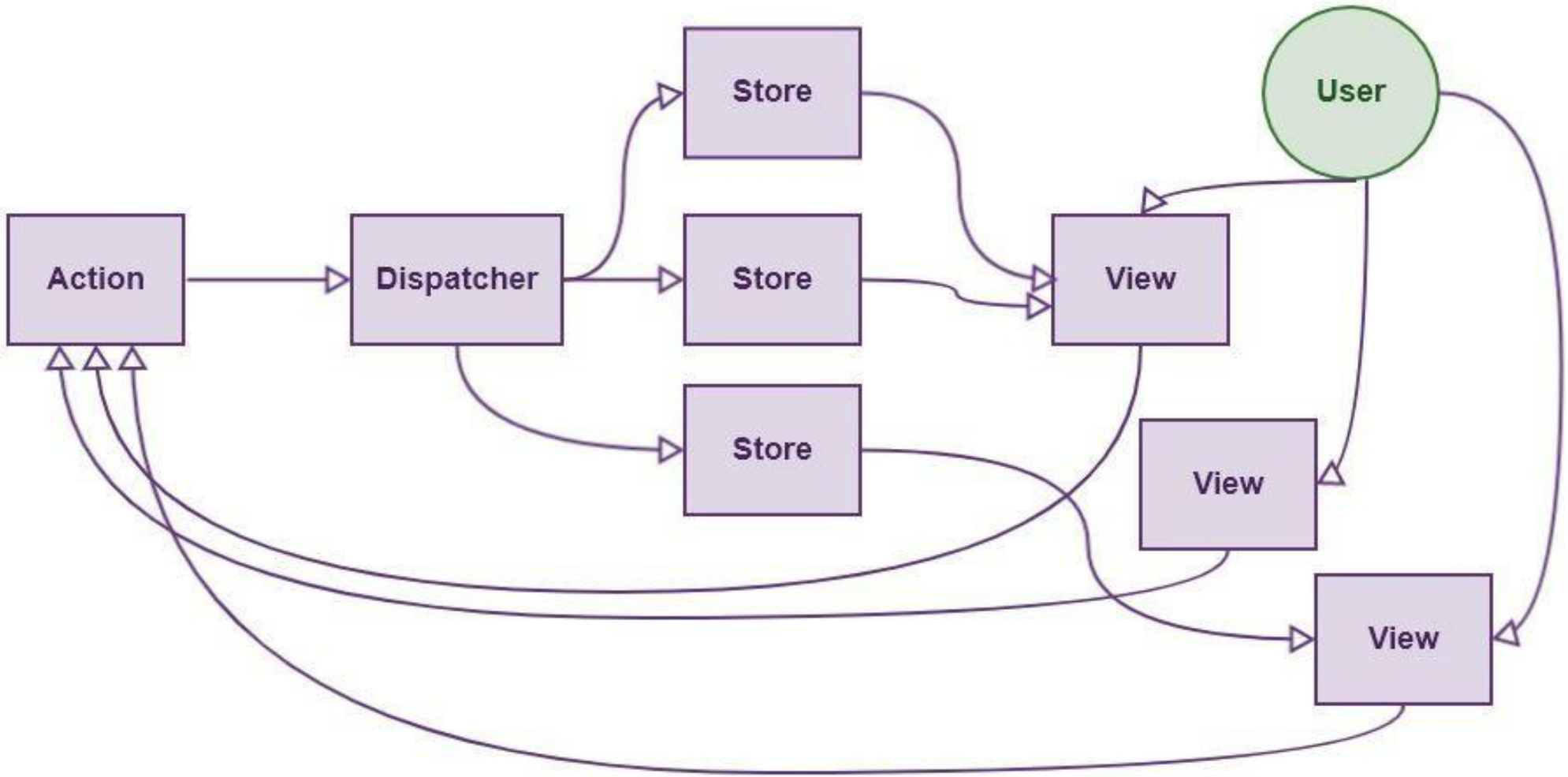
Redux

Flux Architecture

- Flux is an application architecture introduced by Facebook for building client-side applications with a unidirectional data flow.
- Responsible for creating data layers in JavaScript applications
- Follows the concept of unidirectional data flow
- Not actually an framework/library rather an architecture
- Many implementations out there like Redux, Reflux, Fluxxor, Flummox and etc.
- Both Flux and Redux are architectures for managing

Flux Data FLow



Why Redux?

