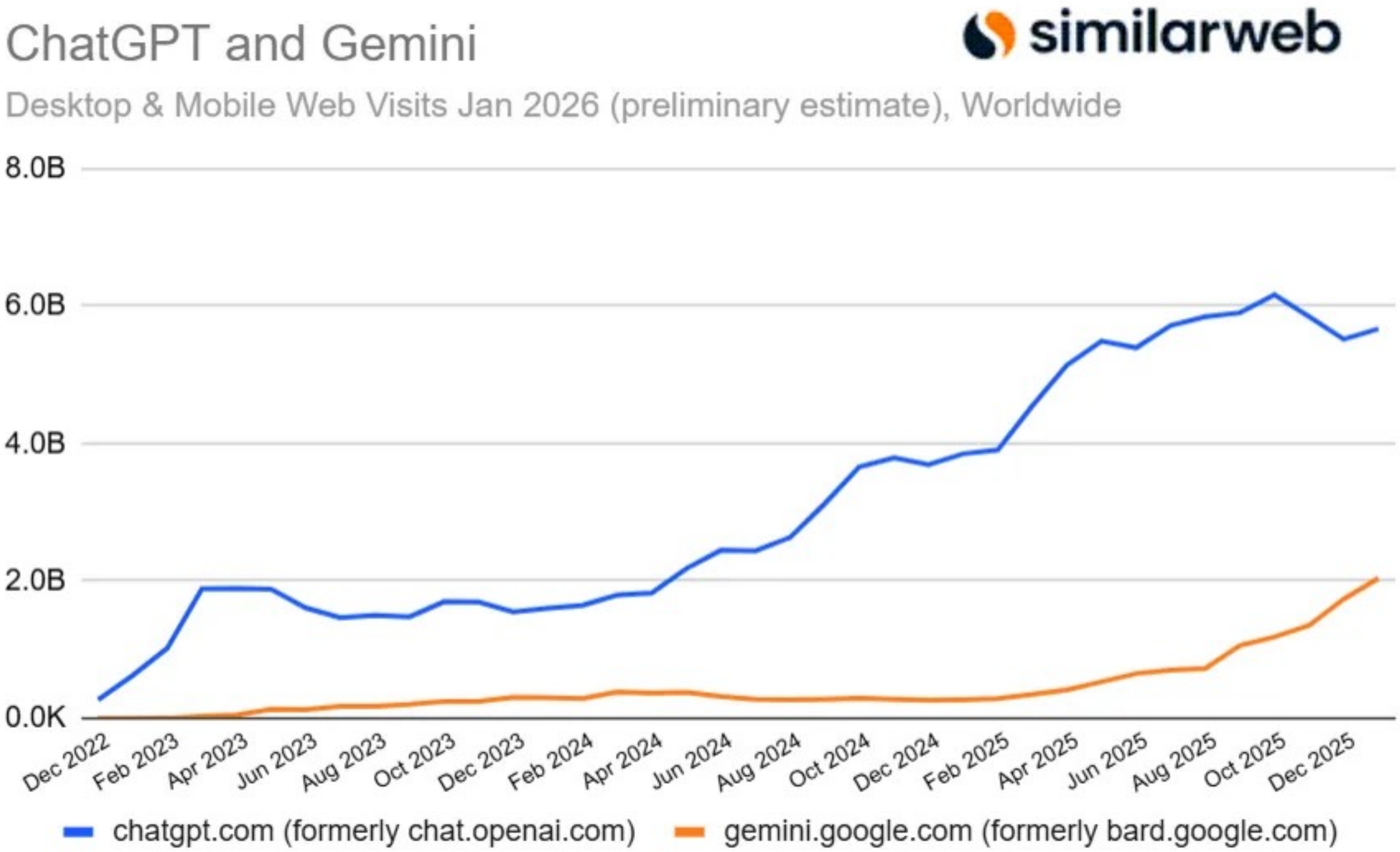
CS 668 Applied Large Language Models
Spring Semester, 2026

Doc 05 Tools, Skills, Agents
Feb 3, 2026



https://www.bigtechnology.com/p/new-data-openais-lead-is-contracting



https://www.bigtechnology.com/p/new-data-openais-lead-is-contracting

	Points
Working Code	15
Organization	5
Code Quality	5
Discussion of relevancy of return values	5

Formatting

Format your code

- Uniformly
- Consistently
- Show the block structure of your code

```
public void commandAction(Command c, Displayable d) {
   if (c == restartCmd) {
theGame.restart();
   } else if (c == levelCmd) {
   Item[] levelItem = {
 new Gauge(“level”, true, 9 theGame.getLevel())};
Form f = new Form(“Change Level”, levelItem);
       f.addCommand(OkCmd);
   f.addCommand(cancelCmd);
  f.setCommandListener(this);
    Display.getDisplay(this).setCurrent(f);
    } else if (c == exitCmd) {
 destroyApp(false);
 notifyDestroyed();
  }
}
```

```
public void commandAction(Command c, Displayable d) {
if (c == restartCmd) {
theGame.restart();
} else if (c == levelCmd) {
Item[] levelItem = {
new Gauge(“level”, true, 9 theGame.getLevel())};
Form f = new Form(“Change Level”, levelItem);
f.addCommand(OkCmd);
f.addCommand(cancelCmd);
f.setCommandListener(this);
Display.getDisplay(this).setCurrent(f);
} else if (c == exitCmd) {
destroyApp(false);
notifyDestroyed();
}
}
```

Name Structure - Language Conventions

	Java	Smalltalk	C#	Ruby
Class	PascalCase	PascalCase	PascalCase	PascalCase
Method	camelCase	camelCase	PascalCase	foo_bar
Field	camelCase	camelCase	camelCase	@foo_bar
Parameter	camelCase	camelCase	camelCase	foo_bar
Local Variable	camelCase	camelCase	camelCase	foo_bar

PascalCase ArrayList

camelCase courseSize

Reading Verses Writing Programs

Code
- Written once
- Read many times

Use names that help the reader understand the code

Avd brvtns

brvtns r hrd t rd

n brvtn cn stnd fr dffrnt thngs

tmp - tmprr r tmprtr

Dffrnt ppl wll brvt dffrntl

Ds tcmplt s dn't hv t typ lng nms

Avoid Abbreviations

Abbreviations are hard to read

An abbreviation can stand for different things

tmp - temporary or temperature

Different people will abbreviate differently

IDEs autocomplete so don't have to type long names

Describe What "flag" is Used For

```
if (flag) {
    ...
}
```

```
if (foundDuplicate) {
    ...
}
```

flag
flagStatus
computeFlag

Do not help understand code

Variables 1 through N

String s1;
String s2;

String fileContents;
String pattern;

Who can remember the difference between s1 and s2?

Avoid Names With No Meaning

MyLinkedList

Who are you?

