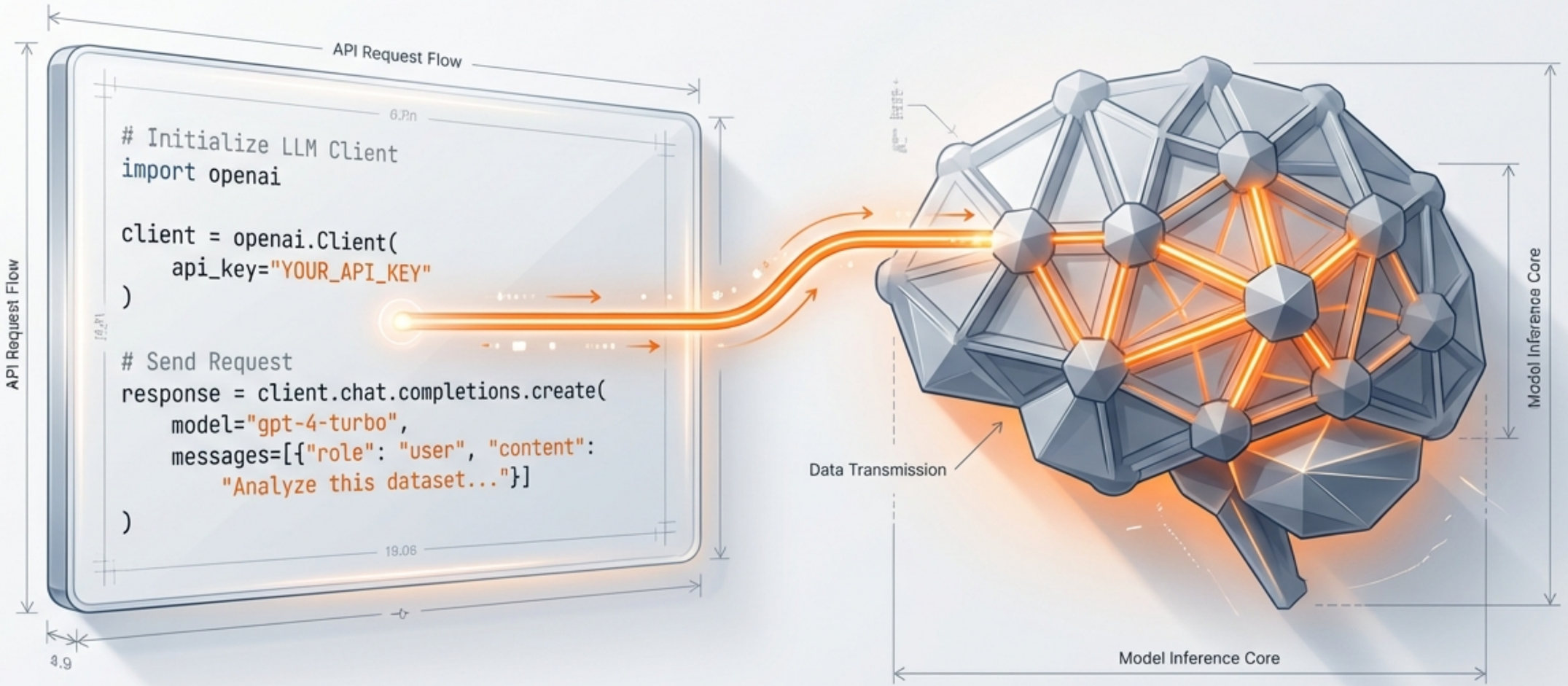
Connecting Your Code to a Large Language Model

