neuralware.

How to Build and Deploy an AI Agent using LangGraph from Scratch

AI AGENTS DEPLOYMENT LANGGRAPH

AUTHOR
Prashant Patel

PUBLISHED
April 25, 2025

Building and deploying a LangGraph AI Agent from scratch involves understanding the framework's architecture, defining your agent's workflow as a graph, implementing nodes and state management, and finally deploying the agent either locally or on LangGraph Cloud. Below is a detailed guide covering all these steps.

Before we dive-in, let's get ourselves familiarised with some components involved in developing and deploying a LangGraph application.

Components

- **LangGraph** is the foundational library enabling agent workflow creation in Python and JavaScript.
- **LangGraph API** wraps the graph logic, managing asynchronous tasks and state persistence, serving as the backend engine.
- **LangGraph Cloud** hosts the API, providing deployment, monitoring, and accessible endpoints for running graphs in production.
- **LangGraph Studio** is the development environment that leverages the API backend for real-time graph building and testing, usable locally or in the cloud.
- **LangGraph SDK** offers programmatic access to LangGraph graphs, abstracting whether the graph is local or cloud-hosted, facilitating client creation and workflow execution.

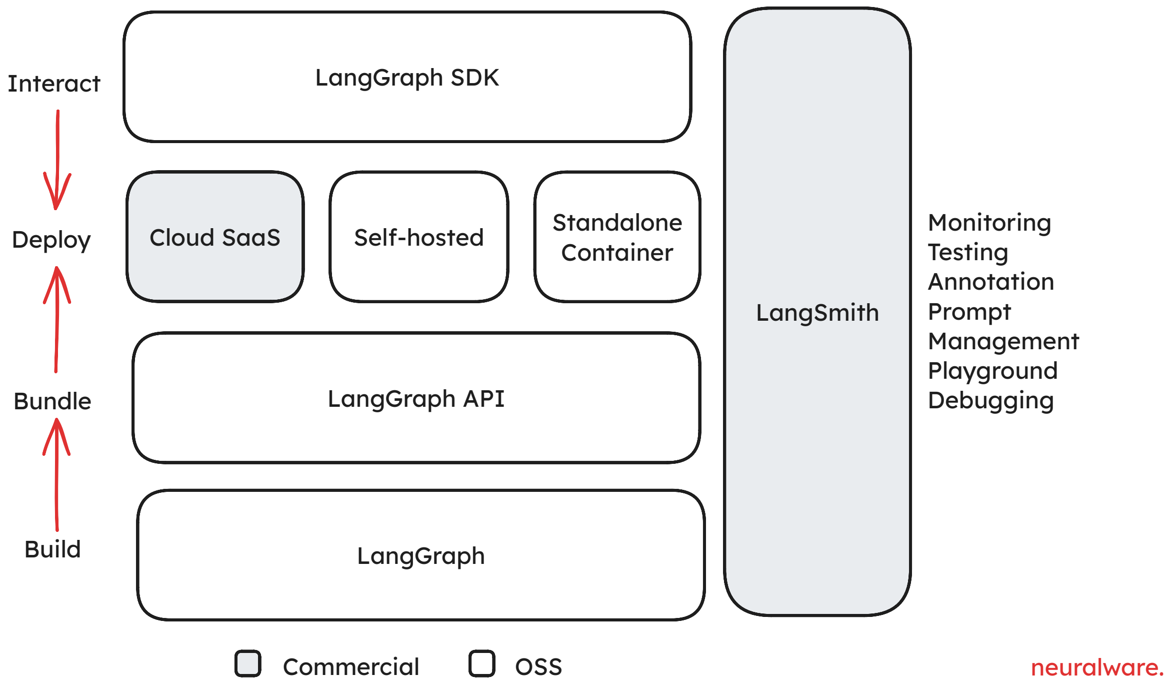


Figure 1: LangGraph Components

Set Up

We'll be building a simple Agent to demonstrate the end-to-end process. Building more sophisticated AI agents is a topic better suited for a dedicated post.