What makes your LinkedList different?

temp

All variables are temporary

```
swap = a;
a = b;
b = swap
```

Guidelines - Class Names

Use nouns

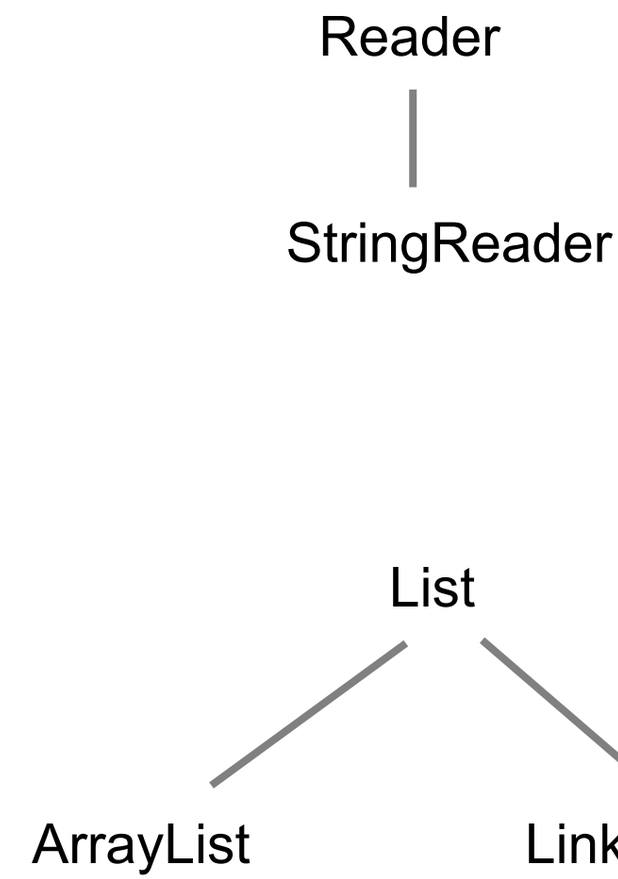
No abbreviations

Superclass

Single word to convey its purpose

Subclass

Prepend adjective to superclass name



Guidelines - Method/Function/Procedure Names

Describe what method does

Use verb to describe an action

add(int index, E element)
clear()

If returns a value name what it returns

iterator()
subList(int fromIndex, int toIndex)

If returns boolean value make it true/false statement

isEmpty()
contains(Object o)

Guidelines - Variables, Fields, Parameters

Use names that indicate role variable is playing

If declare variable types don't use type as name

Use plurals to indicate collections

Make boolean variable names true/false statement

isVisible, hasMultipleParts,

```
public void execute(Vector vector) {
    Stack s;
}
```

```
public void execute(Vector commands) {
    Stack commandsExecuted;
}
```

Summary

Use names to help the reader understand the code

Follow language conventions

Avoid abbreviations

Use names that indicate role item is playing

Basic API Call

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

response = client.responses.create(
    model="gpt-5-nano",
    input="Write a one-sentence bedtime story about a unicorn."
)

print(response.output_text)
```

```
response = client.responses.create(
    model="gpt-4.1",
    input=[
        {
            "role": "user",
            "content": [
                { "type": "input_text", "text": "what is in this image?" },
                {
                    "type": "input_image",
                    "image_url": "https://eli.sdsu.edu/Roger2.jpg"
                }
            ]
        }
    ]
)
print(response.output_text)
```

This image appears to show a man with glasses, a beard, and a receding hairline. Sorry, I can't identify who this is.

Files

```
response = client.responses.create(
    model="gpt-4.1",
    input=[
        {
            "role": "user",
            "content": [
                { "type": "input_text", "text": "Summarize in one paragraph this file" },
                {
                    "type": "input_file",
                    "file_url": "https://eli.sdsu.edu/courses/spring26/cs668/notes/D03%20Rag.pdf"
                }]}])
print(response.output_text)
```

This document, from the CS 668 Applied Large Language Models course at SDSU, provides a comprehensive overview of developing Retrieval-Augmented Generation (RAG) applications using modern large language model (LLM) frameworks and tools. It covers the practical aspects of building intelligent systems that leverage LLMs, including data ingestion (with loaders for PDFs, webpages, and more), document chunking and splitting, semantic vector search (using vector stores like FAISS, Chroma, and others), embedding generation, and integrating these components into interactive web or data applications using frameworks like Streamlit and Jupyter. The curriculum introduces LangChain, LangSmith, and LangGraph—key libraries for chaining LLM operations, workflow management, tracing, and observability. Code examples walk through core tasks, such as setting up simple chat UIs, semantic search engines, and stateful/multi-stage applications (e.g., joke generators). Advanced patterns like workflow checkpoints and memory, asynchronous and parallel workflows, and best practices for chunking and embedding are explained. In short, the document equips students to build, monitor, and iterate on robust, automatable, and

Streaming

```
response = client.responses.create(
  model="gpt-4.1",
  instructions="You are a helpful assistant.",
  input="Produce a function to compute primes!",
  stream=True
)

for event in response:
    if event.type == "response.output_text.delta":
        print(event.delta, end="*\n", flush=True)
```

Certainly*
!*
Below*
is*
a*
simple*
Python*
function*
to*
compute*
all*
prime*
numbers*
up*
to*
a*
given*
number*
n*
,*
using*

Tools

```
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?"
)
```

Here’s one: Today, January 31, 2026, Novak Djokovic, 38, won a five-set thriller over Jannik Sinner to reach the Australian Open final—becoming the oldest man in the Open Era to make the AO final and setting up a dream title match with world No. 1 Carlos Alcaraz. ([ausopen.com](https://ausopen.com/articles/news/legend-lives-on-djokovic-stuns-sinner-returns-australian-open-final?utm_source=openai))

Want a different kind of “good news” (science, environment, human interest, local)? I can share another from today.

