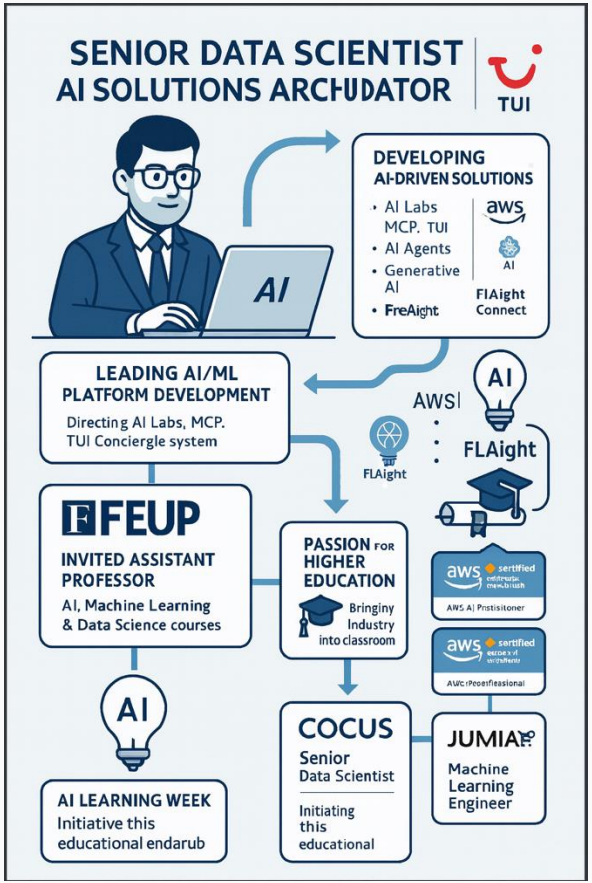
BUILDING AND DEPLOYING AI AGENTS

Class 5 - AI Agents in Production

Who the heck is Luis Dias?

Meet the man behing the glasses



luis.f.s.m.dias@gmail.com

https://www.linkedin.com/in/luisfilipedias/

Today's Roadmap

We will focus on popular technologies to make it happen

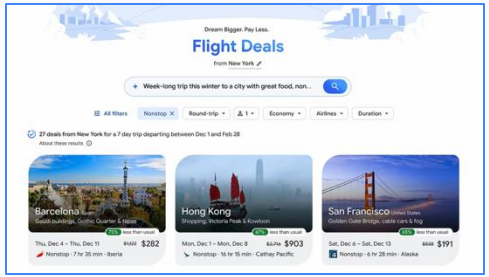
1. **Agents in Production –** Understanding the moving components.
2. **Streamlit -** Will be used to build the user interface (UI).
3. **Streamlit cloud deployment -** Remote deployment environments for our Streamlit UI.
4. **FastAPI -** Python framework that abstracts the complexity of developing an API.
5. **Render -** Remote deployment environments for our API.
6. **Langfuse -** To set some standard monitoring from the beginning.
7. **Reflection -** Connecting the dots of what we learned today.
8. **Assignment -** Hands-on exercise to consolidate today's learnings.
9. **Wrap up -** Next steps

AI Agents in production

Agents in Production

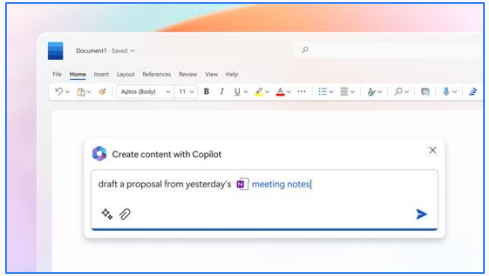
Since the rise of ChatGPT in 2022 we have seen many new AI Applications using AI Agents

Google flight deals

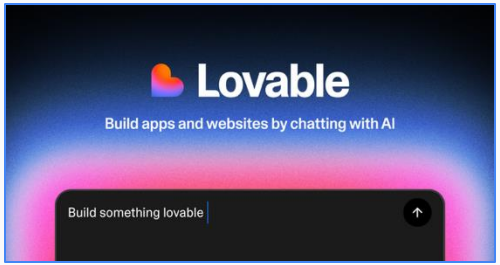


Claude Code

ChatGPT



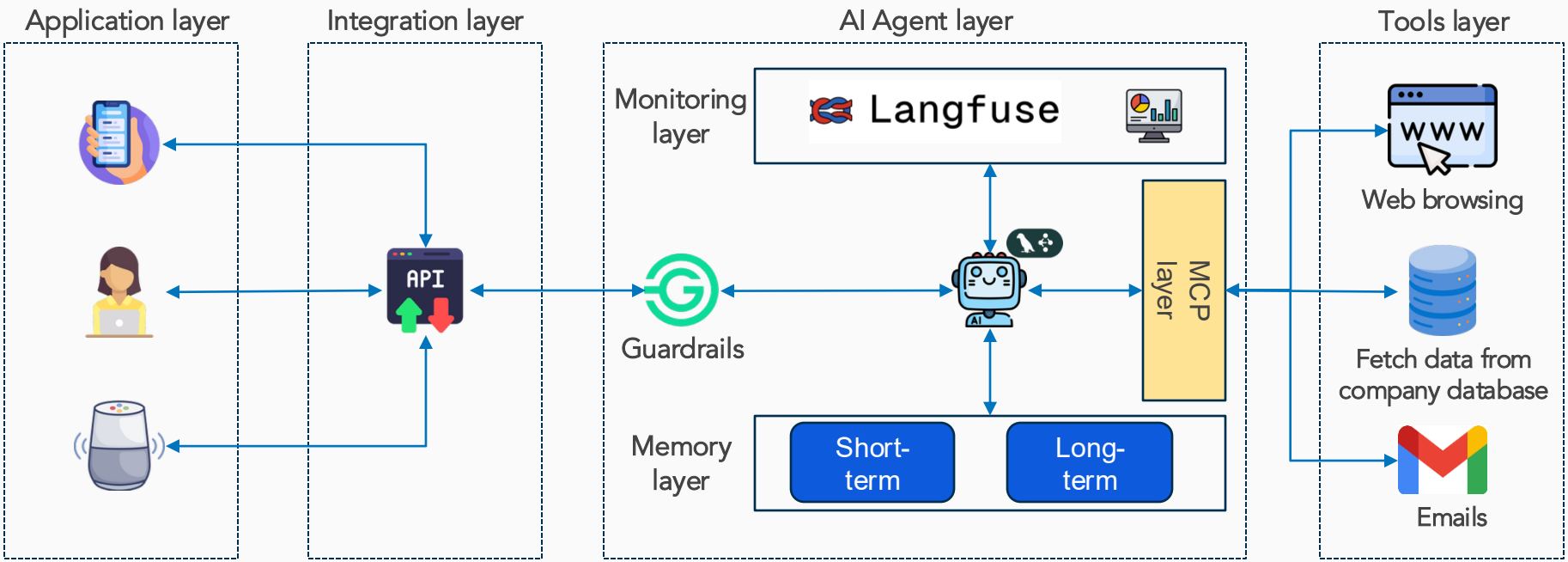
Microsoft Copilot



Lovable

What Does an Agent in Operations Look Like?

High-level overview of the main components of an AI Agent ready for production



Where the interaction is started and maintained

Allows the information to flow between the Agent and customer

Contains the core logic for the AI Agents, including Guardrails to protect against prompt injection, the Agent's decision-making logic, as well as data and monitoring systems.