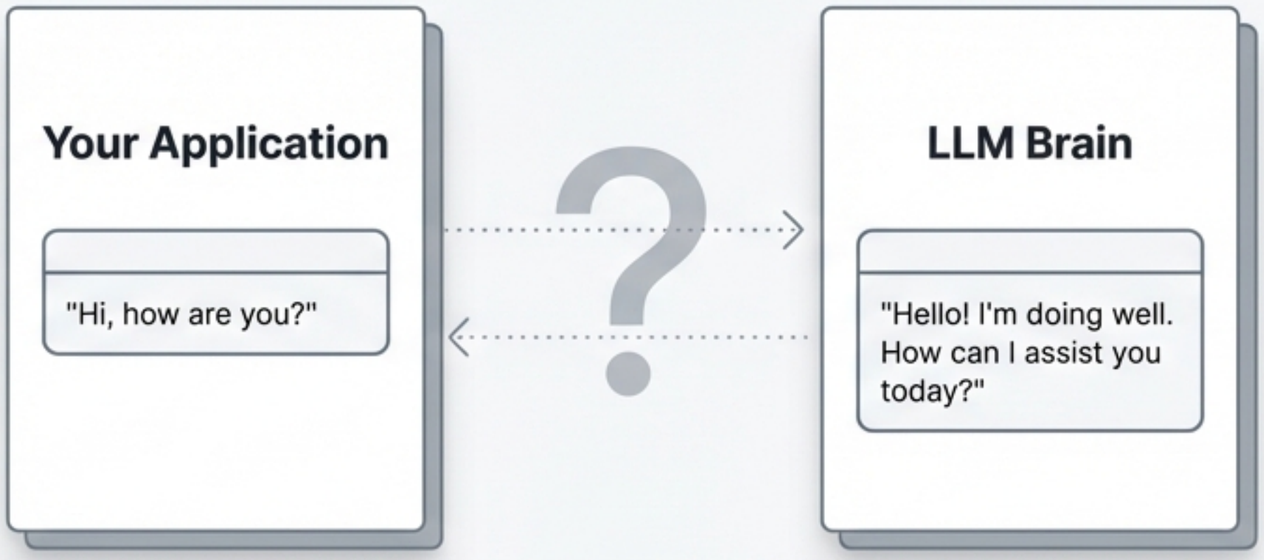
A Developer's Guide to API Integration & Model Access



Every Great Application Starts with a Simple Idea

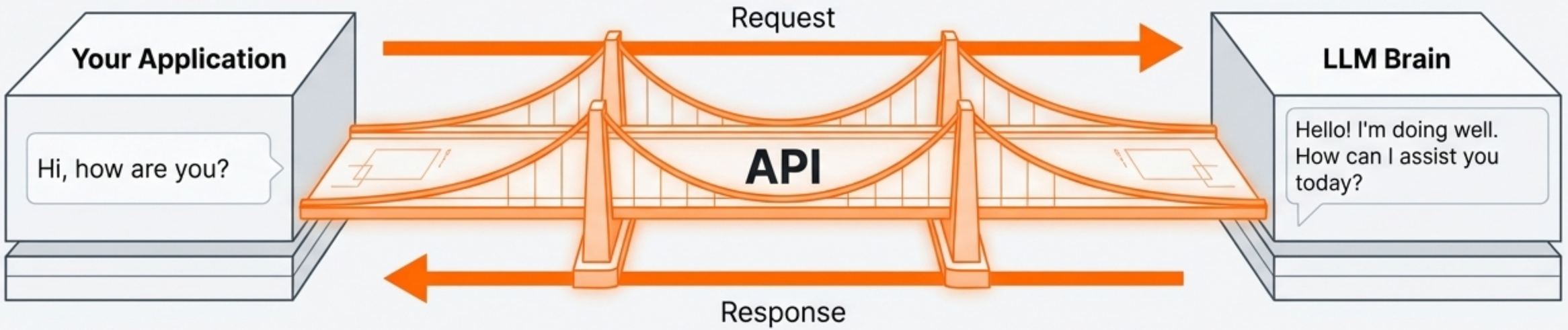
Before we dive into the mechanics, let's anchor our journey to a goal: building a simple chatbot.

An application needs a 'brain' to process user input and generate intelligent responses. That brain is a Large Language Model. But how do we establish the connection?



The API is the Bridge to Your Model's Intelligence

An Application Programming Interface (API) is the crucial link. It's a standardized contract that allows your application to send requests (like a user's question) to a remote LLM and receive its generated response. This approach eliminates the need to host and manage massive models yourself, providing access to state-of-the-art AI with a simple network call.



Scalable: Access massive computational power on demand.

Managed: No need to worry about model hosting, maintenance, or hardware.