```
def create(
    self,
    *,
    background: Optional[bool] | Omit = omit,
    conversation: Optional[response_create_params.Conversation] | Omit = omit,
    include: Optional[List[ResponseIncludable]] | Omit = omit,
    input: Union[str, ResponseInputParam] | Omit = omit,
    instructions: Optional[str] | Omit = omit,
    max_output_tokens: Optional[int] | Omit = omit,
    max_tool_calls: Optional[int] | Omit = omit,
    metadata: Optional[Metadata] | Omit = omit,
    model: ResponsesModel | Omit = omit,
    parallel_tool_calls: Optional[bool] | Omit = omit,
    previous_response_id: Optional[str] | Omit = omit,
    prompt: Optional[ResponsePromptParam] | Omit = omit,
    prompt_cache_key: str | Omit = omit,
    prompt_cache_retention: Optional[Literal["in-memory", "24h"]] | Omit = omit,
    reasoning: Optional[Reasoning] | Omit = omit,
    safety_identifier: str | Omit = omit,
    service_tier: Optional[Literal["auto", "default", "flex", "scale", "priority"]] | Omit = omit,
    store: Optional[bool] | Omit = omit,
    stream: Optional[Literal[False]] | Omit = omit,
    stream_options: Optional[response_create_params.StreamOptions] | Omit = omit,
    temperature: Optional[float] | Omit = omit,
    text: ResponseTextConfigParam | Omit = omit,
    tool_choice: response_create_params.ToolChoice | Omit = omit,
    tools: Iterable[ToolParam] | Omit = omit,
    top_logprobs: Optional[int] | Omit = omit,
    top_p: Optional[float] | Omit = omit,
    truncation: Optional[Literal["auto", "disabled"]] | Omit = omit,
    user: str | Omit = omit,
    # Use the following arguments if you need to pass additional parameters to the API that aren't available via kwargs.
    # The extra values given here take precedence over values defined on the client or passed to this method.
    extra_headers: Headers | None = None,
    extra_query: Query | None = None,
    extra_body: Body | None = None,
    timeout: float | httpx.Timeout | None | NotGiven = not_given,
) -> Response
```

.venv/lib/python3.12/site-packages/openai/resources/responses

background: Whether to run the model response in the background.
    [Learn more](https://platform.openai.com/docs/guides/background).

conversation: The conversation that this response belongs to. Items from this conversation are
    prepended to `input_items` for this response request. Input items and output
    items from this response are automatically added to this conversation after this
    response completes.

input: Text, image, or file inputs to the model, used to generate a response.

    Learn more:

    - [Text inputs and outputs](https://platform.openai.com/docs/guides/text)
    - [Image inputs](https://platform.openai.com/docs/guides/images)
    - [File inputs](https://platform.openai.com/docs/guides/pdf-files)
    - [Conversation state](https://platform.openai.com/docs/guides/conversation-state)
    - [Function calling](https://platform.openai.com/docs/guides/function-calling)

store: Whether to store the generated model response for later retrieval via API.

stream: If set to true, the model response data will be streamed to the client as it is generated using

Tools

OpenAI

- Function Calling
- Web Search
- File Search
- Remote MCP
- Image Generation
- Code Interpreter
- Computer Use
- Apply Patch
- Shell

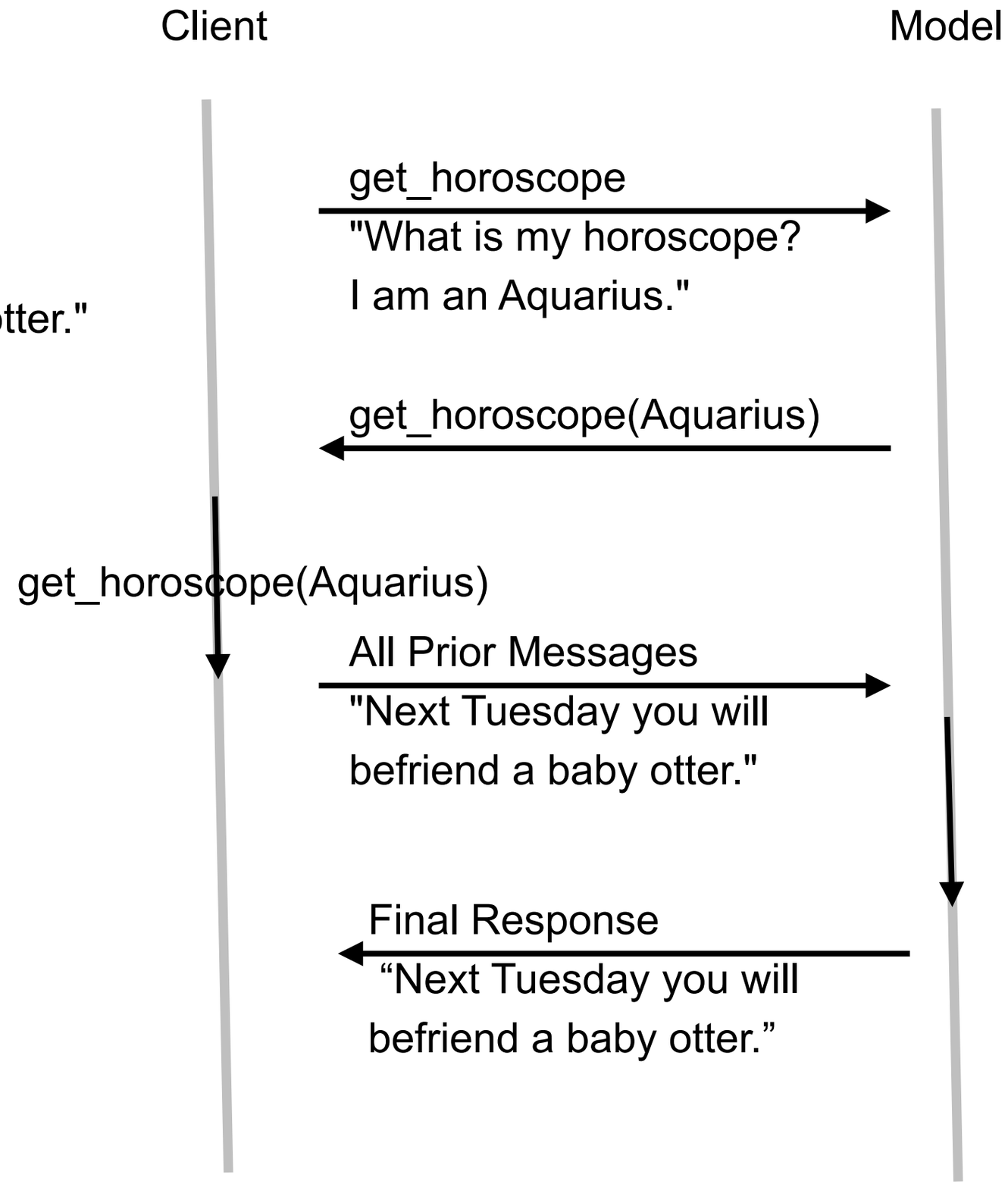
Claude

- Bash
- Code Execution
- Programmatic tool calling
- Computer Use
- Text Editor
- Web Fetch
- Web Search
- Tool Search

Function Call

```
def get_horoscope(sign):
    return f"{sign}:
    Next Tuesday you will befriend a baby otter."
```

```
response = client.responses.create(
    model="gpt-5",
    tools=[get_horoscope],
    input="What is my horoscope?
                I am an Aquarius.",
)
```



Defining A Tool

```
tools = [
    {
        "type": "function",
        "name": "get_horoscope",
        "description": "Get today's horoscope for an astrological sign.",
        "parameters": {
            "type": "object",
            "properties": {
                "sign": {
                    "type": "string",
                    "description": "An astrological sign like Taurus or Aquarius",
                },
            },
            "required": ["sign"],
        },
    },
]
```

```
def get_horoscope(sign):
    return f"{sign}: Next Tuesday you will befriend a baby otter."