Potential tools for AI Agents to leverage

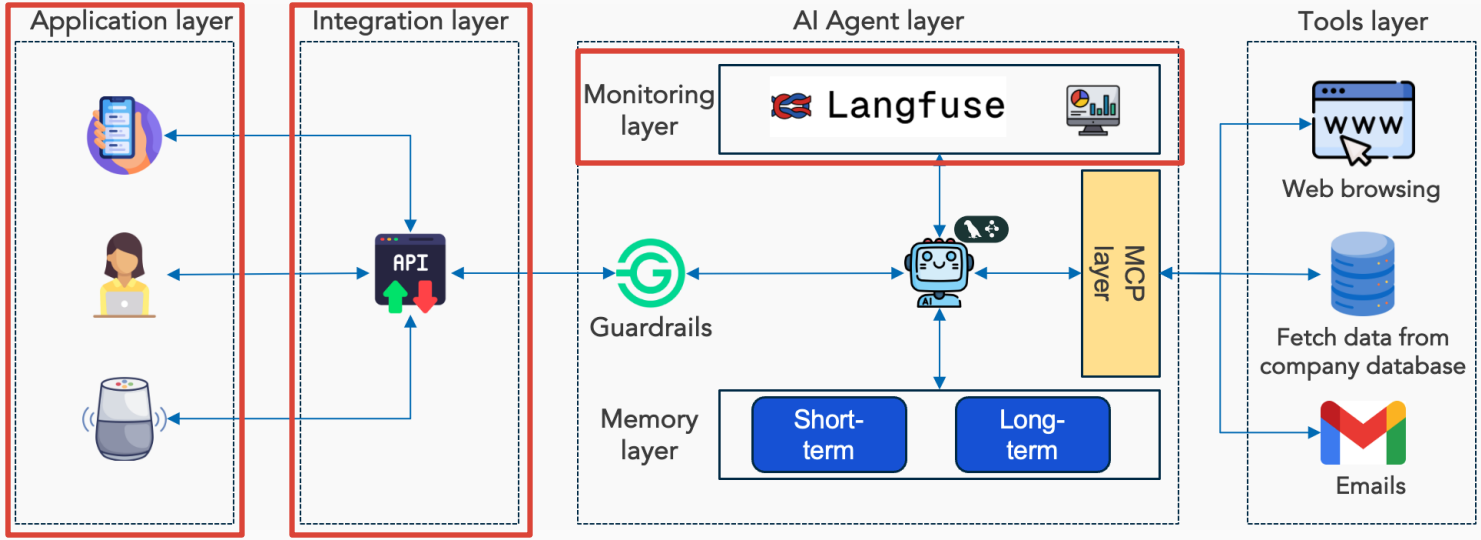
Why Deployment Matters

Agents are not useful if they stay in a notebook.

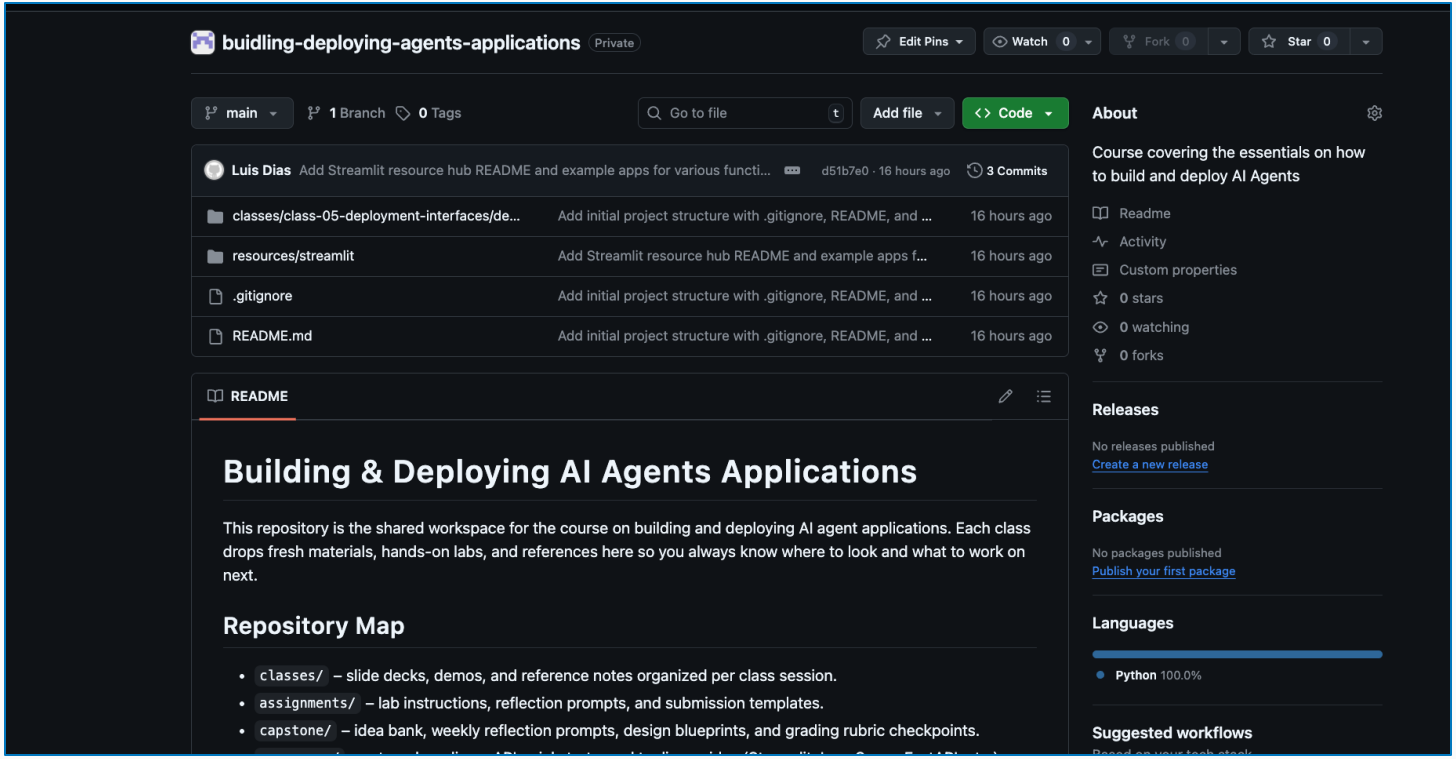
Deployment = making agents usable by people and systems. This means:

- **UI-first**: easy access for end users
- **API-first**: scalable, reusable, workflow integration
- **Monitoring**: track performance and health

Today we will tackle three layers of an AI Agent in Production



All class demos and examples are accessible in the provided Github repo



https://github.com/diaxz12/BUILDING-AND-DEPLOYING-AI-AGENTS---Part-2/tree/main/assignments/class-05-deployment-lab