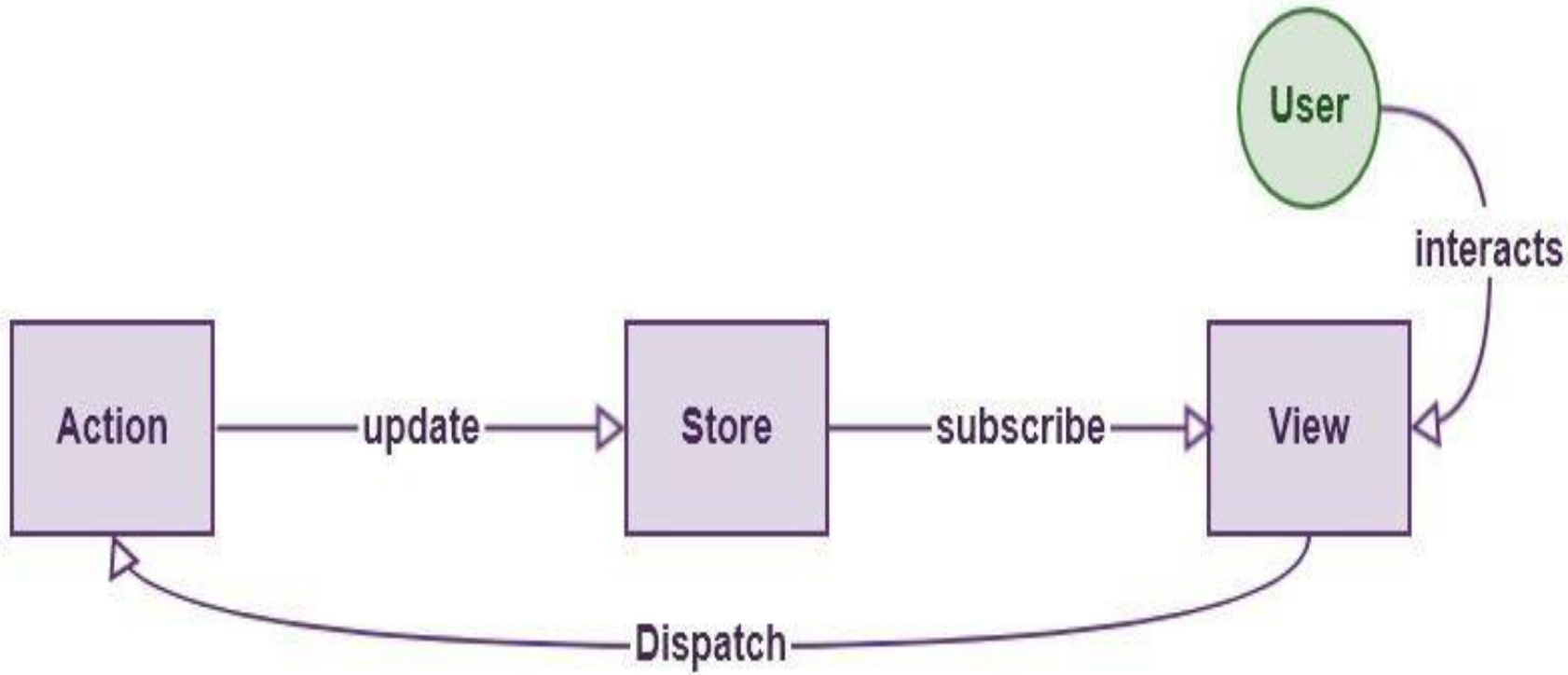
1. **Centralized State Management**
   - Redux stores the entire application state in a single object, making it easier to track and manage compared to scattered component states.
2. **Predictability & Debugging**
   - Redux follows a strict unidirectional data flow.
   - With tools like Redux DevTools, you can track state changes, time travel, and debug issues easily.
3. **Scalability & Maintainability**
   - Helps structure large applications by separating concerns (Actions, Reducers, Store).
   - Makes collaboration easier in teams by enforcing predictable patterns.
4. **Performance Optimization**
   - Uses immutability and state snapshots to avoid unnecessary re-renders.
   - Selectors and memoization (e.g., `reselect`) improve efficiency.

When to Use Redux

- Your app has complex state shared across multiple components.
- State changes frequently and needs to be logged/debugged.
- You want predictability and testability in state management.

When Not to Use Redux?

- Small apps with simple state (React Context is enough).
- When components rarely share state.
- If you don’t need advanced debugging or predictability.


User
interacts
Action
update
Store
subscribe
View
Dispatch

Building blocks of redux

1. Store (Centralized State Container)
   - The store holds the entire application state in one place.
   - It provides methods like `getState()`, `dispatch()`, and `subscribe()`.

```
import { createStore } from 'redux'; import
rootReducer from './reducers';
const store = createStore(rootReducer);
```

2. Actions (Describing What Happened)
   - Actions are plain JavaScript objects that describe what should happen in the app.
   - They must have a `type` property and can carry additional data (`payload`).

```
const incrementAction = { type:
'INCREMENT' };
```

3. Action Creators (Functions that Create Actions)
   - Action creators are functions that return action objects.
   - Useful for dynamic data and reusability.

```
const addUser = (name) => {
   return { type: 'ADD_USER', payload: { name } };
};
```

4. Reducers (Pure Functions that Modify State)
   - A reducer is a pure function that takes the current state and an action, then returns a new state.
   - It never modifies the original state directly.

```
const counterReducer = (state = { count: 0 }, action) => {
   switch (action.type) {
      case 'INCREMENT': return { count: state.count + 1 };
```

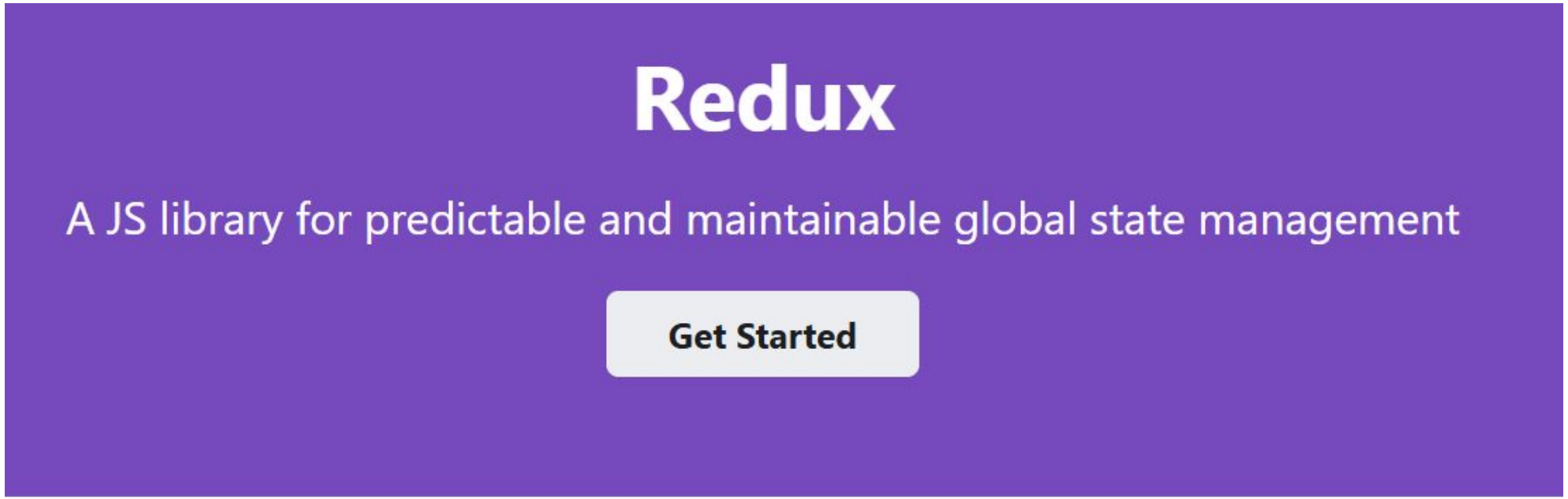
5. Dispatch (Sending Actions to the Store)
   - The `dispatch()` method is used to send actions to the store.
   - This triggers the reducers to update the state.

```
store.dispatch({ type: 'INCREMENT' });
```