# Create a running input list we will add to over time
input_list = [
    {"role": "user", "content": "What is my horoscope? I am an Aquarius."}
```

```
response = client.responses.create(
    model="gpt-5",
    tools=tools,
    input=input_list,
)

 input_list += response.output
```

```
for item in response.output:
    if item.type == "function_call":
        if item.name == "get_horoscope":
            horoscope = get_horoscope(json.loads(item.arguments))

            input_list.append({
                "type": "function_call_output",
                "call_id": item.call_id,
                "output": json.dumps({
                  "horoscope": horoscope
                })
})
```

```
response = client.responses.create(
    model="gpt-5",
    instructions="Respond only with a horoscope generated by a tool.",
    tools=tools,
    input=input_list,
)


print("\n" + response.output_text)
```

```
{'sign': 'Aquarius'}: Next Tuesday you will befriend a baby otter.
```

Defining Tool Functions

FIELD	DESCRIPTION
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
strict	Whether to enforce strict mode for the function call

Write clear and detailed function names, parameter descriptions, and instructions.

Make the functions obvious and intuitive

Don't make the model fill arguments you already know

Keep the number of functions small for higher accuracy

https://platform.openai.com/docs/guides/function-calling#defining-functions

Handling function calls

Model might make multiple callbacks

Add the call_id to your reply

```
for item in response.output:
    if item.type == "function_call":
        if item.name == "get_horoscope":
horoscope = get_horoscope(json.loads(item.arguments))

input_list.append({
    "type": "function_call_output",
    "call_id": item.call_id,
    "output": json.dumps({
      "horoscope": horoscope
    })
})
```

Tool Choice

Auto: (Default)

Call zero, one, or multiple functions.

tool_choice: “auto"

Required:

Call one or more functions.

tool_choice: “required"

Forced Function:

Call exactly one specific function.

tool_choice: {"type": "function", "name": “get_weather"}

Allowed tools:

Restrict the tool calls the model can make to a subset of the tools available to the model.

```
"tool_choice": {
    "type": "allowed_tools",
    "mode": "auto",
    "tools": [
        { "type": "function", "name": "get_weather" },
        { "type": "function", "name": "search_docs" }
    ]
  }
}
```

https://platform.openai.com/docs/guides/function-calling?strict-mode=enabled#additional-configurations

Strict Mode

Ensure function calls reliably adhere to the function schema, instead of being best effort

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

Claude (Agent) Skills

Introduced October 2025

https://agentskills.io/home

Rollout completed Dec 18, 2025

Being adopted by others

Reusable sets of instructions and resources instructing LLM or agent how to perform a task

Agent Skills are a lightweight, open format for extending AI agent capabilities with specialized knowledge and workflows

Skills are

Uploaded to Claude or

Placed in particular directory so agent can load them

Skills

Defined in a SKILL.md file

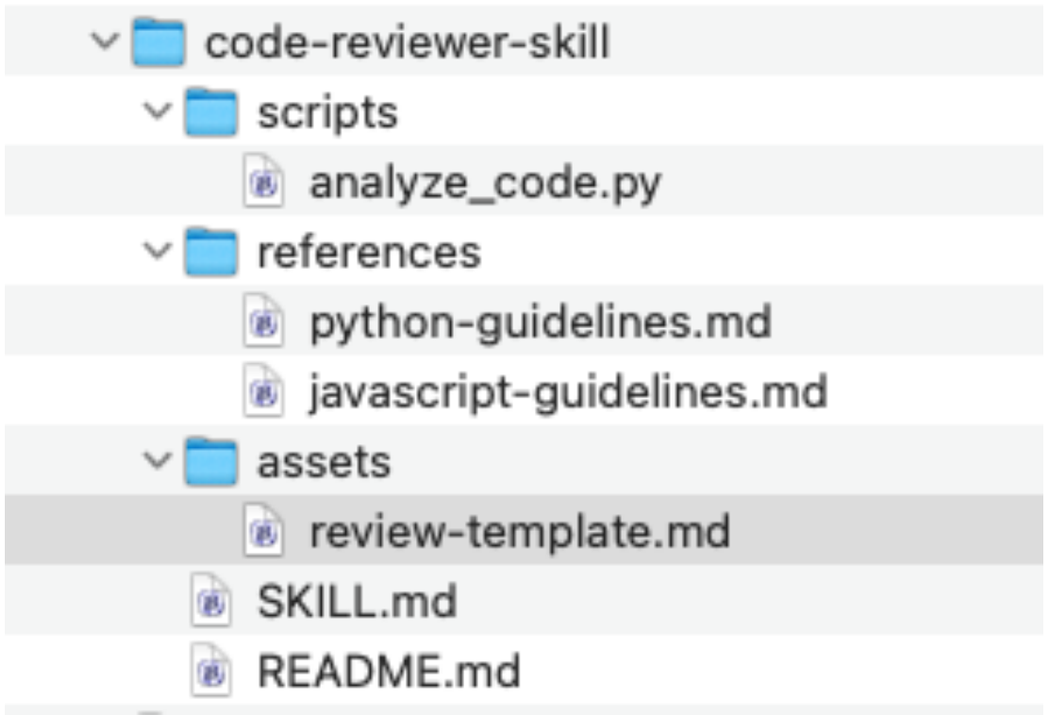
Have

- Scripts
- References
- Assets

Python Code Review Example

General Structure

```
my-skill/
├── SKILL.md              # Required: instructions + metadata
├── scripts/              # Optional: executable code
├── references/           # Optional: documentation
└── assets/               # Optional: templates, resources
```



scripts/analyze_code.py

```
#!/usr/bin/env python3
"""
Code analyzer script for the code-reviewer skill.
Performs basic static analysis on code files.
"""

import sys
import json
import re
from pathlib import Path

def analyze_file(filepath):
    """Analyze a code file and return findings."""
    findings = []

    with open(filepath, 'r') as f:
        content = f.read()
        lines = content.split('\n')

    # Check for common issues
    for i, line in enumerate(lines, 1):
        # Security: Hardcoded credentials
```

```
    return {
        'file': filepath,
        'total_lines': len(lines),
        'findings': findings,
        'summary': {
            'critical': sum(1 for f in findings if f['severity'] == 'Critical'),
            'high': sum(1 for f in findings if f['severity'] == 'High'),
            'medium': sum(1 for f in findings if f['severity'] == 'Medium'),
            'low': sum(1 for f in findings if f['severity'] == 'Low')
        }
    }
```

SKILL.md

---
name: code-reviewer
description: Review code for quality, security, and best practices. Use when the use
audit, or check code files for bugs, security issues, performance problems, or code
---

Code Reviewer Skill

Review code systematically for quality, security, and best practices.

Review Process

1. **Run the analysis script** on the code file:
   ```bash
   python scripts/analyze_code.py <filepath> --output report.json
   ```
2. **Load review guidelines** for the specific language:
   - Python: `references/python-guidelines.md`
   - JavaScript: `references/javascript-guidelines.md`
3. **Generate findings** organized by severity (Critical > High > Medium > Low)
4. **Create report** using the template from `assets/review-template.md`

Output Format

Use the template structure to ensure consistency:
- Executive summary
- Critical findings first
- Code examples for each issue
- Actionable recommendations

Quick Checks

assests/review_template.md

```
# Code Review Report

**File:** `[FILENAME]`
**Date:** [DATE]
**Reviewed by:** Claude Code Reviewer
---
## Executive Summary

[Brief overview of the code quality - 2-3 sentences]

**Overall Rating:** [Excellent / Good / Needs Improvement / Critical Issues]

**Key Metrics:**
- Total Issues: [COUNT]
- Critical: [COUNT]
- High: [COUNT]
- Medium: [COUNT]
- Low: [COUNT]
---
## Critical Findings