Streamlit

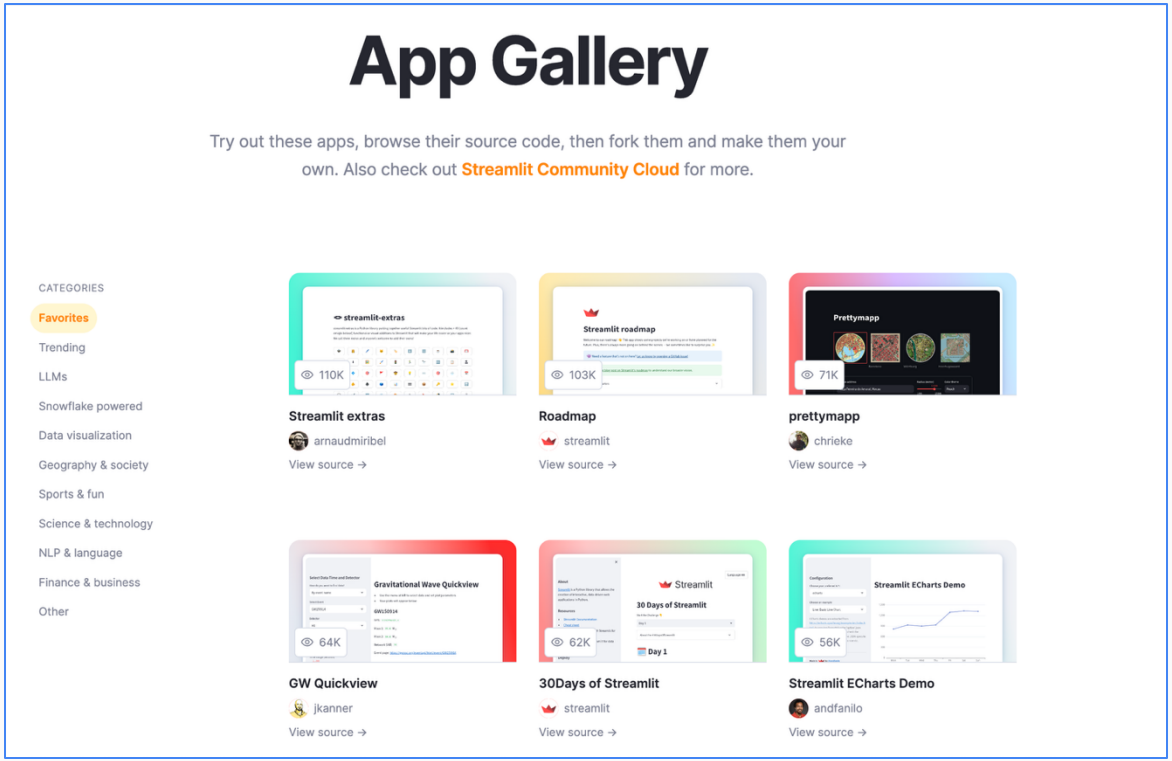
The king of UI in Python for non frontend developers

Why use streamlit?

- Ideal for supporting Data Science, Machine Learning and AI projects
- Enables creation of interactive interfaces
- Designed for beginners – no front-end skills required
- With available widgets and elements, web pages can be built with just a few lines of code
- Compatible with most Python libraries

Streamlit application examples

You can find many examples here: https://streamlit.io/gallery



How to start?

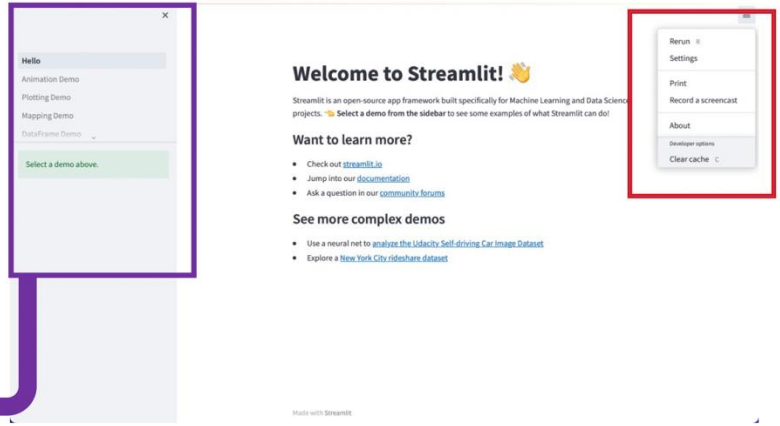
Install the Streamlit library using 'pip install streamlit'. To test the installation, you can run the command 'streamlit hello'

```
(streamlitTutorial) (base) ➜  streamlitTutorial streamlit hello

  Welcome to Streamlit. Check out our demo in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.1.89:8501

  Ready to create your own Python apps super quickly?
  Head over to https://docs.streamlit.io

  May you create awesome apps!
```

Command to start Streamlit

URL to reach the web server at port 8501

Hamburger menu



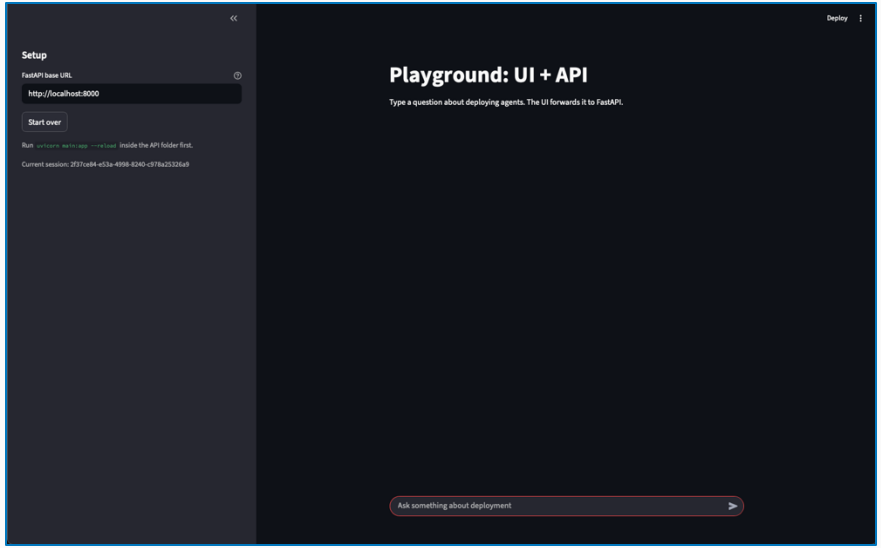
Sidebar with access to sample demos

Using the ‘streamlit run’

Class 5 streamlit-chat-ui demo

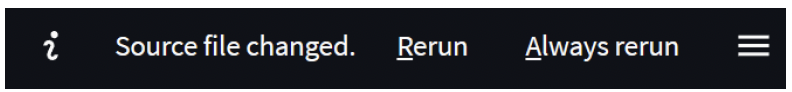
```
.venv ~/Desktop/buidling-deploying-agents-applications/classes/class-05-deployment-interfaces/demos/streamlit-chat-ui git:(main)
streamlit run app.py

  You can now view your Streamlit app in your browser.

  Local URL: http://localhost:8501
  Network URL: http://192.168.1.25:8501

  For better performance, install the Watchdog module:

  $ xcode-select --install
  $ pip install watchdog
```

To run a Streamlit application, use the command ’streamlit run app.py’ in your terminal, where app.py (example) is the Python script containing the Streamlit code.



During development

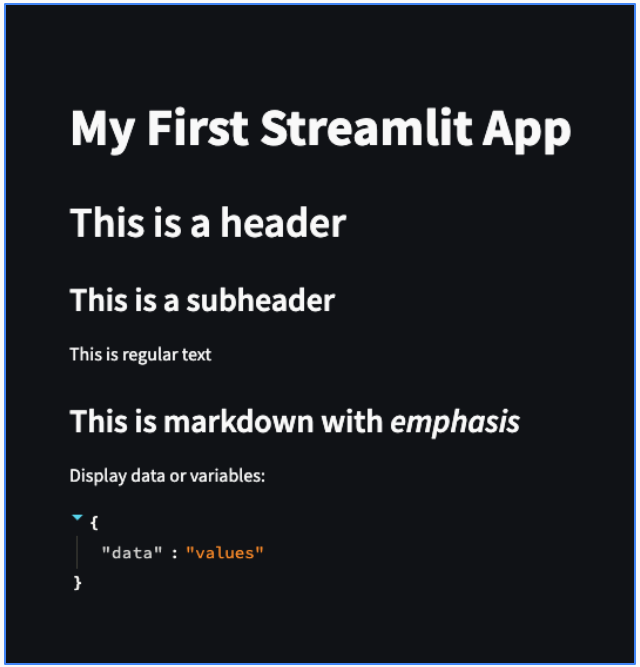
- Whenever the Python script is saved, the application can be refreshed by clicking **'Rerun'**, without needing to restart the server.
- By choosing **'Always rerun'**, the app updates automatically with each save, allowing you to see changes immediately.
- Whenever something needs to be updated on the screen (including user interactions), Streamlit re-runs the script from top to bottom.
- You can stop Streamlit with the command **'CTRL+C'**.