References

https://redux.js.org/

https://redux.js.org/tutorials/fundamentals/part-1-overview



LLM Introduction

What is an LLM?

- A probabilistic text model that predicts the next token
- Trained on large dataset makes it better at learning statistical patterns & world knowledge
- **Transformer** is the dominant architecture
- Typical capabilities: Q&A, summarization, extraction, classification, coding
- **Examples:** GPT-class models, Llama, Gemma, Mistral, etc.

What is Tokenization and Embeddings

Tokenization = split text into model-friendly pieces (tokens) and map them to IDs.

- Why: models work on integers, not raw text; subwords (e.g., BPE) avoid OOV.
- Example (BPE): “Transformers are amazing!” → tokens like `["Transform", "ers", " are", " amazing", "!"]`

- **Embeddings = look up (or compute) a dense vector for each token/word/sentence.**
- Why: vectors let us measure meaning/similarity with math (dot product / cosine).
- Example (token embeddings): each token ID indexes an embedding table → `["Transform", "ers", " are", " amazing", "!"]` → five vectors like `[0.12, -0.08, …, 0.31]` (say 768-D).
  The model’s first layer turns the ID sequence into this matrix of vectors; downstream layers use them for attention.

Tokenization & BPE (Byte Pair Encoding)

```
# pip install tiktoken
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")
text = "Transformers are amazing!"
ids = enc.encode(text)
print(ids)
print(enc.decode(ids))
```

Output (example):

```
[16984, 481, 257, 14296, 0]
Transformers are amazing!
```

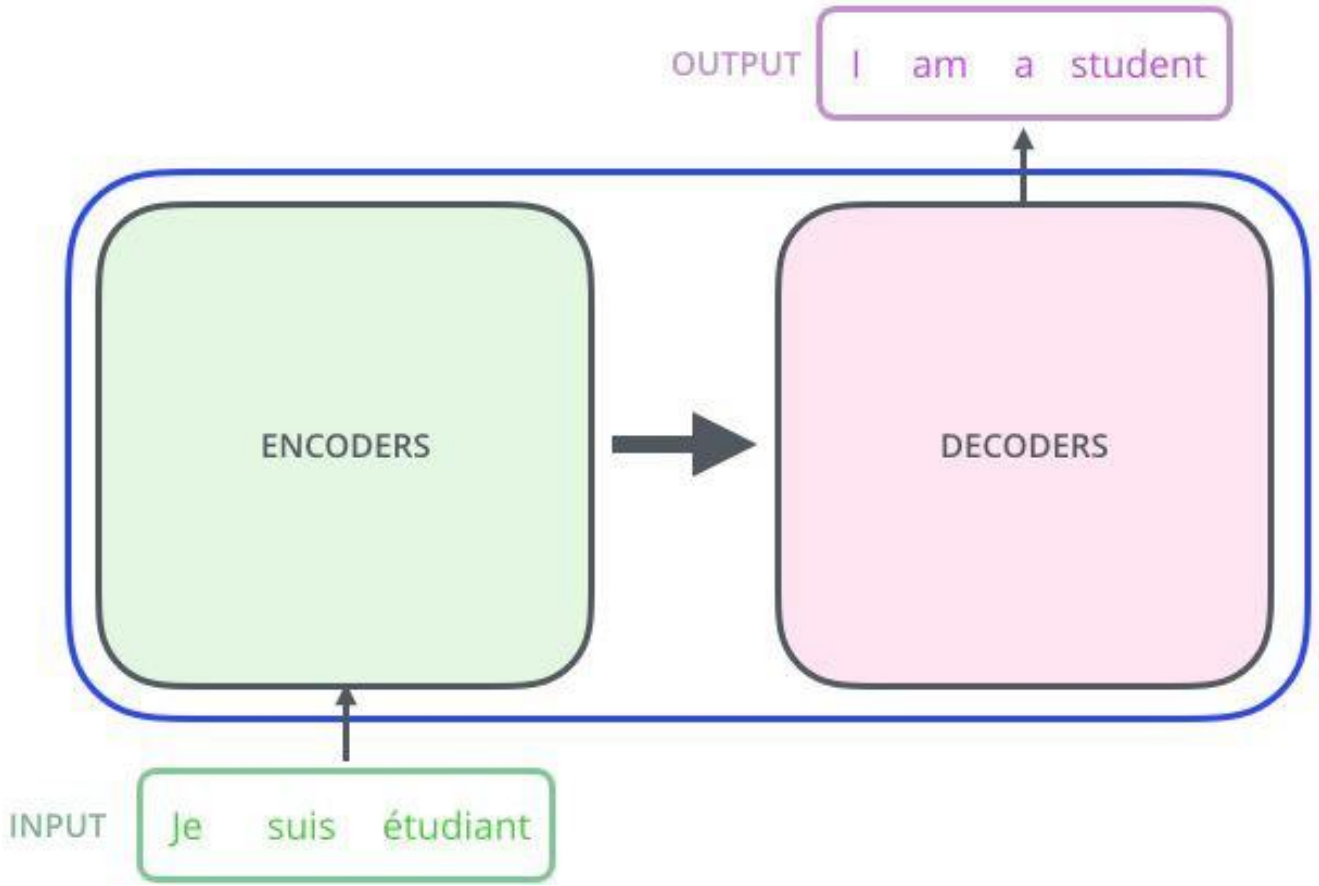
Key idea: **subword units** reduce OOV issues and keep vocab compact.
OOV = out-of-vocabulary—a word your tokenizer/model hasn't seen in its fixed vocabulary (classic word-level systems map it to an <unk> token).