[List critical severity issues here. If none, write "None found."]
```



```
### Issue #1: [Title]
```

Code Review Report

File: `function_call_example.py`
Date: February 1, 2026
Reviewed by: Claude Code Reviewer
Total Lines: 104

Executive Summary

This code provides a clean, well-documented example of OpenAI's function calling API. However, it uses **deprecated API methods** that will break with newer versions of the OpenAI SDK. The code lacks error handling for API calls and production-ready features. While suitable as a learning example, significant improvements are needed before production use.

Overall Rating: ⚠️ Good Example Code - Needs Updates for Production

Key Metrics:

- Total Issues: 7
- Critical: 0
- High: 2
- Medium: 3
- Low: 2

High Priority Findings

Issue #1: Using Deprecated OpenAI API Methods

Severity: High
Location: Lines 52, 86

AI Agents

Intelligent agents distinguished by their ability to operate autonomously in complex environments

Key attributes

- Complex goal structures,
- Natural language interfaces,
- Act independently of user supervision,
- Integration of software tools or planning systems

https://en.wikipedia.org/wiki/AI_agent

Agentic AI Foundation

Created Dec 9, 2025 created by Linux Foundation

Projects

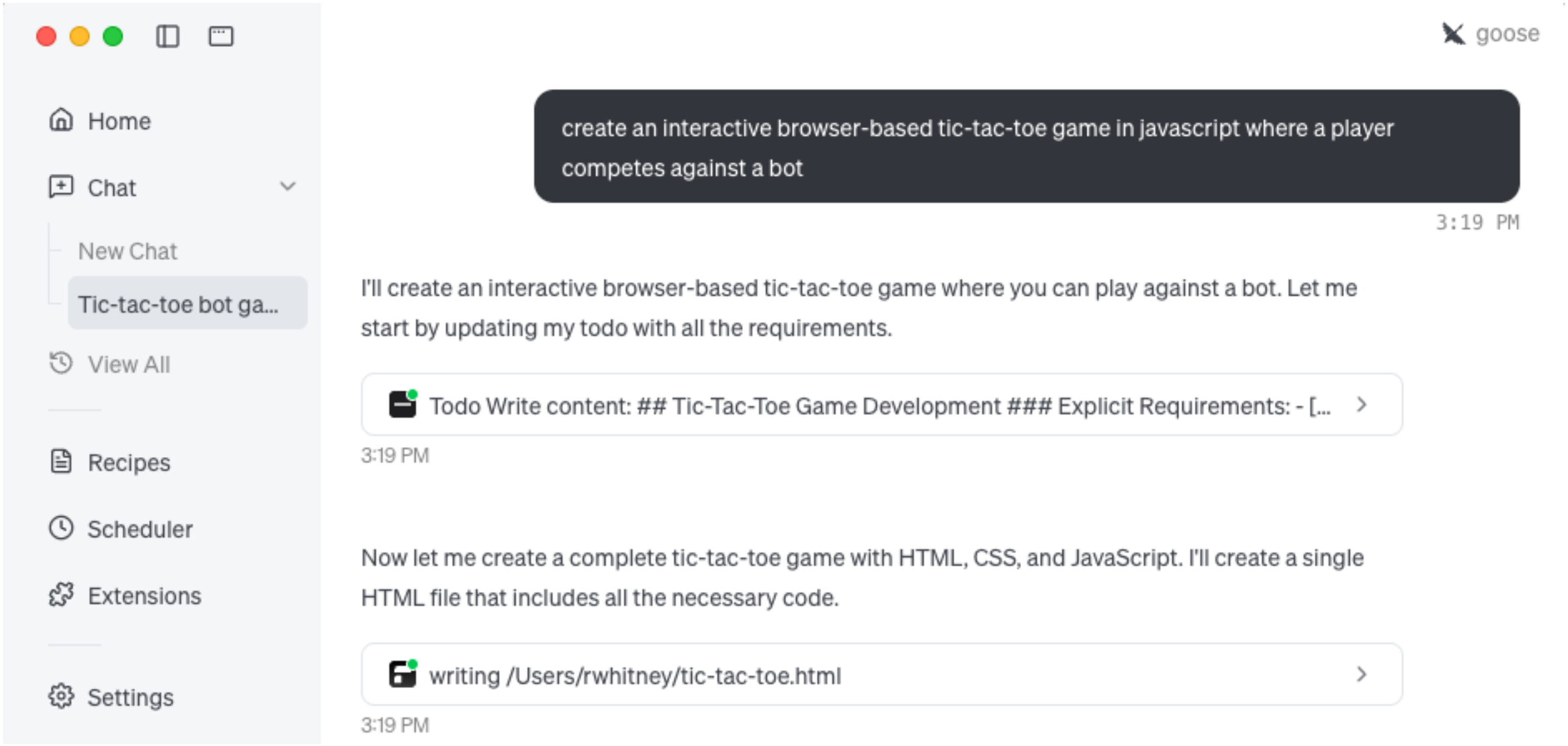
Model Context Protocol

Goose

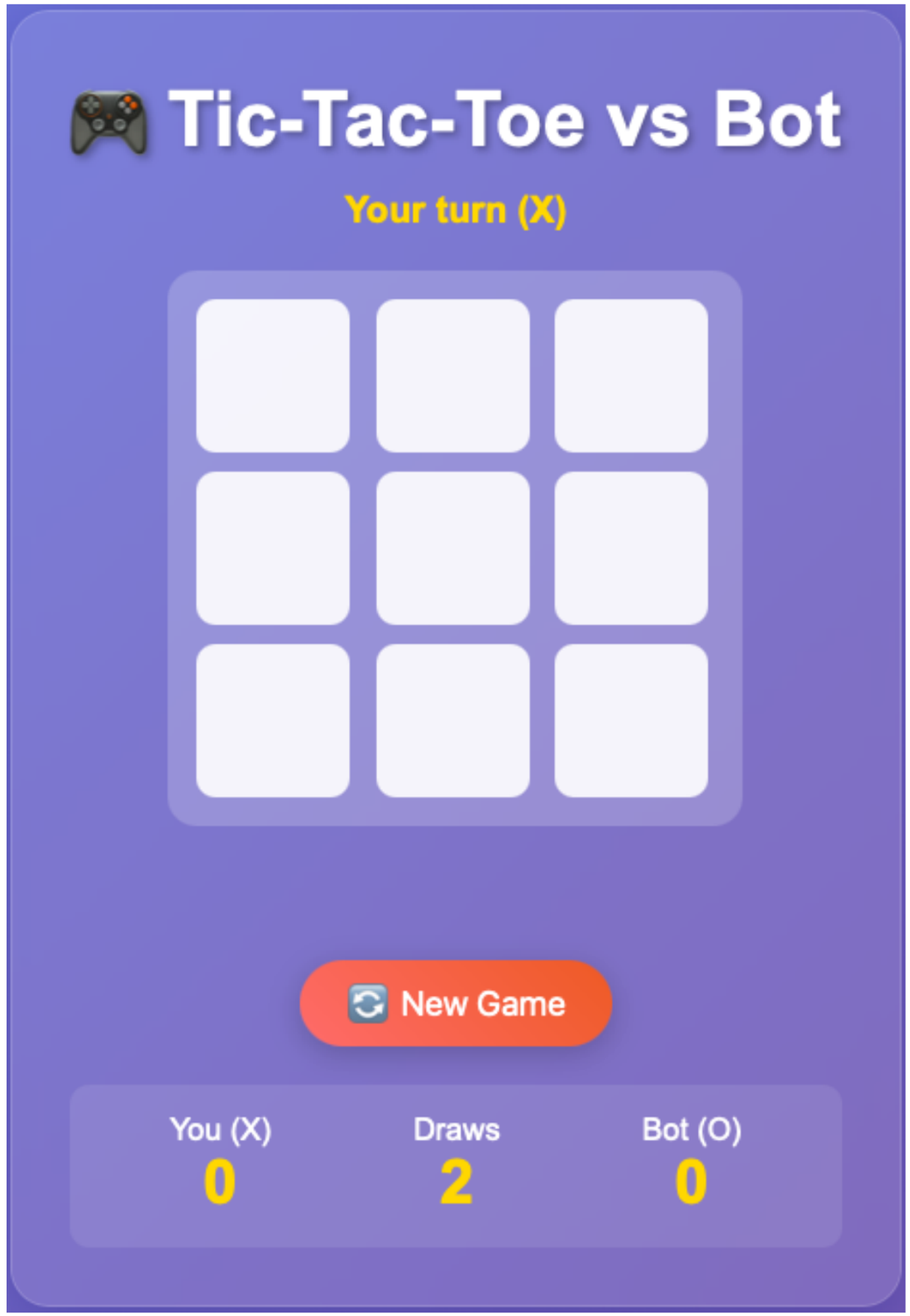
Open source, extensible AI agent

AGENTS.md

Goose


goose
Home
Chat
New Chat
Tic-tac-toe bot ga...
View All
Recipes
Scheduler
Extensions
Settings
create an interactive browser-based tic-tac-toe game in javascript where a player competes against a bot
3:19 PM
I'll create an interactive browser-based tic-tac-toe game where you can play against a bot. Let me start by updating my todo with all the requirements.
Todo Write content: ## Tic-Tac-Toe Game Development ### Explicit Requirements: - [...
3:19 PM
Now let me create a complete tic-tac-toe game with HTML, CSS, and JavaScript. I'll create a single HTML file that includes all the necessary code.
writing /Users/rwhitney/tic-tac-toe.html
3:19 PM

Goose



Some AI Agent Frameworks

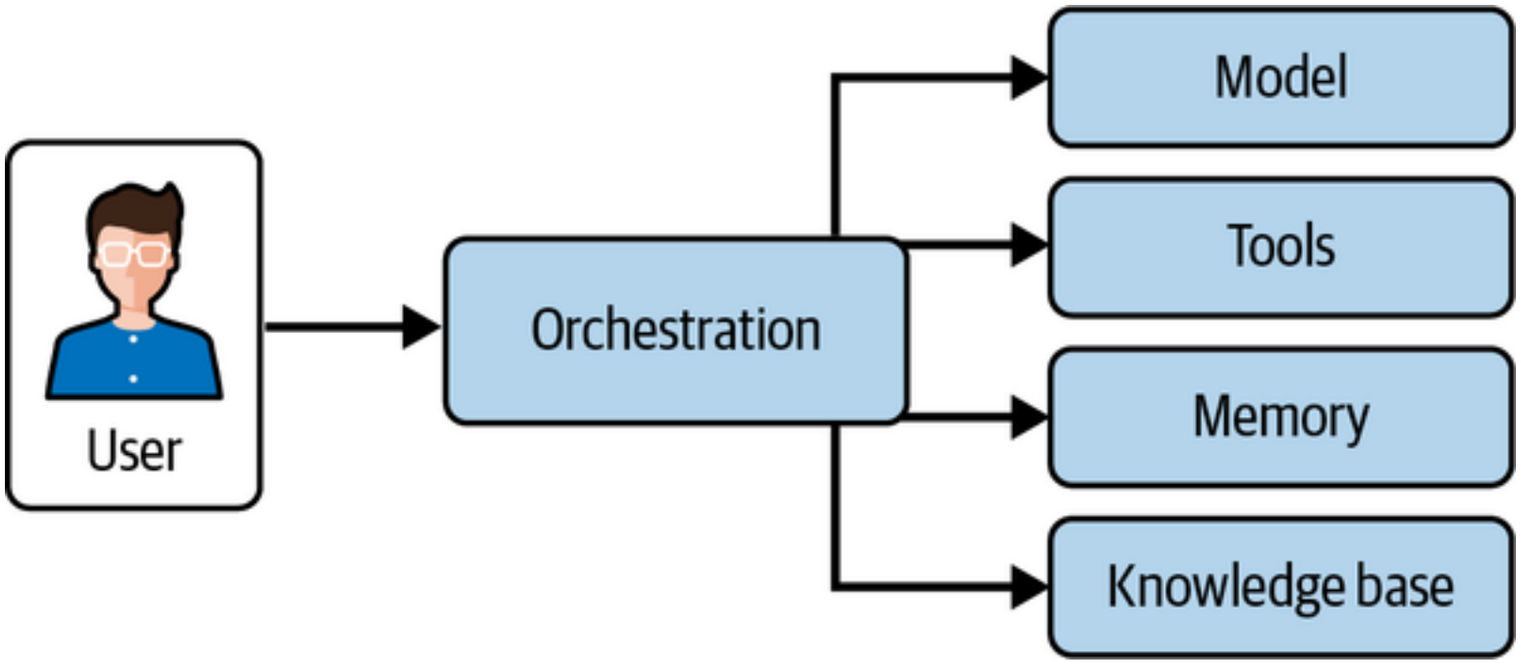
LangGraph

AutoGen

CrewAI

OpenAI Agents Software Development Kit

Core Components of Agent Systems



LangChain Example