Streamlit basics

Text and titles

```python
import streamlit as st

# Add a title
st.title('My First Streamlit App')

# Add header
st.header('This is a header')

# Add subheader
st.subheader('This is a subheader')

# Add normal text
st.text('This is regular text')

# Add markdown
st.markdown('### This is markdown with *emphasis*')

# Display a Python object's information
st.write('Display data or variables:', {'data': 'values'})
```



Streamlit basics

Work with data

```python
import streamlit as st
import pandas as pd
import numpy as np

# Create sample data
df = pd.DataFrame({
    'Name': ['John', 'Mary', 'Bob', 'Jane'],
    'Age': [25, 30, 22, 28],
    'City': ['New York', 'Boston', 'Chicago', 'Seattle']
})

# Display the dataframe
st.write('### Sample Data:')
st.dataframe(df)

# Display static table
st.table(df)

# Display statistics
st.write('### Data Statistics:')
st.write(df.describe())
```

Sample Data:

	Name	Age	City
0	John	25	New York
1	Mary	30	Boston
2	Bob	22	Chicago
3	Jane	28	Seattle

	Name	Age	City
0	John	25	New York
1	Mary	30	Boston
2	Bob	22	Chicago
3	Jane	28	Seattle

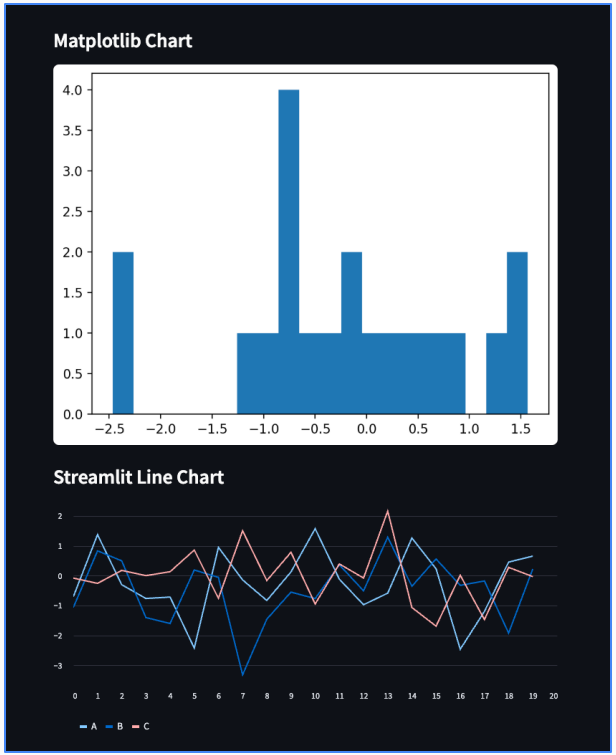
Data Statistics:

	Age
count	4
mean	26.25
std	3.5
min	22
25%	24.25
50%	26.5
75%	28.5
max	30

Streamlit basics

Build plots

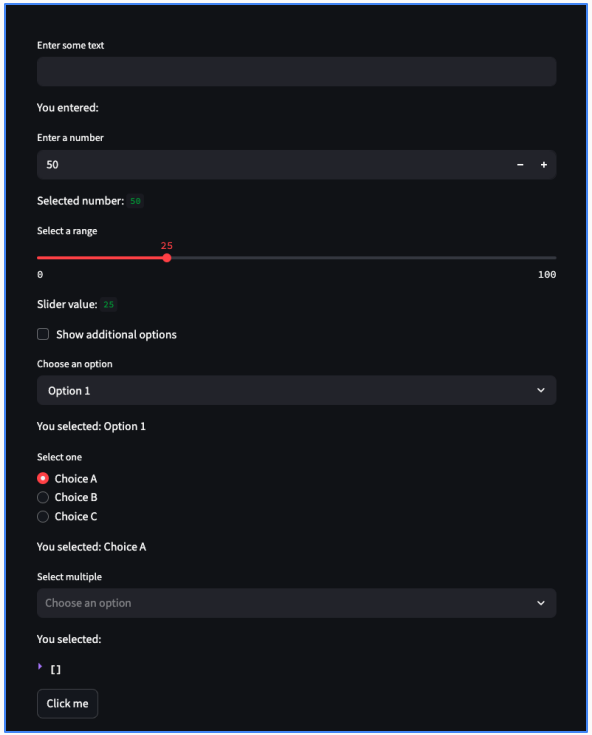
```python
import streamlit as st
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px

# Create sample data
chart_data = pd.DataFrame(np.random.randn(20, 3), columns=['A', 'B', 'C'])

# Matplotlib chart
st.write('### Matplotlib Chart')
fig, ax = plt.subplots()
ax.hist(chart_data['A'], bins=20)
st.pyplot(fig)

# Native Streamlit line chart
st.write('### Streamlit Line Chart')
st.line_chart(chart_data)

# Native Streamlit area chart
st.write('### Streamlit Area Chart')
st.area_chart(chart_data)

# Native Streamlit bar chart
st.write('### Streamlit Bar Chart')
st.bar_chart(chart_data)

# Plotly chart (more interactive)
st.write('### Plotly Chart')
fig = px.scatter(chart_data, x='A', y='B', color='C', size='C')
st.plotly_chart(fig)
```



Streamlit basics

Widgets

```python
import streamlit as st

# Text input
user_input = st.text_input("Enter some text")
st.write('You entered:', user_input)

# Number input
number = st.number_input("Enter a number", min_value=0, max_value=100, value=50)
st.write('Selected number:', number)

# Slider
slider_value = st.slider("Select a range", 0, 100, 25)
st.write('Slider value:', slider_value)

# Checkbox
if st.checkbox("Show additional options"):
    st.write("You selected to show additional options!")

# Selectbox
option = st.selectbox("Choose an option", ["Option 1", "Option 2", "Option 3"])
st.write('You selected:', option)

# Radio buttons
radio_option = st.radio("Select one", ["Choice A", "Choice B", "Choice C"])
st.write('You selected:', radio_option)

# Multiselect
multi_options = st.multiselect("Select multiple", ["Item 1", "Item 2", "Item 3", "Item 4"])
st.write('You selected:', multi_options)

# Button
if st.button("Click me"):
    st.write("Button clicked!")
```



Streamlit basics

Caching to improve the app response time

```python
import streamlit as st
import pandas as pd
import time

# Caching data loading
@st.cache_data
def load_large_dataset():
    time.sleep(2)  # Simulate long loading time
    return pd.DataFrame({
        'A': range(1000),
        'B': range(1000, 2000)
    })

data = load_large_dataset()
st.write('Loaded data:', data.head())
```