To start with, create a following project structure and open `langgraph_deployment` directory in your favorite code editor.

```
langgraph_deployment/
├──src
│   ├──chat_agent/
│   │   ├── utils/
│   │   │   ├── __init__.py
│   │   │   ├── nodes.py   # Node functions
│   │   │   └── state.py   # State definitions
│   │   ├── __init__.py
│   │   └── agent.py       # Graph construction code
│   └── __init__.py
├── main.py                # LangGraph SDK demo
├── .env                   # Environment variables (add OPENAI_API_KEY)
├── .python-version        # Python version for this project (3.11)
└── langgraph.json         # LangGraph API configuration
```

Install dependencies

We will be using `uv` for dependency management, if you haven't used `uv` package manager before - now is the good time to start. You can install `uv` by following instructions on their official page.

Once installed run the below commands to set up the python environment.

```
uv python install 3.11
uv init
```

On successful completion of above commands, you will see `pyproject.toml` added to the project. This is uv equivalent for the `requirements.txt` for managing project dependencies and bundling the project.

Append following lines to `pyproject.toml`. This is so that uv recognizes the build system and can build and install your package into the project environment.

pyproject.toml

```toml
[build-system]
requires = ["setuptools>=42", "wheel"]
build-backend = "setuptools.build_meta"

[tool.setuptools]
package-dir = {"" = "src"}
```

Finally, install the dependencies.

```
uv add "langchain[openai]" "langgraph-cli[inmem]" \
        langgraph langgraph-api langgraph-sdk
```

Build

Time to get our hands dirty (in a good way)!

We will create a marketing research AI assistant using OpenAI's `gpt-4o` model. Since we won't be using any tools or any special patterns, our graph will be simple (intentionally). Let's start adding code to the files we created.

Define the Agent State

LangGraph uses a shared state object to pass information between nodes. Define the state using Python's TypedDict to specify the data structure.

state.py

```python
from operator import add
from typing import Annotated, TypedDict, Any
from langchain_core.messages import AnyMessage

class State(TypedDict):
    messages: Annotated[list[AnyMessage], add]  # Accumulates messages
```

This state holds a list of messages (e.g., user input, AI responses) that are appended to as the conversation proceeds.

Implement Node Functions

Nodes are the processing units of your graph. Each node is a function that takes the current state as input and returns an updated state. We will add a system message to instruct the model to follow a specific role, tone, or behavior during its execution within the node.

nodes.py

```python
from dotenv import load_dotenv
from langchain.chat_models import init_chat_model
from langchain_core.messages import SystemMessage

from .state import State

_ = load_dotenv()

model = init_chat_model(model="gpt-4o", model_provider="openai")

instructions = (
    "You are a Marketing Research Assistant. "
    "Your task is to analyze market trends, competitor strategies, "
    "and customer insights. "
    "Provide concise, data-backed summaries and actionable recommendations. "
)
system_message = [SystemMessage(content=instructions)]


def chat(state: State) -> dict:
    messages = system_message + state["messages"]
    message = model.invoke(messages)  # Generate response
    return {"messages": [message]}  # Return updated messages
```

Construct the Graph Workflow

Use LangGraph's `StateGraph` to build your agent's workflow by adding nodes and edges that define the flow of execution.

agent.py

```python
from langgraph.graph import END, START, StateGraph
from utils.nodes import chat
from utils.state import State

graph_builder = StateGraph(State)

graph_builder.add_node("chat", chat)

graph_builder.add_edge(START, "chat")
graph_builder.add_edge("chat", END)

graph = graph_builder.compile()
```

Bundle

Typically, you'd need to write a backend to expose your application as an API. However, with LangGraph, that entire step is streamlined - simply add a config file and launch the service.