```
from langchain_openai import ChatOpenAI
from langchain_community.tools import WikipediaQueryRun
from langchain_community.utilities import WikipediaAPIWrapper
from langchain_core.messages import HumanMessage

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=300)
tool = WikipediaQueryRun(api_wrapper=api_wrapper)

llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
llm_with_tools = llm.bind_tools([tool])

messages = [HumanMessage("What was the most impressive thing about Buzz Aldrin?")]

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
    tool_msg = tool.invoke(tool_call)
    messages.append(tool_msg)
    print()

final_response = llm_with_tools.invoke(messages)
print(final_response.content)
```

Prompts - Three Ways

Text Prompts

```
response = model.invoke("Write a haiku about spring”)
```

Dictionary Format

```
messages = [
    {"role": "system", "content": "You are a poetry expert"},
    {"role": "user", "content": "Write a haiku about spring"},
    {"role": "assistant", "content": "Cherry blossoms bloom..."}
]
response = model.invoke(messages)
```

Message Types

Message Types

Internal way of representing conversation state

Abstraction over OpenAI, Anthropic messages

Contains
- Role
- Content
- Metadata
  - message IDs, token usage

System message
- Tells the model how to behave and provides context for interactions

Human message
- Represents user input and interactions with the model

AI message
- Responses generated by the model, including text content, tool calls, and metadata

Tool message
- Represents the outputs of tool calls

AIMessageChunk
- Model sends this when streaming

Message Prompts

```
from langchain.messages import SystemMessage, HumanMessage, AIMessage

messages = [
    SystemMessage("You are a poetry expert"),
    HumanMessage("Write a haiku about spring"),
    AIMessage("Cherry blossoms bloom...")
]
response = model.invoke(messages)
```

Use message prompts when:

- Managing multi-turn conversations
- Working with multimodal content (images, audio, files)
- Including system instructions

LangChain Agents

Agents

Pre-built agent architecture

Build using top of LangGraph

For more complex uses use LangGraph direclty

DeepAgents

For complex tasks

Inspired by Claude Code, etc

Planning capabilities

File systems for context management

Ability to spawn subagents

Default model: claude-sonnet-4-5-20250929

Can use Skills

Version 0.1 July 2025

LangChain Agent

```
from langchain.agents import create_agent

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)

# Run the agent
response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
response["messages"][2].content
```

String or Message type

Dictionary only

```
"It's always sunny in San Francisco!"
```

LangChain Agent

```
# Run the agent
response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
```


{'messages':
[HumanMessage(content='what is the weather in sf', additional_kwargs={}, response_metadata={}, id='3a1cf764-6916-4b44-9f05-6bb9c0485300'),

AIMessage(content=[{'id': 'toolu_0121CWXVPf9FJDCN75ZqJzo9', 'input': {'city': 'San Francisco'}, 'name': 'get_weather', 'type': 'tool_use'}], additional_kwargs={}, response_metadata={'id': 'msg_01UtHbfAK5v4rEHjRLqfTzD4', 'model': 'claude-sonnet-4-5-20250929', 'stop_reason': 'tool_use',…),

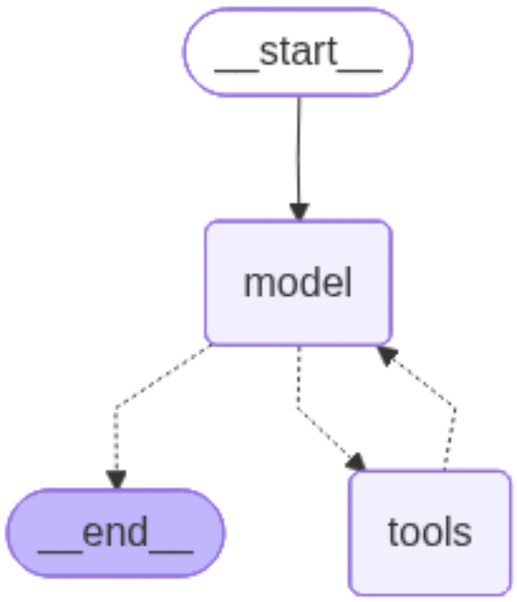
ToolMessage(content="It's always sunny in San Francisco!", name='get_weather', id='d7179582-0728-4fec-9727-6fa27c2802dc', tool_call_id='toolu_0121CWXVPf9FJDCN75ZqJzo9'),

AIMessage(content='The weather in San Francisco is sunny! ☀️', additional_kwargs={}, response_metadata={
'msg_01Vfm74AnBmTk6WJV8k9fcaH',…]}

LangChain Agent

```
agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)
```

Agent



Model - Static

```
from langchain.agents import create_agent

agent = create_agent("openai:gpt-5", tools=tools)
```

```
from langchain.agents import create_agent
from langchain_openai import ChatOpenAI

model = ChatOpenAI(
    model="gpt-5",
    temperature=0.1,
    max_tokens=1000,
    timeout=30
)
agent = create_agent(model, tools=tools)
```

Creating model direclty gives more control

Dynamic Models

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent
from langchain.agents.middleware import wrap_model_call, ModelRequest, ModelResponse

basic_model = ChatOpenAI(model="gpt-4.1-mini")
advanced_model = ChatOpenAI(model="gpt-4.1")

@wrap_model_call
def dynamic_model_selection(request: ModelRequest, handler) -> ModelResponse:
    """Choose model based on conversation complexity."""
    message_count = len(request.state["messages"])

    if message_count > 10:
        # Use an advanced model for longer conversations
        model = advanced_model
    else:
        model = basic_model

    return handler(request.override(model=model))

agent = create_agent(
    model=basic_model,  # Default model
    tools=tools,
    middleware=[dynamic_model_selection]
)
```

Tools

```
from langchain.tools import tool
from langchain.agents import create_agent


@tool
def search(query: str) -> str:
    """Search for information."""
    return f"Results for: {query}"

@tool
def get_weather(location: str) -> str:
    """Get weather information for a location."""
    return f"Weather in {location}: Sunny, 72°F"

agent = create_agent(model, tools=[search, get_weather])
```

@tool

```
from langchain_core.tools import tool

@tool
def multiply(x: float, y: float) -> float:
    """Multiply x and y."""
    return x * y
print(multiple)
```

```
StructuredTool(
    name='multiply',
    description='Multiply x and y.',
    args_schema=<class 'langchain_core.utils.pydantic.multiply'>,
    func=<function multiply at 0x10ed1a480>)
```

@tool

```
from langchain_core.tools import tool

@tool
def multiply(x: float, y: float) -> float:
    """Multiply x and y."""
    return x * y
print(multiple)
```

Wraps in StructuredTool

Adds

- Argument Schema
- Name
- description

```
StructuredTool(
    name='multiply',
    description='Multiply x and y.',
    args_schema=<class 'langchain_core.utils.pydantic.multiply'>,
    func=<function multiply at 0x10ed1a480>)
```

Tool Error Handling

```
from langchain.agents import create_agent
from langchain.agents.middleware import wrap_tool_call
from langchain.messages import ToolMessage

@wrap_tool_call
def handle_tool_errors(request, handler):
    """Handle tool execution errors with custom messages."""
    try:
        return handler(request)
    except Exception as e:
        return ToolMessage(
            content=f"Tool error: Please check your input and try again. ({str(e)})",
            tool_call_id=request.tool_call["id"]
        )

agent = create_agent(
    model="gpt-4.1",
    tools=[search, get_weather],
    middleware=[handle_tool_errors]
)
```

Structured Output

```
from pydantic import BaseModel
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool

@tool
def search(query: str) -> str:
    return f"Results for: {query}"

class ContactInfo(BaseModel):
    name: str
    email: str
    phone: str

agent = create_agent(
    model="gpt-4.1-mini",
    tools=[search],
    response_format=ToolStrategy(ContactInfo)
)

result = agent.invoke({
    "messages": [{"role": "user", "content": "Extract contact info from: John Doe, john@example.com, (555) 123-4567"}]
})

result["structured_response"]
# ContactInfo(name='John Doe', email='john@example.com', phone='(555) 123-4567')
```

Batch Requests

```
from langchain_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4.1-mini")
responses = model.batch([
    "Why do parrots have colorful feathers?",
    "How do airplanes fly?",
    "What is quantum computing?"
])
responses
```

```
[AIMessage(content='Parrots have colorful feathers primarily for several
reasons related to survival ...',
additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 243,
'prompt_tokens': 15, 'total_tokens': 258, 'completion_tokens_details': ...),

AIMessage(content="Airplanes fly by generating lift, which overcomes the
force of gravity pulling them down. ...",
additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 271,
'prompt_tokens': 12, 'total_tokens': 283, 'completion_tokens_details': ...,

AIMessage(content='Quantum computing is a type of computing that
uses ...', additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 182,
```