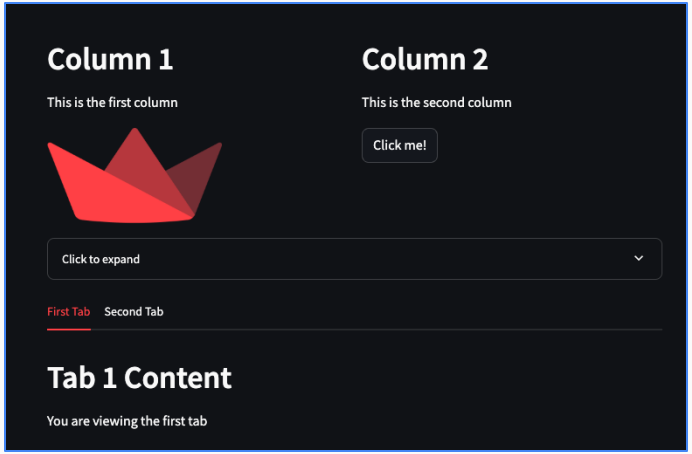
Loaded data:

	A	B
0	0	1000
1	1	1001
2	2	1002
3	3	1003
4	4	1004

Streamlit basics

Layout options

```python
import streamlit as st

# Create columns
col1, col2 = st.columns(2)

with col1:
    st.header("Column 1")
    st.write("This is the first column")
    st.image("https://streamlit.io/images/brand/streamlit-mark-color.png", width=200)

with col2:
    st.header("Column 2")
    st.write("This is the second column")
    if st.button("Click me!"):
        st.write("Thanks for clicking!")

# Create expandable sections
with st.expander("Click to expand"):
    st.write("This content is hidden until expanded")
    st.bar_chart({"data": [1, 5, 2, 6, 2, 1]})

# Create tabs
tab1, tab2 = st.tabs(["First Tab", "Second Tab"])

with tab1:
    st.header("Tab 1 Content")
    st.write("You are viewing the first tab")

with tab2:
    st.header("Tab 2 Content")
    st.write("You are viewing the second tab")
```



Streamlit demo

Class Streamlit demo

Today we will deploy a Chat UI in the Streamlit Community cloud

buidling-deploying-agents-applications / classes / class-05-deployment-interfaces / demos / **streamlit-chat-ui** /

Add file

Luis Dias · Add initial project structure with .gitignore, README, and demo files... · 423cd4e · 17 hours ago · History

Name	Last commit message	Last commit date
..		
README.md	Add initial project structure with .gitignore, README, and demo files...	17 hours ago
app.py	Add initial project structure with .gitignore, README, and demo files...	17 hours ago
requirements.txt	Add initial project structure with .gitignore, README, and demo files...	17 hours ago

README.md

Streamlit Chat UI

A beginner-friendly front-end for the Class 5 agent API. Everything is in one file (`app.py`) so you can trace the flow from input → request → response.

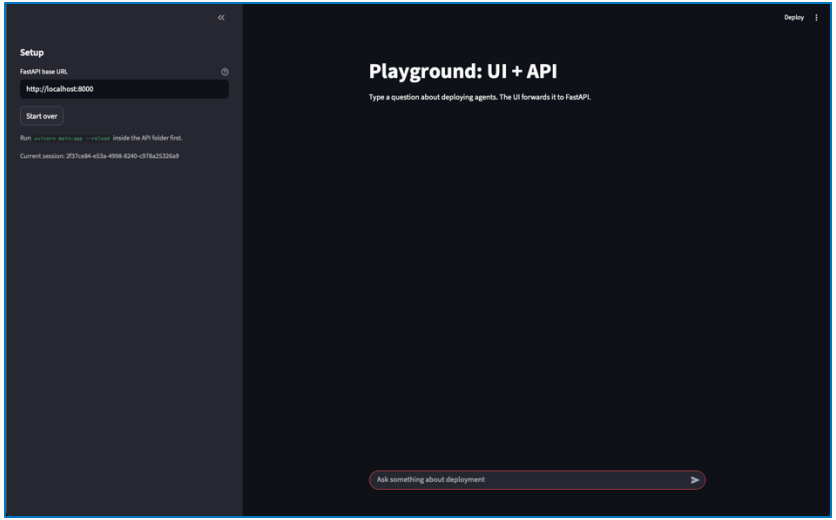
Before you launch the UI

- Make sure the FastAPI demo is already running on `http://127.0.0.1:8000` (or a deployed URL).
- Stay inside this folder: `classes/class-05-deployment-interfaces/demos/streamlit-chat-ui` .
- Python 3.10+ needs to be installed on your machine.

Step-by-step: run Streamlit locally

1. **Create a virtual environment** (skip if you reuse the one from the API):
   - macOS / Linux:

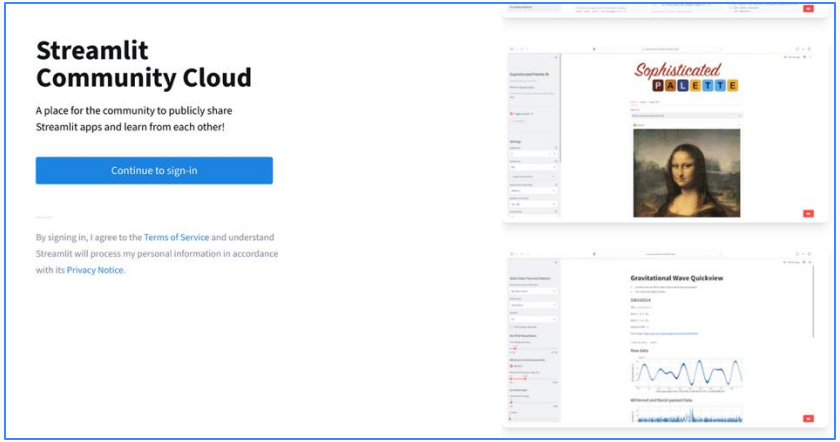
```
python3 -m venv .venv
```



Link to demo: https://github.com/diaxz12/BUILDING-AND-DEPLOYING-AI-AGENTS---Part-2/tree/main/classes/class-05-deployment-interfaces/demos/streamlit-chat-ui

Make the UI available to everyone

- **To share your application with others, Streamlit Community Cloud offers free hosting:**
- Push your code to a GitHub repository
- Visit https://share.streamlit.io/
- Connect your GitHub account
- Select your repository and main Python file
- Deploy your application



Want to learn more about streamlit?

The Streamlit demo was just a quick intro. I advise to go through a dedicated tutorial of Streamlit!

Official documentation:

- Streamlit Documentation - Full usage details
- Streamlit GitHub Repository - Streamlit source code

Tutoriais:

- Streamlit Community Forum - Official Streamlit forum
- 30 Days of Streamlit - 30-day course with challenges
- Streamlit YouTube Channel - Tutorials and video demos

Streamlit App examples:

- Streamlit Gallery - encompasses community built app
- GitHub Example Repositories - open-source examples

FastAPI

Modern, fast (high-performance), web framework for building APIs with Python

What is an API?