Transformers at a glance

- End-to-end attention; no recurrence or convolutions

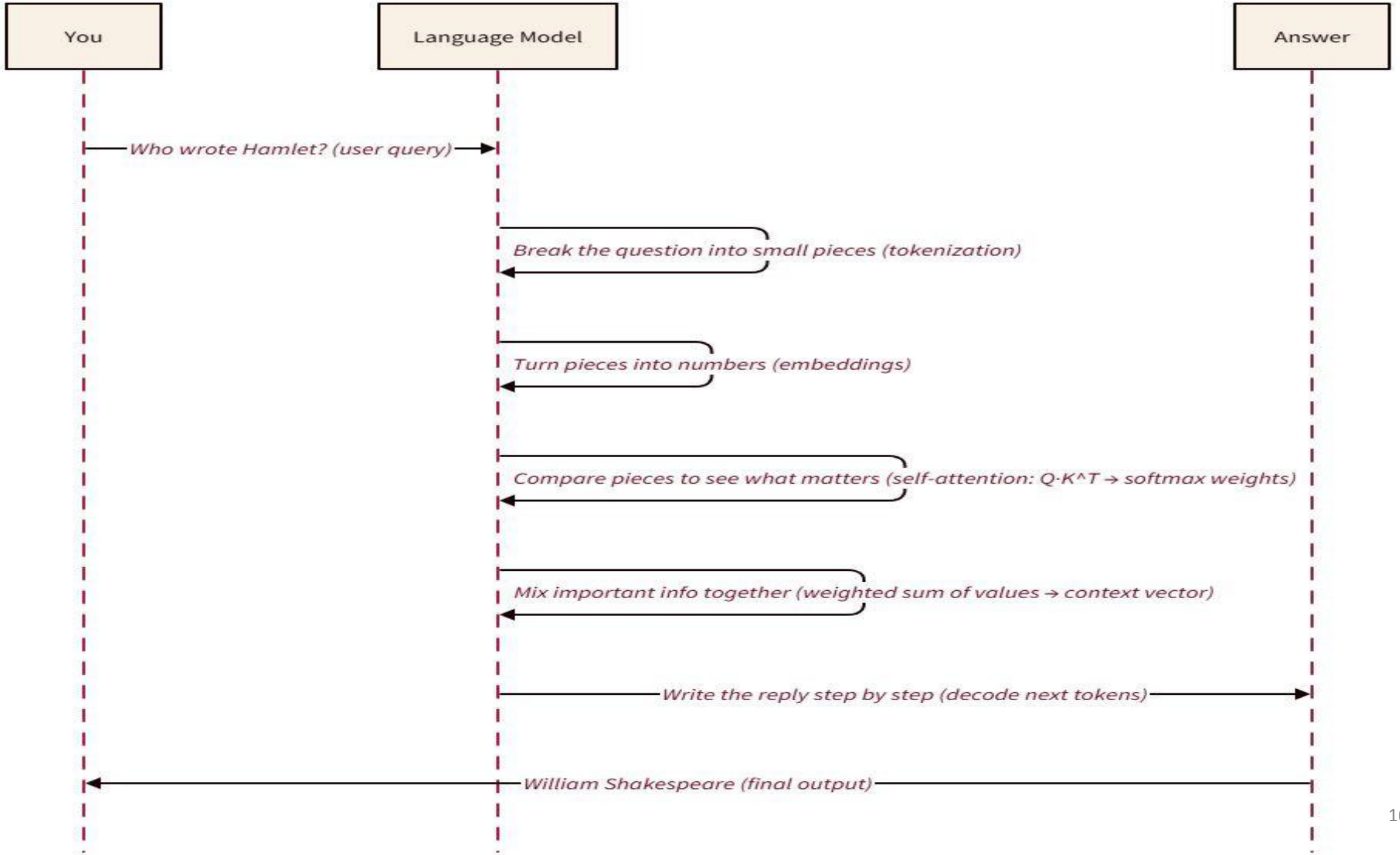
  Highly parallelizable → faster training & inference

Self-attention intuition



Scaled dot-product attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$


You
Language Model
Answer
Who wrote Hamlet? (user query)
Break the question into small pieces (tokenization)
Turn pieces into numbers (embeddings)
Compare pieces to see what matters (self-attention: Q·K^T → softmax weights)
Mix important info together (weighted sum of values → context vector)
Write the reply step by step (decode next tokens)
William Shakespeare (final output)