Standardized: A consistent way to interact with different models.

Choosing Your AI Engine: A Look at the Leading API Providers

Several platforms offer robust LLM APIs, each catering to different needs. Your choice depends on factors like model availability, performance requirements, cost, and customization options.

OpenAI

Best For

Access to state-of-the-art models (GPT-4o), ease of use, strong ecosystem.

Key Models

GPT-4, GPT-3.5-Turbo

Hugging Face

Best For

Unparalleled variety of open-source models, custom fine-tuning, and dedicated Inference Endpoints.

Key Feature

Access thousands of community models (Mistral, Llama, etc.).

GroqCloud

Best For

Unmatched inference speed and low latency, powered by custom LPU hardware.

Key Feature

Language Processing Unit (LPU) architecture designed to solve LLM bottlenecks like compute density and memory bandwidth.

Your API Key is the Key to the Engine Room

To use an API, you first need to authenticate your application. This is done using a unique API key, a secret token that identifies your project and tracks your usage. Treat your API keys like passwords—never expose them in your client-side code or commit them to public repositories.

The Developer Workflow

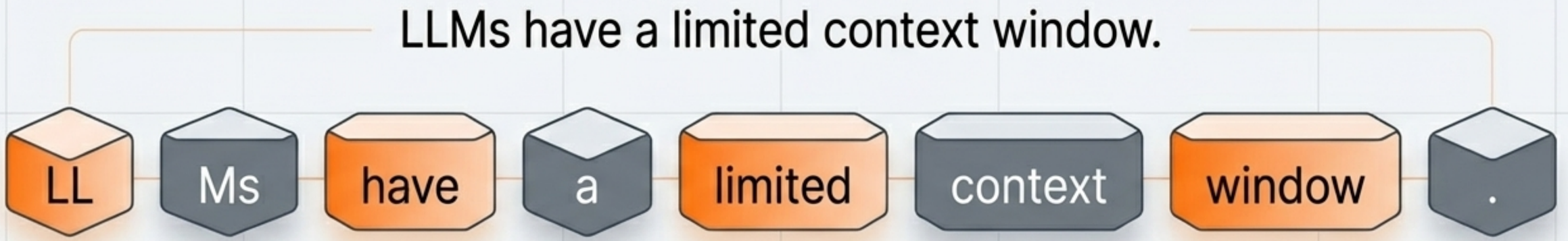
1.  **Sign Up:** Create an account on the provider's platform (e.g., OpenAI, Groq).
2.  **Generate Key:** Navigate to the API section and create a new secret key.
3.  **Store Securely:** Store the key in an environment variable file (e.g., `.env`).

Standard Practice: Environment Variables

```
# 1. Create a .env file
GROQ_API_KEY="your_secret_key_here"

# 2. Load the key in your application
import os
from dotenv import load_dotenv

load_dotenv()
api_key = os.getenv("GROQ_API_KEY")
```

Understanding Tokens: The Currency of LLMs

LLMs don't see words; they see "tokens." A token can be a word, a part of a word, or punctuation. The text you send to the model and the text it generates are both measured in tokens. This is the fundamental unit that determines cost and fits within the model's "context window."



Tokenization

The process of breaking text into tokens. Different models use different tokenizers.

Context Window

The maximum number of tokens an LLM can process in a single request (input + output). For example, Groq's Llama 2 70b has a 4096-token context window. This is the model's short-term memory.

Managing Your Fuel & Speed: Costs and Rate Limits

Every API call consumes tokens, which translates directly to cost. Providers typically price per million tokens, often with different rates for input (prompt) and output (completion). To ensure fair usage and system stability, they also impose rate limits.

Understanding Costs

```
`Total Cost = (Input Tokens ×
   Price_per_Input_Token) +
   (Output Tokens × Price_per_Output_Token)`
```

Example (using Groq pricing for Gamma 7B)	
Input:	$0.10 / 1M tokens
Output:	$0.10 / 1M tokens

Understanding Rate Limits

Limits on how many requests you can make or how many tokens you can process in a given time period (e.g., requests per minute, tokens per minute). This prevents a single user from overwhelming the system.