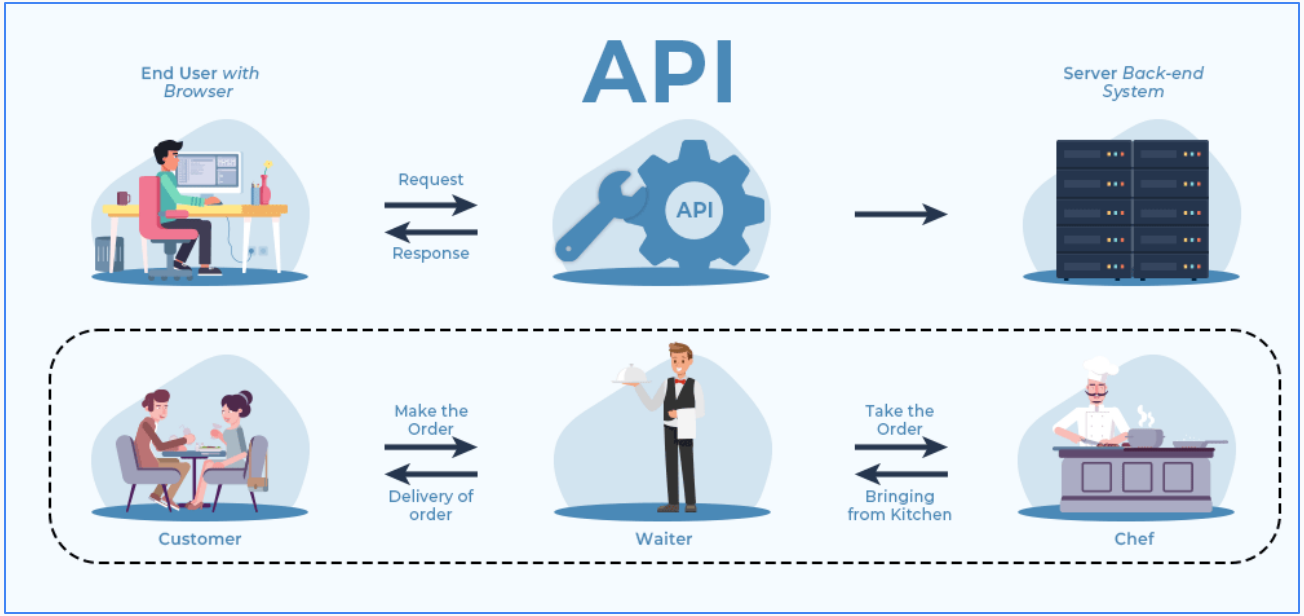
Like APIs themselves, API calls vary according to the specifications outlined in the API documentation. However, an API call follows three basic steps:

1. **The API client initiates the API call, or request for information.** The API client must format the request according to the protocol and schema provided by the API endpoint.
2. **The API endpoint receives the request.** The API endpoint then authenticates the API client and validates the API schema. This helps ensure that a) the call is coming from a verified source, and b) the conditions of the request have been met.
3. **The API endpoint returns the requested information to the API client.** The API schema determines the type of responses that may be returned to the client.

For a more in-depth explanation of API calls, read What is an API call?

What Is an API (High-Level)?

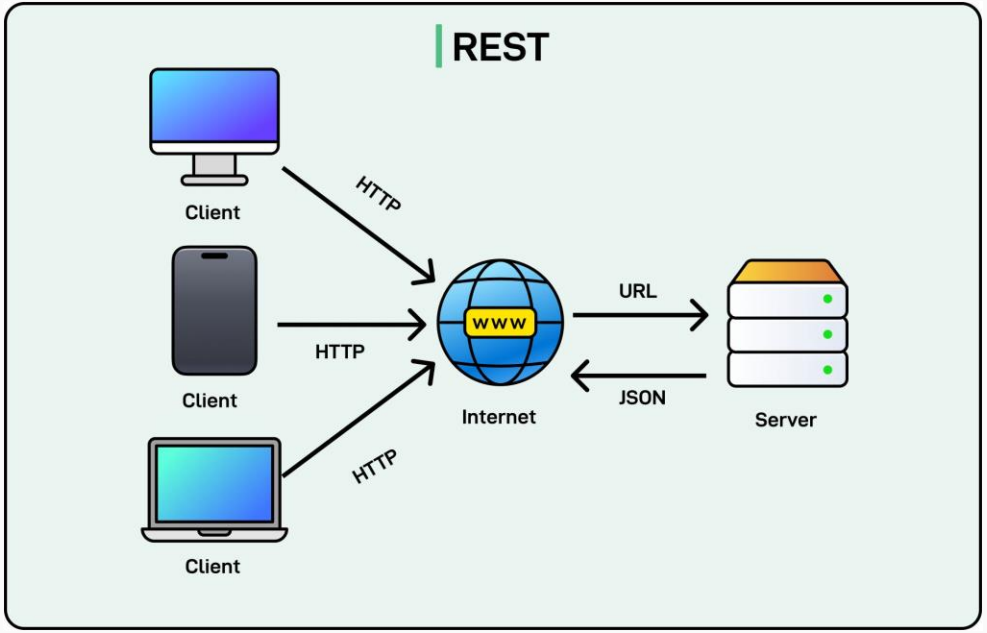
- API = Application Programming Interface.
- Like a “contract” → defines how systems talk to each other.
- Used to integrate agents into apps, websites, workflows.



Let's just review some important concepts

What Is an API Client?

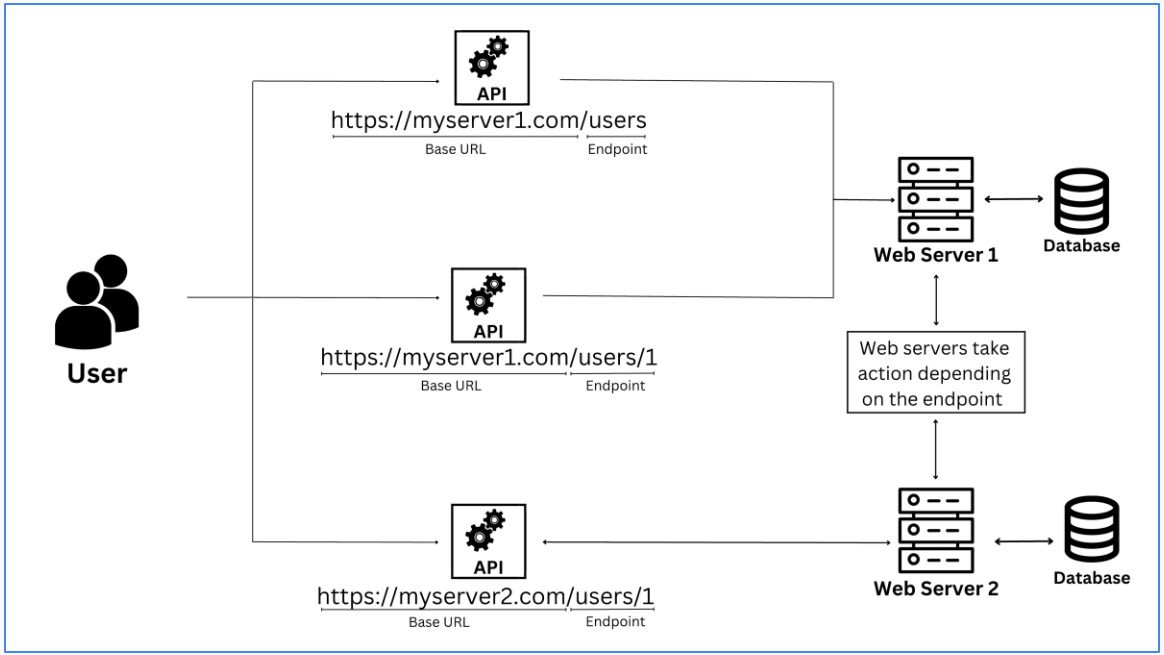
- An **API** defines rules about how two programs talk to each other.
- An **API client** is the program that follows those rules to send requests (like "give me user data") and process responses (like "here's the user data in JSON").