LangGraph API Config

To configure the API, define the settings in `langgraph.json` by adding the following lines. The configuration is straightforward, so we won't go into the details of each field here. However, if you're interested in exploring advanced configuration options, you can refer to the official documentation here.

langgraph.json

```json
{
    "graphs": {
        "agent": "chat_agent/agent.py:graph"
    },
    "env": ".env",
    "python_version": "3.11",
    "dependencies": [
        "."
    ]
}
```

Launch Service

Once your configuration is in place, use the following command to bundle the application and launch it as a web service:

```
langgraph dev
```

This command not only generates an API with ready-to-use endpoints but also spins up a web-based chat interface (Studio UI) that makes it easy to test and debug your application in real time.

🚀 API: http://127.0.0.1:2024
🎨 Studio UI: https://smith.langchain.com/studio/?baseUrl=http://127.0.0.1:2024
📚 API Docs: http://127.0.0.1:2024/docs

Deploy

At this point, we have the LangGraph server running locally. Now, it's time to deploy it. As shown in the component diagram, there are three primary deployment options available:

1. Cloud SaaS: Deploy LangGraph Servers to LangChain's managed cloud infrastructure.
2. Self-hosted: Host and manage the LangGraph infrastructure within your own cloud environment - either just the data plane (with LangChain maintaining the control plane), or fully self-host both for complete control.
3. Standalone container: Package the LangGraph Server as a Docker image and deploy it wherever you like - such as a Kubernetes cluster - while connecting to separately hosted Postgres and Redis instances.

Each option comes with trade-offs:

- Option 1 requires the least setup but involves commercial licensing and vendor lock-in.
- Option 2 demands the most effort in terms of design, setup, and ongoing management, but offers maximum control.
- Option 3 strikes a balance, offering flexibility in deployment without full platform dependency.

Although there are trade-offs either option can be selected based on your internal evaluation and convenience. We will explore the Option 3 - Standalone Container.

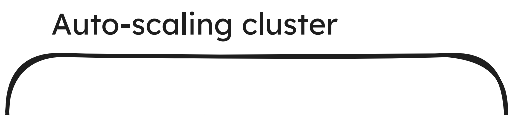
Standalone Container

LangGraph supports standalone container deployments using Docker, making it possible to run a scalable, self-managed service - ideal for deploying to environments like Kubernetes. This option is non-commercial and gives you full control over the infrastructure.

While we won't be deploying to a Kubernetes cluster in this guide, we'll simulate the setup locally using Docker to mirror the architecture shown in the diagram below. Specifically, we'll deploy three core services:

- LangGraph API – the same service we configured and tested locally in the previous step.
- PostgreSQL – used for persistent storage by LangGraph.
- Redis – used for task orchestration and streaming output events through Pub/Sub.

This local deployment will serve as a lightweight, production-like replica, allowing us to validate the architecture and interactions between components without the overhead of a full Kubernetes setup.