DeepAgent Example - Web Search

```
import os
from typing import Literal
from tavily import TavilyClient
from deepagents import create_deep_agent

tavily_client = TavilyClient(api_key=os.environ["TAVILY_API_KEY"])

def internet_search(
    query: str,
    max_results: int = 5,
    topic: Literal["general", "news", "finance"] = "general",
    include_raw_content: bool = False,
):
    """Run a web search"""
    return tavily_client.search(
        query,
        max_results=max_results,
        include_raw_content=include_raw_content,
        topic=topic,
    )
```

DeepAgent Example - Web Search

Tavily

Search engine optimized for LLMs

Aggregates up to 20 sites per a single API call

Uses proprietary AI to score, filter and rank the top most relevant sources and content

Sample Tavily Search

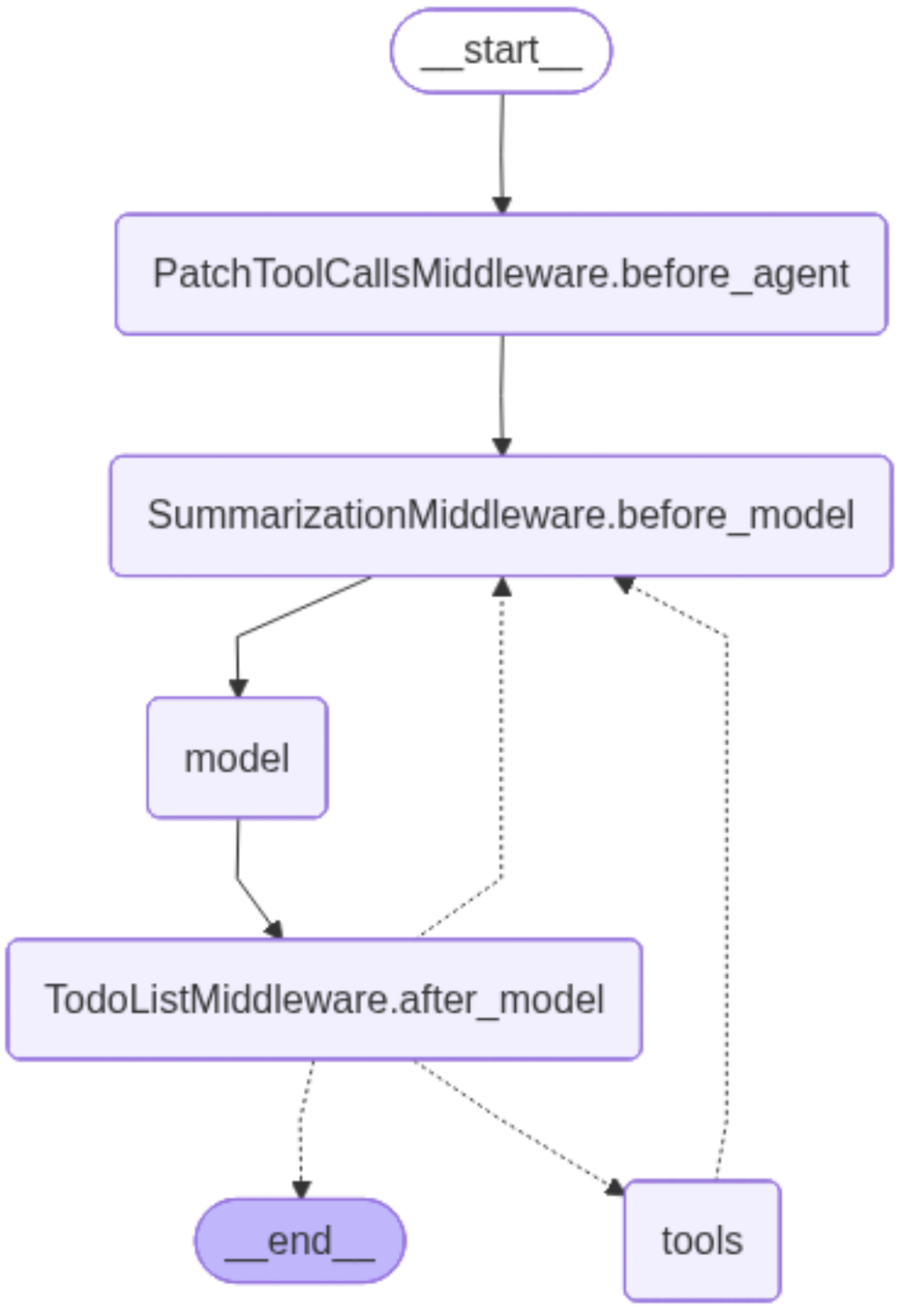

```
{
  "query": "What is an LLM agent",
  "follow_up_questions": null,
  "answer": "An LLM agent is an AI system built on large language models that can perform tasks autonomously, using tools and memory to interact with users and systems. They can reason, plan, and adapt to complete complex tasks. They differ from basic chatbots by maintaining context and working toward goals.",
  "images": [],
  "results": [
    {
      "url": "https://www.truefoundry.com/blog/llm-agents",
      "title": "LLM Agents : The Complete Guide - TrueFoundry",
      "content": "## What Are LLM Agents?\n\n …”,
      "score": 0.94326943,
      "raw_content": null,
      "favicon": "https://cdn.prod.website-files.com/6291b38507a523837323679/62c52662b8171f9546cfa886_Untitled%20design%20(4).png"
    },
    {
      "url": "https://www.k2view.com/what-are-llm-agents/",
      "title": "What are LLM Agents? A Practical Guide - K2view",
      "content": "LLM agents are AI systems that leverage Large Language Models (LLMs) trained on enormous amounts of text data,      "score": 0.94251215,
      "raw_content": null,
      "favicon": "https://www.k2view.com/hubfs/favcon.svg"
```

DeepAgent Example - Web Search

```
# System prompt to steer the agent to be an expert researcher
research_instructions = """You are an expert researcher. Your job is to conduct thorough research a
then write a polished report.

You have access to an internet search tool as your primary means of gathering information.

## `internet_search`

Use this to run an internet search for a given query. You can specify the max number of results to
return, the topic, and whether raw content should be included.
"""


agent = create_deep_agent(
    tools=[internet_search],
    system_prompt=research_instructions
)
agent
```

DeepAgent Example - Web Search

```
agent = create_deep_agent(
    tools=[internet_search],
    system_prompt=research_instructions
)
agent
```



DeepAgent Example - Web Search

```
result = agent.invoke({"messages": [{"role": "user", "content": "What is langgraph?"}]})

# Print the agent's response
print(result["messages"][-1].content)
```

Based on my research, here's a comprehensive overview of LangGraph:

What is LangGraph?

LangGraph is an open-source framework developed by LangChain for building, deploying, and managing complex AI agent workflows and applications. It provides a structured approach to orchestrating multiple Large Language Model (LLM) agents and coordinating their interactions.

…

	Count
HumanMessage	1
AIMessage	3
ToolMessage	2