Example: when you open a weather app, the app is the client, it calls a weather API to fetch today's forecast.

What is an API endpoint?

- An **API endpoint** is a specific URL (address) inside the API where you can access or send data.
- It usually corresponds to a **resource** (like /users, /orders, /products).
- Each endpoint can support different **methods**:
  - GET: fetch data
  - POST: create new data
  - PUT/PATCH: update data
  - DELETE: remove data



Example: /chat endpoint = system sends text → agent replies.

Calling API endpoints in Python

Example

In Python, we often use libraries like requests or httpx to act as an API client:

- The Python script = API client.
- The weather API = the server.
- The JSON result = the server's response.

example.py

```
import requests


response = requests.get(
"https://api.open-meteo.com/v1/forecast
?latitude=40.4&longitude=-3.7&
current_weather=true"
)

print(response.json())
```

FastAPI Basics

Overview

- Lightweight framework for building APIs.
- Define endpoints (/, /chat).
- Comes with automatic docs (/docs).
- *Follow FASTAPI example:* https://fastapi.tiangolo.com/pt/tutorial/
- We will focus only on:
    1. First steps
    2. Path parameters
    3. Query parameters
    4. Request body

Install FastAPI

Class 5 fastapi-agent-service demo

Make sure you create a virtual environment, activate it, and then **install FastAPI**:

```bash
$ pip install "fastapi[standard]"

100%
```

restart ↻

First steps

The simplest FastAPI file could look like this:

Copy the example to file named ‘main.py’ and run the server with ”fastapi dev main.py”. Use the provided API Server URL to GET the ”Hello World” message.

example.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
  return {"message": "Hello World"}
```

bash

```
$ fastapi dev main.py

  FastAPI   Starting development server 🚀

            Searching for package file structure from directories
            with __init__.py files
            Importing from /home/user/code/awesomeapp

   module   🐍 main.py

     code   Importing the FastAPI app object from the module with
            the following code:

            from main import app

      app   Using import string: main:app

   server   Server started at http://127.0.0.1:8000
   server   Documentation at http://127.0.0.1:8000/docs

      tip   Running in development mode, for production use:
            fastapi run

            Logs:

     INFO   Will watch for changes in these directories:
            [ '/home/user/code/awesomeapp' ]
     INFO   Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C
            to quit)
     INFO   Started reloader process [383138] using WatchFiles
     INFO   Started server process [383153]
     INFO   Waiting for application startup.
     INFO   Application startup complete.

                                                        restart ↻
```

Connecting to the right endpoint

You can declare path "parameters" or "variables" with the same syntax used by Python format strings:

example.py

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/{item_id}")
async def read_item(item_id):
  return {"item_id": item_id}
```

The value of the path parameter item_id will be passed to your function as the argument item_id.

Query Parameters

Sending data in the path without being on the parameters

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters.

example.py

```
from fastapi import FastAPI

app = FastAPI()

fake_items_db = [{"item_name": "Foo"}, {"item_name": "Bar"}, {"item_name":
"Baz"}]

@app.get("/items/")
async def read_item(skip: int = 0, limit: int = 10):
  return fake_items_db[skip : skip + limit]
```

The query is the set of key-value pairs that go after the ? in a URL, separated by & characters.

Request Body

Sending data in the path without being on the parameters

- When you need to send data from a client (let's say, a browser) to your API, you send it as a **request body**.
- A **request** body is data sent by the client to your API. A **response** body is the data your API sends to the client.
- Your API almost always has to send a **response** body. But clients don't necessarily need to send **request bodies** all the time, sometimes they only request a path, maybe with some query parameters, but don't send a body.

Request Body

For Body requests we need to use Pydantic's BaseModel

First, you need to import BaseModel from Pydantic and then you declare your data model as a class that inherits from BaseModel.

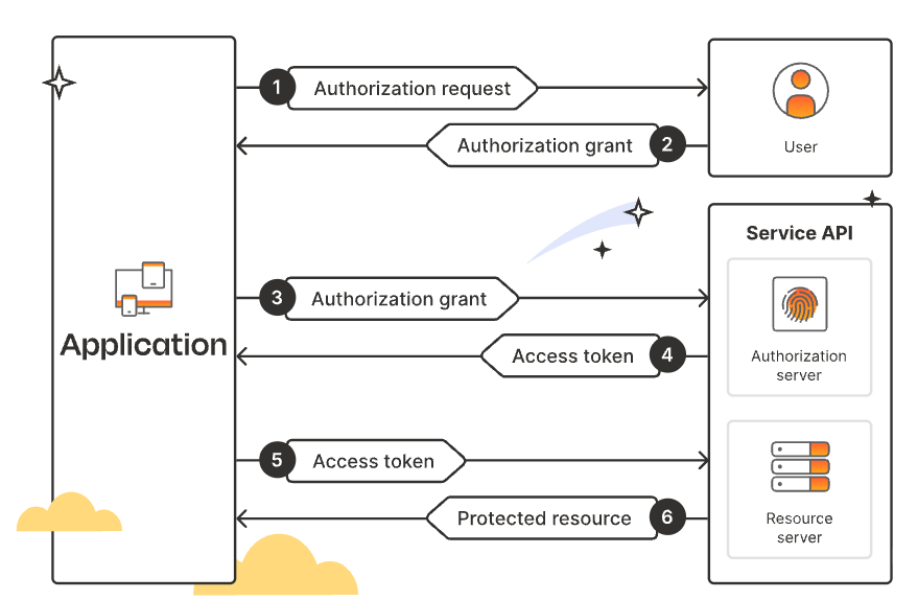
example.py

```python
from fastapi import FastAPI
from pydantic import BaseModel

class Item(BaseModel):
  name: str
  description: str | None = None
  price: float
  tax: float | None = None

app = FastAPI()

@app.post("/items/")
async def create_item(item: Item):
  return item
```

Securing our API

It is important that we add security to our applications

OAuth is a token-based authentication mechanism that enables a user to grant third-party access to their account without having to share their login credentials. OAuth 2.0, which provides greater flexibility and scalability than OAuth 1.0, has become the gold standard for API authentication, and it supports extensive API integration without putting user data at risk.



FastAPI & Render demo

Render demo

Today we will deploy an API on Render

main
Expand file tree
buidling-deploying-agents-applications / classes / class-05-deployment-interfaces / demos / fastapi-agent-service /
Go to file t
Add file

Luis Dias Add initial project structure with .gitignore, README, and demo files... 423cd4e · yesterday

Name	Last commit message	Last commit date
..		
.env.example	Add initial project structure with .gitignore, README, and dem...	yesterday
README.md	Add initial project structure with .gitignore, README, and dem...	yesterday
demo_client.py	Add initial project structure with .gitignore, README, and dem...	yesterday
main.py	Add initial project structure with .gitignore, README, and dem...	yesterday
requirements.txt	Add initial project structure with .gitignore, README, and dem...	yesterday

README.md

FastAPI Agent Service

A small API that powers the Streamlit demo. The entire project fits in this folder so you can see how UI, API, and monitoring connect. The `/chat` endpoint now uses a LangGraph ReAct agent with two simple tools so you can show students how agents access helper functions.

What you need first

- Python 3.10 or newer installed on your machine

Personal
Projects
Search ⌘ K
+ New
Upgrade

Projects
Blueprints
Environment Groups

INTEGRATIONS
Observability
Webhooks
Notifications

NETWORKING
Private Links New

WORKSPACE
Billing
Settings

Overview

Invite your team
+ Add new

Projects

Get organized with Projects

An easier way to organize your resources and collaborate with team members.