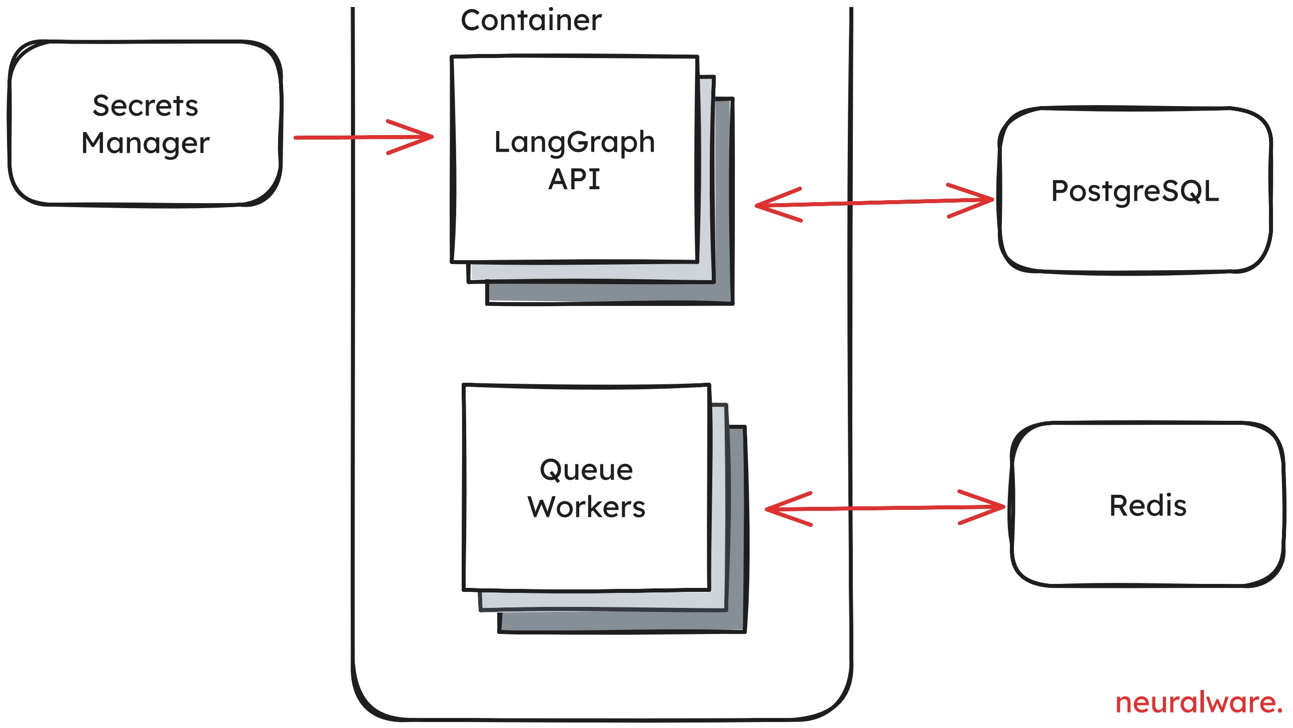


Figure 2: Standalone Container Deployment

Local Deployment with Docker Compose

To simplify service orchestration, we'll use Docker Compose to bring up the LangGraph API along with its required dependencies - PostgreSQL and Redis. This setup replicates a production-like environment locally and abstracts away the need to manage containers individually.

LangGraph API

If the application is structured correctly, you can build a docker image with the LangGraph Deploy server.

```
langgraph build --tag chat_agent:v1.0.0
```

This command packages your application into a Docker image named `chat_agent:v1.0.0`, ready for deployment.

Docker Compose

We'll define the services in a `docker-compose.yml` file as shown below.

docker-compose.yml

```
volumes:
  langgraph-data:
    driver: local

services:
  langgraph-redis:
    image: redis:6
    healthcheck:
      test: [ "CMD", "redis-cli", "ping" ]
      interval: 5s
      timeout: 1s
      retries: 5

  langgraph-postgres:
    image: postgres:16
    ports:
      - "5433:5432"
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
    volumes:
      - langgraph-data:/var/lib/postgresql/data
    healthcheck:
      test: [ "CMD", "pg_isready", "-U", "postgres" ]
      start_period: 10s
      timeout: 1s
      retries: 5
      interval: 5s

  langgraph-api:
    image: ${IMAGE_NAME}
    ports:
      - "8123:8000"
    depends_on:
      langgraph-redis:
        condition: service_healthy
      langgraph-postgres:
        condition: service_healthy
    env_file:
      - .env
    environment:
      REDIS_URI: redis://langgraph-redis:6379
      LANGSMITH_API_KEY: ${LANGSMITH_API_KEY}
      POSTGRES_URI: postgres://postgres:postgres@langgraph-postgres:5432/postgres?sslm
```

Running the Application

Ensure that your .env file includes valid values for `IMAGE_NAME` and `LANGSMITH_API_KEY` before running the Docker Compose setup.