Bringing It All Together: Your First API Call

With our API key loaded and an understanding of the core concepts, we can now write the code to interact with an LLM. Frameworks like LangChain and LangGraph abstract away the complexity, allowing us to invoke a model with a single line.

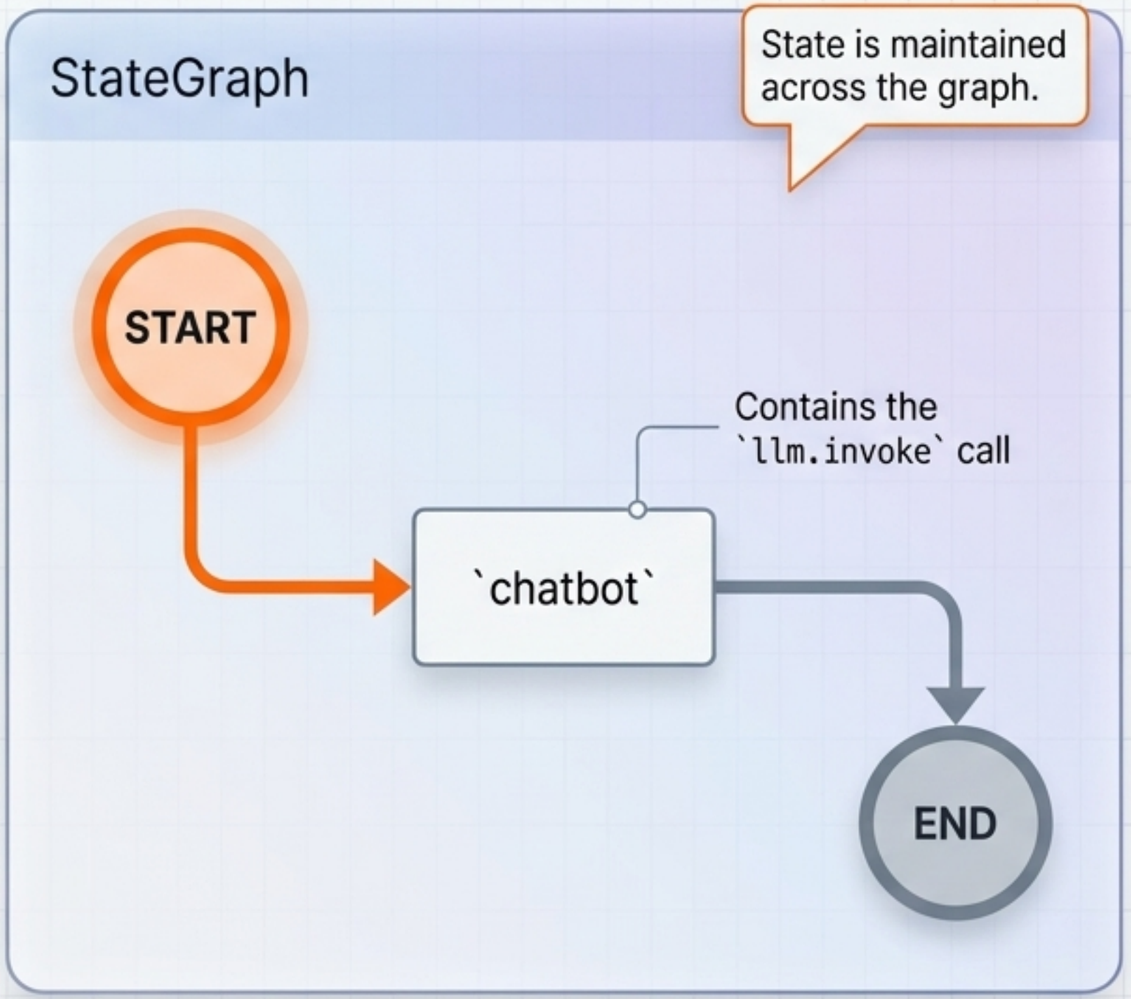
```python
import os
from langchain_groq import ChatGroq
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# 1. Initialize the Model
# Assumes GROQ_API_KEY is in your environment
llm = ChatGroq(model_name="llama2-70b-4096")

# 2. Create a Prompt Template
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("user", "{input}")
])

# 3. Define the Output Parser
output_parser = StrOutputParser()

# 4. Create the Chain
chain = prompt | llm | output_parser

# 5. Invoke the Chain
response = chain.invoke({"input": "What is an LPU?"})
print(response)
```

From a Single Call to a Stateful Application

Real applications require more than one-off responses. They need to manage state, handle multi-turn conversations, and execute complex workflows. **LangGraph** allows us to define these workflows as a `StateGraph`, where each node is a function and edges direct the flow of logic.