+ Create your first project

Learn more

Ungrouped Services

Active (0) | Suspended (1) | All (1)

Search services

SERVICE NAME	STATUS	RUNTIME	REGION	DEPLOYED
agent-api-render-test	Suspended by you	Python 3	Frankfurt	19d

https://github.com/diaxz12/BUILDING-AND-DEPLOYING-AI-AGENTS---Part-2/tree/main/classes/class-05-deployment-interfaces/demos/fastapi-agent-service

Render expected outcome

Class 5 fastapi-agent-service demo

1. Use the provided OpenAI API Key so you can use GPT models.
2. Create a new Github repo and follow the step-by-step instructions defined on the README.md
3. Create a new **Web Service** on Render and give Render permission to access your new repo.
4. Provide the following values during service creation:

SETTING	VALUE
Language	Python 3
Build Command	pip install -r requirements.txt
Start Command	uvicorn main:app --host 0.0.0.0 --port $PORT

Want to learn more about FastAPI?

Today FastAPI demo was just a quick intro. I advise to go through a dedicated tutorial of FastAPI!

Official documentation:

- FastAPI Documentation - Full usage details
- FastAPI GitHub Repository - FastAPI source code

Tutoriais:

- FastAPI Community - Official FastAPI community
- User Guide - FastAPI step by step tutorial covering all features

FastAPI examples:

- GitHub Example Repositories - open-source examples

Langfuse

Adding observability to our AI Agents

Monitoring our AI Agents

Let's use Langfuse to monitor our AI Agents applications

Monitoring an AI Agent is key to understand it's usage, debugging and improve the developed agent overtime. In our demo we will require to **create a .env file based** on the **provided .env.example file**.

https://github.com/diaxz12/BUILDING-AND-DEPLOYING-AI-AGENTS---Part-2/blob/main/classes/class-05-deployment-interfaces/demos/fastapi-agent-service/.env.example

Monitoring with Langfuse

Let's use the Langfuse CallbackHandler() to monitor the LLM calls

We will focus mainly on tracing all calls to the AI Agent

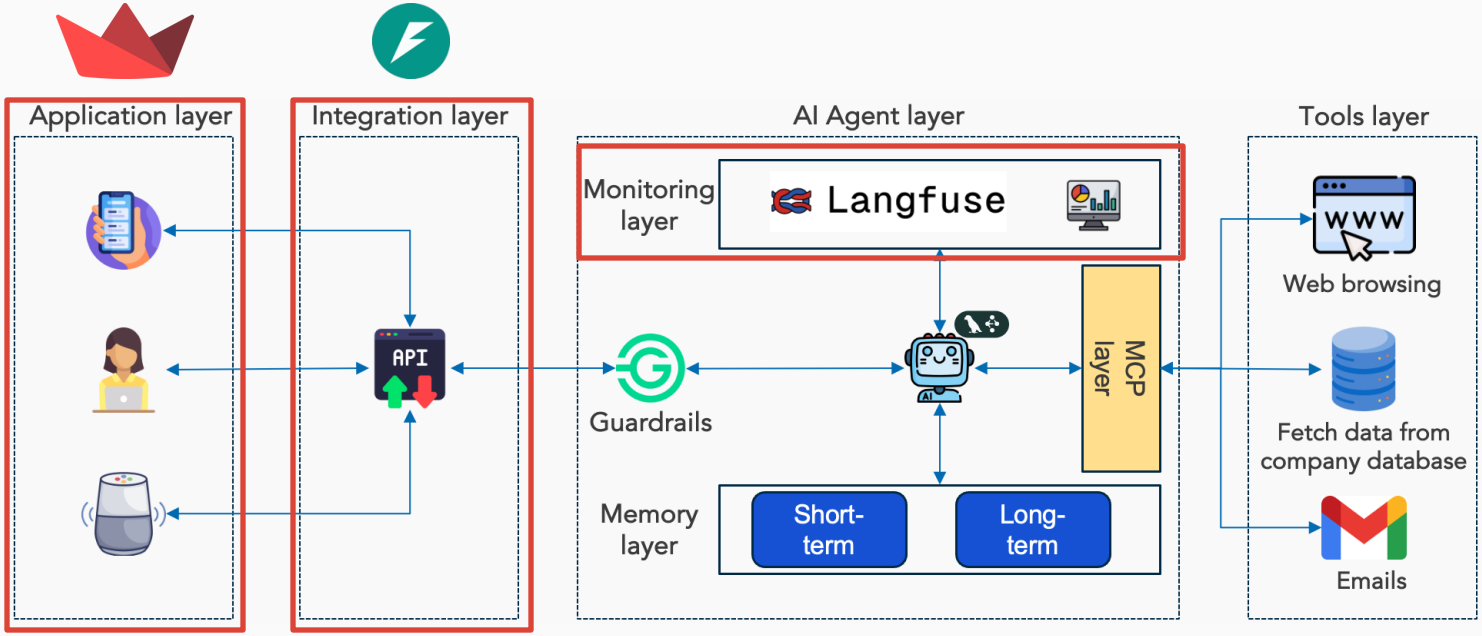
```python
def run_agent(message: str) -> Optional[str]:
    """Ask the LangGraph agent for a reply; return None if unavailable."""

    # Initialize Langfuse CallbackHandler for Langchain (tracing)
    langfuse_handler = CallbackHandler()

    if agent_runner is None:
        return None

    try:
        result = agent_runner.invoke({"messages": [("user", message)]}, config={"callbacks": [langfuse_handler]})
    except Exception as exc:  # fall back to rule-based helper on errors
        print(f"[LangGraph] agent invocation failed: {exc}")
        return None

    messages = result.get("messages") if isinstance(result, dict) else None
    if not messages:
        return None

    last_message = messages[-1]
    content = getattr(last_message, "content", last_message)
    return _content_to_text(content)
```

Let’s piece everything

End-to-End Flow

Review the final flow

- User → Streamlit UI → FastAPI API → Agent → Langfuse monitoring.



Reflection Questions

- What new challenges arise when moving an agent from a notebook or prototype into production?
- Which of these layers do you consider most critical for your capstone project, and why?
- How might your capstone project benefit from having a Streamlit UI?
- What is the role of API endpoints and schemas in ensuring reliable communication between the UI and the AI Agent?
- What technical or conceptual gaps do you still need to address before deploying your agent?

Practice Practice Practice

Assignment Instructions

1. Clone the classes repo and use Class 5 demo as the foundation.
2. Set up the FastAPI service.
3. Add the Tavily search tool. Find more details on https://www.tavily.com/.
4. Prime the agent for the class 5 context.
5. Expose locally the API with Langfuse monitoring enabled.
6. Update the Streamlit UI.
7. Deploy the API + UI.
8. Document your work.
9. Submit your work by email the:
    - Source code of the assignment.
    - README.md with sample Q&A transcript(s).
    - The Streamlit UI URL.
    - Confirmation (use screenshots or a video) that Tavily, LangGraph, and Langfuse all ran during testing.

Wrap-up

Wrap-Up & Next Steps

You’ve deployed an AI agent with

- Streamlit to the cloud.
- Developed an API with FastAPI to integrate with the Streamlit UI.
- Deployed the API to render in order to make it available over the public internet.
- You integrated monitoring with Langfuse in order to assess, debug and improve the agent.

Next: build a simple API and UI to start interacting with the AI Agent your are developing for the Medium article.

Questions?

luis.f.s.m.dias@gmail.com
https://www.linkedin.com/in/luisfilipedias/