.env

```
OPENAI_API_KEY=your-openai-api-key
LANGSMITH_API_KEY=your-langsmith-api-key
IMAGE_NAME=chat_agent:v1.0.0
```

Once everything is set, start the stack by running:

```
docker compose up --build
```

This will -

- Start and link the required Redis and PostgreSQL containers

- Start and link the required Redis and PostgreSQL containers
- Build and launch your LangGraph API container
- Wait for dependent services to become healthy before launching the API
- Expose the API locally at http://localhost:8123

You can test that the application is up by checking:

```
curl --request GET --url 0.0.0.0:8123/ok
```

Assuming everything is running correctly, you should see a response like:

```
{"ok":true}
```

Interact

If you've made it this far - congratulations! 🎉

The final step is to interact with your deployed LangGraph server using the LangGraph SDK.

LangGraph provides SDKs for both Python and JavaScript/TypeScript, making it easy to work with deployed graphs - whether hosted locally or in the cloud. The SDK abstracts away the complexity of direct API calls and provides:

- A unified interface to connect with LangGraph graphs
- Easy access to assistants and conversation threads
- Simple run execution and real-time streaming capabilities

Following is the code-snippet in Python to demonstrate interaction with the graph we deployed in the previous step -

main.py

```python
import asyncio

from langchain_core.messages import HumanMessage
from langgraph_sdk import get_client

LANGGRAPH_SERVER_URL = "http://localhost:8123"

async def main() -> None:

    # Initialize the LangGraph client
    client = get_client(url=LANGGRAPH_SERVER_URL)

    # Fetch the list of available assistants from the server
    assistants = await client.assistants.search()

    # Select the first assistant from the list
    agent = assistants[0]

    # Create a new conversation thread with the assistant
    thread = await client.threads.create()
    # Prepare the user message
    user_message = {"messages": [HumanMessage(content="Hi")]}

    # Stream the assistant's response in real-time
    async for chunk in client.runs.stream(
        thread_id=thread["thread_id"],
        assistant_id=agent["assistant_id"],
        input=user_message,
        stream_mode="values",
    ):
        # Filter out metadata events and empty data chunks
        if chunk.data and chunk.event != "metadata":
            # Print the content of latest assistant message
            print(chunk.data["messages"][-1]["content"])


if __name__ == "__main__":
    asyncio.run(main())
```

The above code is intuitive, so I'll leave it to you to run and explore.

At first glance, the response you get might feel modest - a simple message in return for a simple prompt. But behind that is a clean, scalable setup that mirrors how production-grade AI systems are built. You've laid the groundwork for something much bigger. Swapping out this basic agent for a more advanced one is just a matter of choice now. So while it might not feel flashy, make no mistake - you've just put a serious system in place.

> ***Want to take this further?*** *Try integrating it into a Streamlit frontend to create an interactive, real-time chat experience.*

Further Reading

1. How to Deploy to Cloud SaaS
2. Self-Hosted Data Plane
3. Self-Hosted Control Plane
4. How to add custom authentication
5. How to add custom routes
6. How to integrate LangGraph into your React application
7. LangGraph SDK

References

1. LangGraph Deployment Options
2. How to do a Self-hosted deployment of LangGraph
3. How to Deploy a Standalone Container

Reuse

Citation

BibTeX citation:

```
@online{patel2025,
  author = {Patel, Prashant},
  title = {How to {Build} and {Deploy} an {AI} {Agent} Using {LangGraph}
    from {Scratch}},
  date = {2025-04-25},
  url = {https://neuralware.github.io/posts/langgraph-deployment/},
  langid = {en}
}
```

For attribution, please cite this work as:

Patel, Prashant. 2025. "How to Build and Deploy an AI Agent Using LangGraph from Scratch." April 25, 2025. https://neuralware.github.io/posts/langgraph-deployment/.