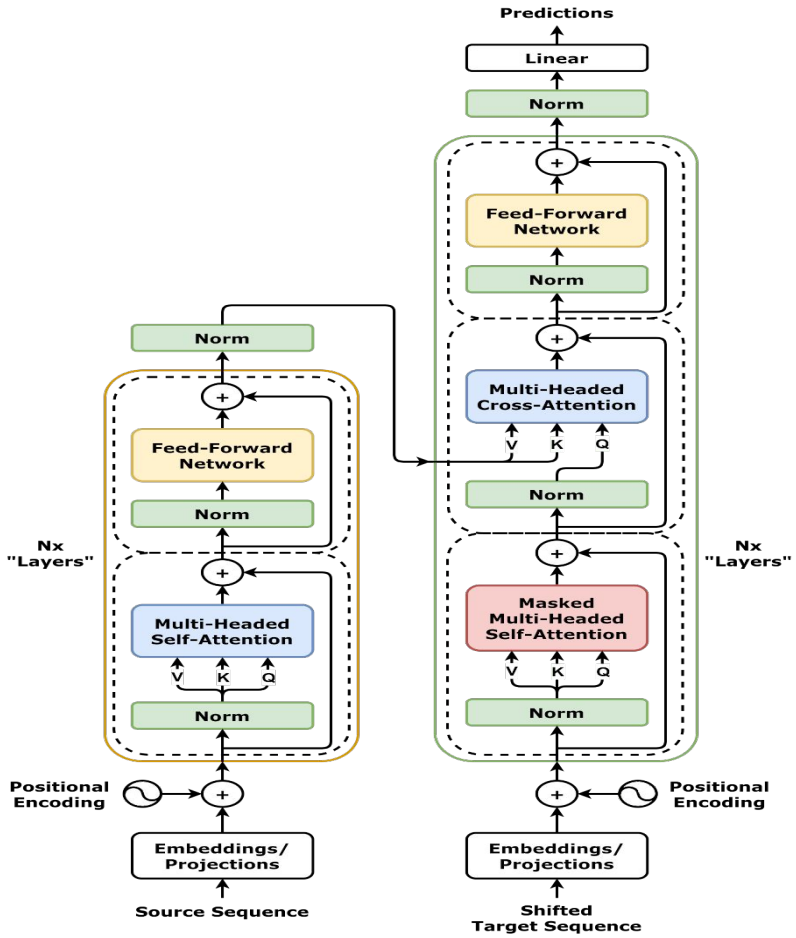
Multi-head attention

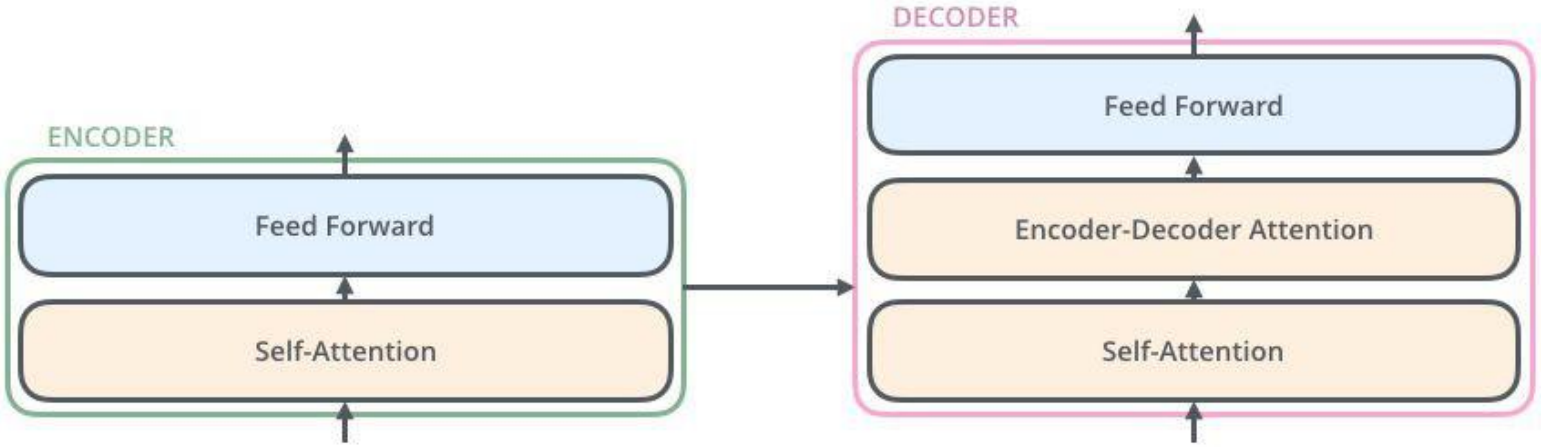
Multiple attention heads = multiple representational subspaces

Heads attend to different patterns (syntax, long-range deps, positions)

```
[Head 1] attends to nouns
[Head 2] attends to positional cues
[Head 3] attends to verb–object relations
```

Encoder vs Decoder vs Encoder-Decoder





- Encoder-only = reads everything at once (bidirectional) → great for understanding.
- Decoder-only = writes one token at a time (causal) → great for generating.
- Encoder-Decoder = reads with an encoder, then writes with a decoder that can look back at what it read → great for sequence-to-sequence (translate, summarize).

- **Encoder-only** (e.g., BERT): representations for classification/extraction
- **Decoder-only** (e.g., GPT): left-to-right generation
- **Encoder-Decoder** (e.g., T5): sequence-to-sequence

FastAPI Agents

if you've built APIs in Node.js (Express/Fastify/Nest), FastAPI will feel very familiar. The patterns—**routing, middleware, request/response lifecycle, async I/O, validation**—are the same; you'll just use Python syntax, type hints, and Pydantic.

What is FastAPI?