Core LangGraph Components

- **State:** A central object (a Python class inheriting from `TypedDict`) that holds information passed between nodes. This is the graph's memory.
- **Nodes:** Python functions that perform actions (like calling an LLM) and modify the state.
- **Edges:** Connections that define the path from one node to the next, including conditional logic.

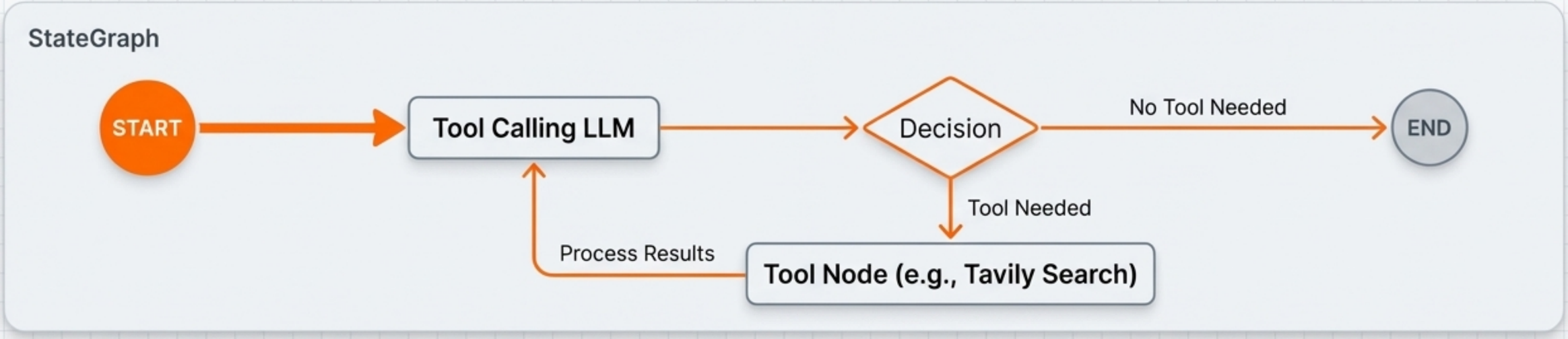


Giving Your Application Tools to Act

The true power of LLMs is unlocked when they can **interact with external systems**. By “binding” tools to an LLM, we give it the ability to decide *when* to call a function or query an external API to get information it doesn’t have. This is the foundation of agentic AI.

Example Use Case:

1. **User Input**
   "What is the latest news in AI?"
2. **LLM Decision**
   Recognizes need for real-time info, decides to use a tool.
3. **Tool Call**
   “TavilySearch” API
4. **Response Generation**
   Search results are returned to the LLM, which synthesizes an answer.



Adding Memory to Your Conversation

By default, each API call is stateless. To build a true chatbot, your application must remember previous interactions.

LangGraph handles this through “checkpointers,” which save the state of the conversation. This allows the LLM to access the history of the dialogue and provide contextually aware responses.

```
# 1. Initialize a memory saver
from langgraph.checkpoints.memory import MemorySaver
memory = MemorySaver()

# 2. Compile the graph with the checkpointer
graph = builder.compile(checkpointer=memory)

# 3. Configure a unique thread_id for each session
config = {"configurable": {"thread_id": "user_session_123"}}

# 4. Invoke with the config to maintain memory
graph.invoke({"messages": ["Hi, my name is Alex."]}, config)
graph.invoke({"messages": ["What is my name?"]}, config)
# The LLM will now remember the name "Alex".
```

Your Developer's Blueprint for LLM Integration

You now have the complete blueprint for connecting your code to an LLM. By understanding these core components, you can move from a simple idea to a sophisticated, agentic application.