FastAPI is a modern, high-performance Python (3.7+) framework for building APIs, using type hints to drive validation and auto-generate OpenAPI/Swagger docs.

https://asgi.readthedocs.io/en/latest/

Purpose: Ship APIs quickly and efficiently.

Key features:

- Speed: Built on Starlette (ASGI) + Pydantic (is a Python library for data validation and parsing).
- Type-driven: Type hints → validation & docs.
- Async-native: `async/await` for high concurrency.
- Auto docs: It automatically generates interactive API documentation using OpenAPI and Swagger UI.
- Data integrity: Pydantic models enforce schemas.
- Easy to learn: Clean, minimal code to get started.

FastAPI

FastAPI framework, high performance, easy to learn, fast to code, ready for production

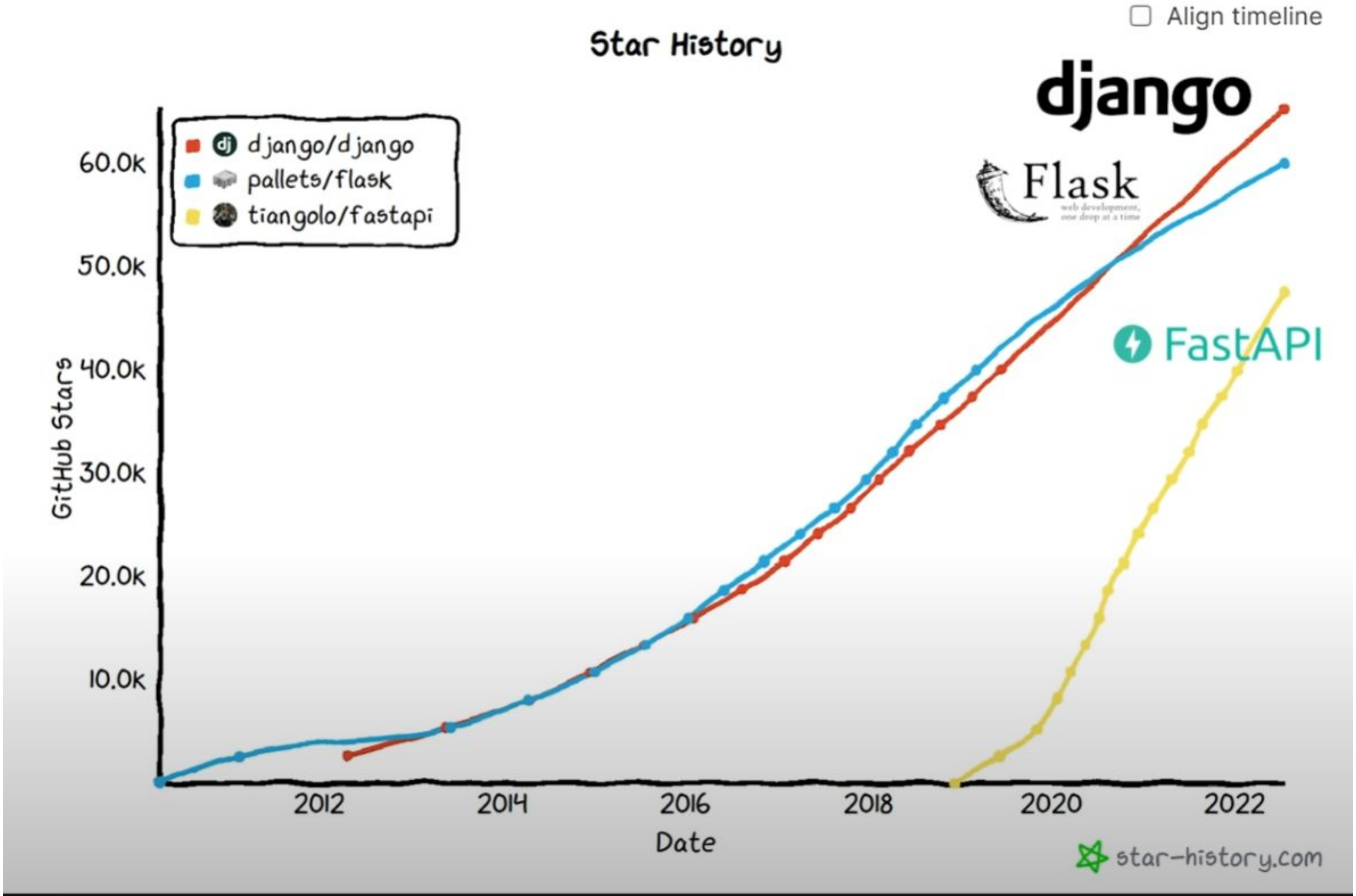
Documentation: https://fastapi.tiangolo.com

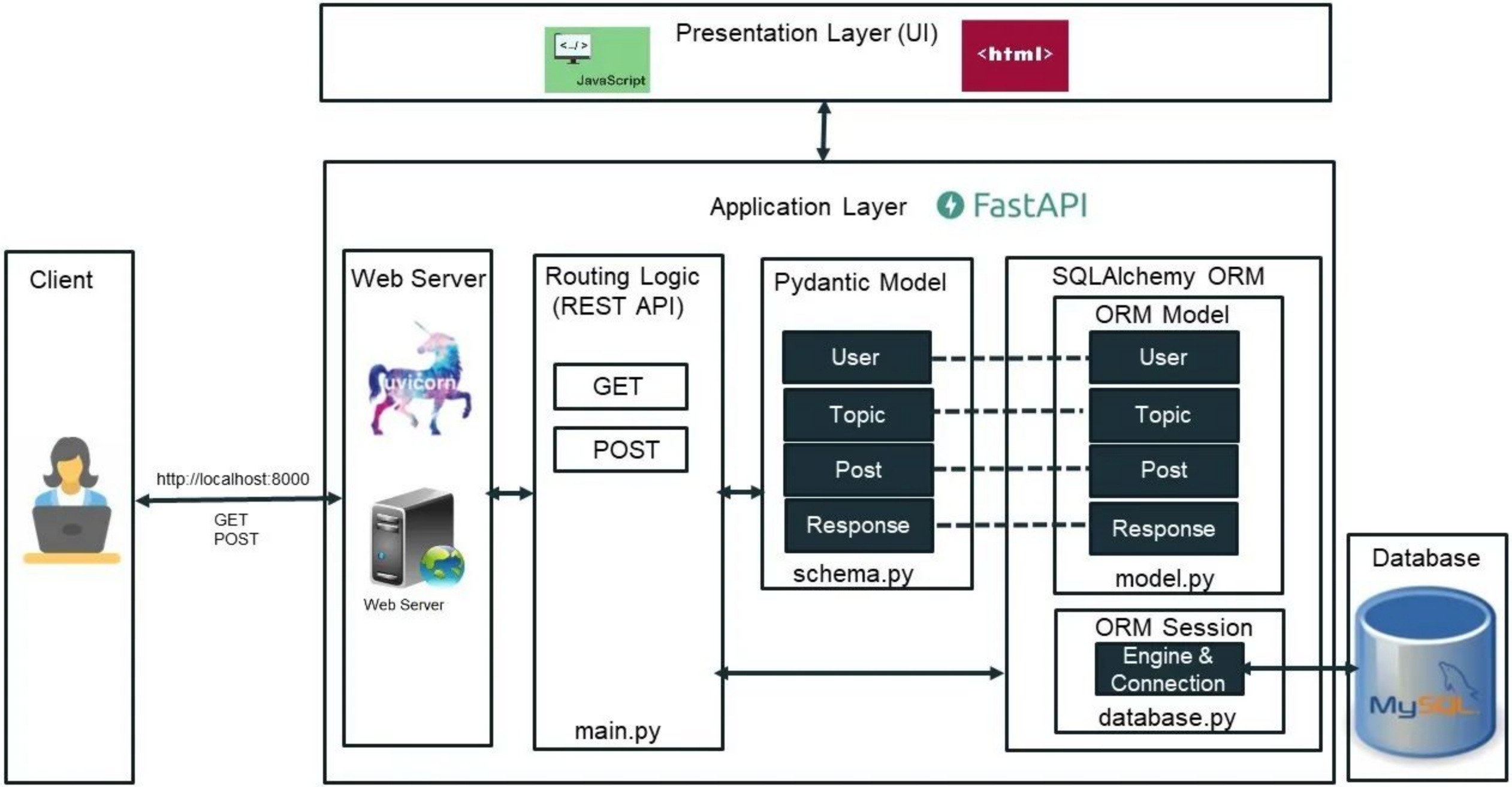
Source Code: https://github.com/fastapi/fastapi

FastAPI is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints.

The key features are:

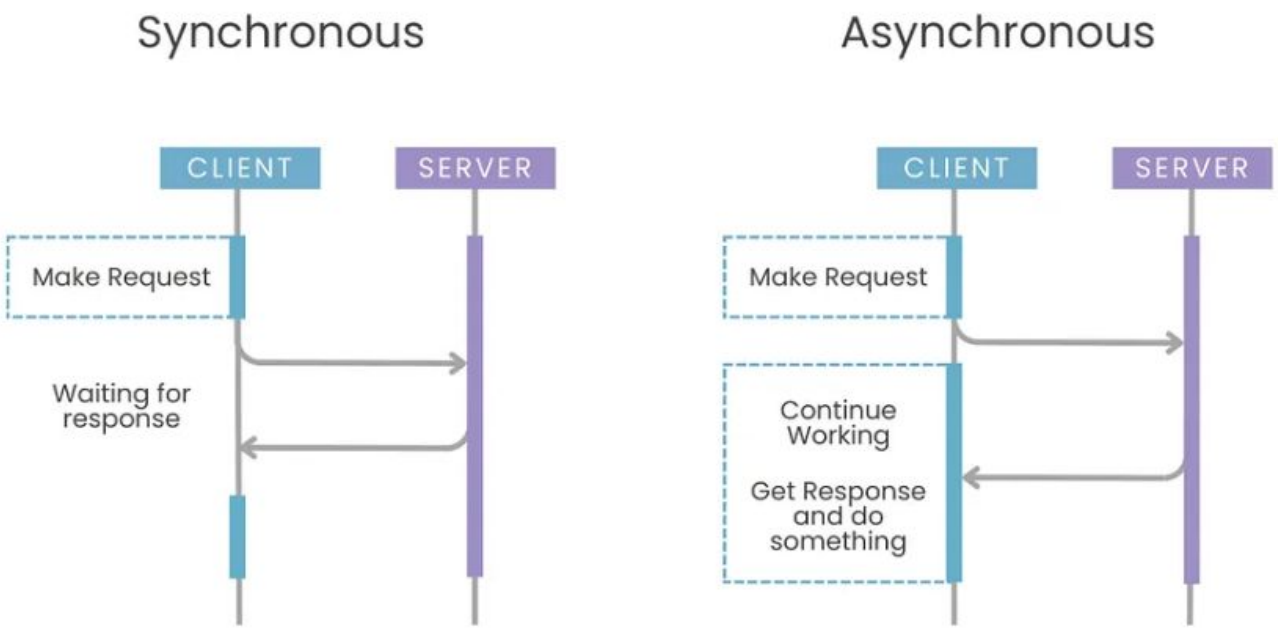
- **Fast**: Very high performance, on par with **NodeJS** and **Go** (thanks to Starlette and Pydantic). One of the fastest Python frameworks available.
- **Fast to code**: Increase the speed to develop features by about 200% to 300%. *
- **Fewer bugs**: Reduce about 40% of human (developer) induced errors. *
- **Intuitive**: Great editor support. Completion everywhere. Less time debugging.
- **Easy**: Designed to be easy to use and learn. Less time reading docs.
- **Short**: Minimize code duplication. Multiple features from each parameter declaration. Fewer bugs.
- **Robust**: Get production-ready code. With automatic interactive documentation.
- **Standards-based**: Based on (and fully compatible with) the open standards for APIs: OpenAPI [↪] (previously known as Swagger) and JSON Schema [↪].


Align timeline
Star History
django
Flask
web development, one drop at a time
FastAPI
django/django
pallets/flask
tiangolo/fastapi
60.0k
50.0k
40.0k
30.0k
20.0k
10.0k
GitHub Stars
2012
2014
2016
2018
2020
2022
Date
star-history.com


Presentation Layer (UI)
JavaScript
<html>
Application Layer
FastAPI
Client
http://localhost:8000
GET
POST
Web Server
uvicorn
Web Server
Routing Logic
(REST API)
GET
POST
main.py
Pydantic Model
User
Topic
Post
Response
schema.py
SQLAlchemy ORM
ORM Model
User
Topic
Post
Response
model.py
ORM Session
Engine &
Connection
database.py
Database
MySQL

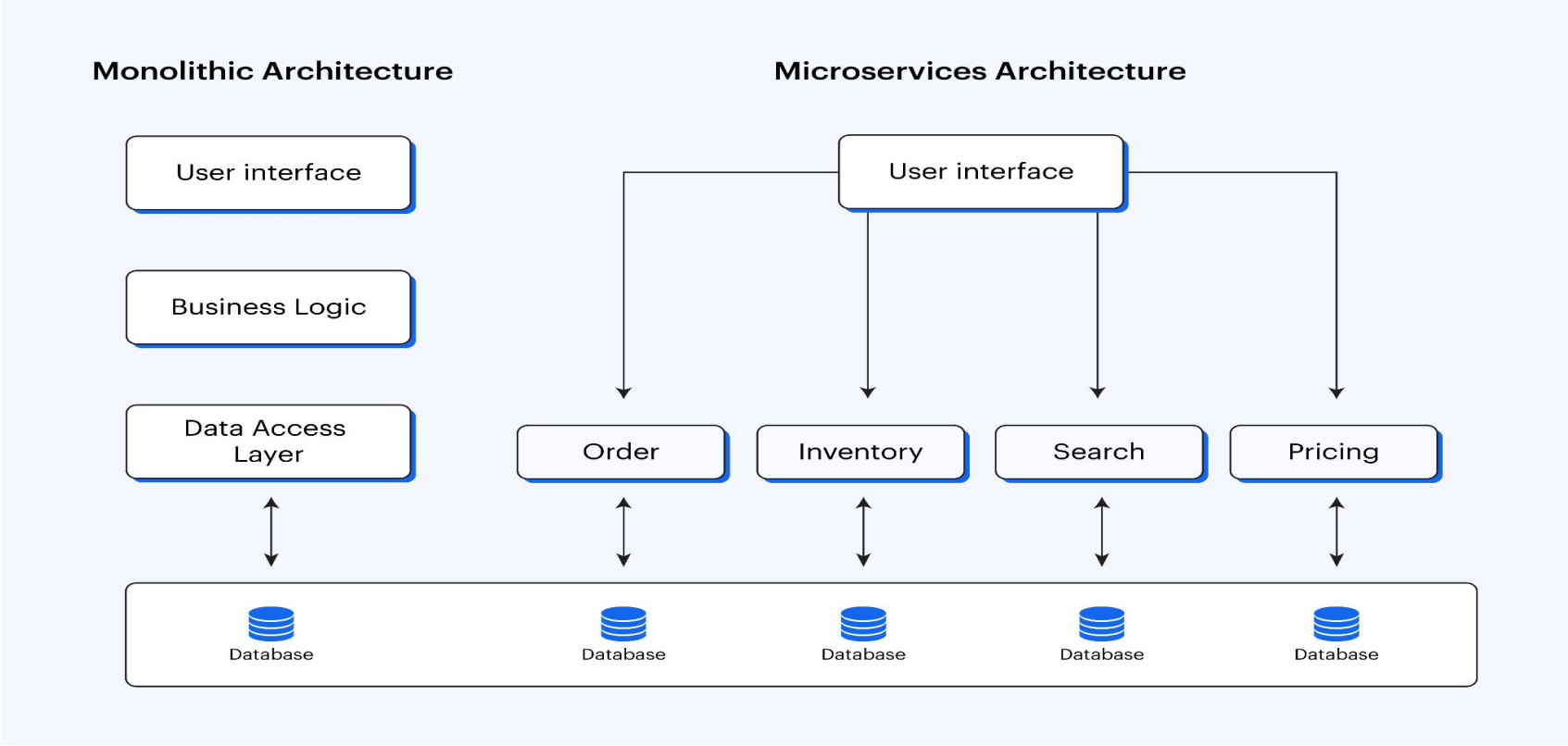
Ideal for Microservices Architecture

- Lightweight and modular
- Async support for concurrency
- Easy to containerize and deploy
- Works well with CI/CD pipelines



Microservices

- Microservice Independence: Every microservice needs its codebase and database, building loosely coupled services that can be updated without relying on the other components.
- Domain-Driven Design: By leveraging Domain-Driven Design principles, you can be sure that every microservice closely aligns with specific business domains, minimising unnecessary inter-service communication and encouraging cohesion within every service

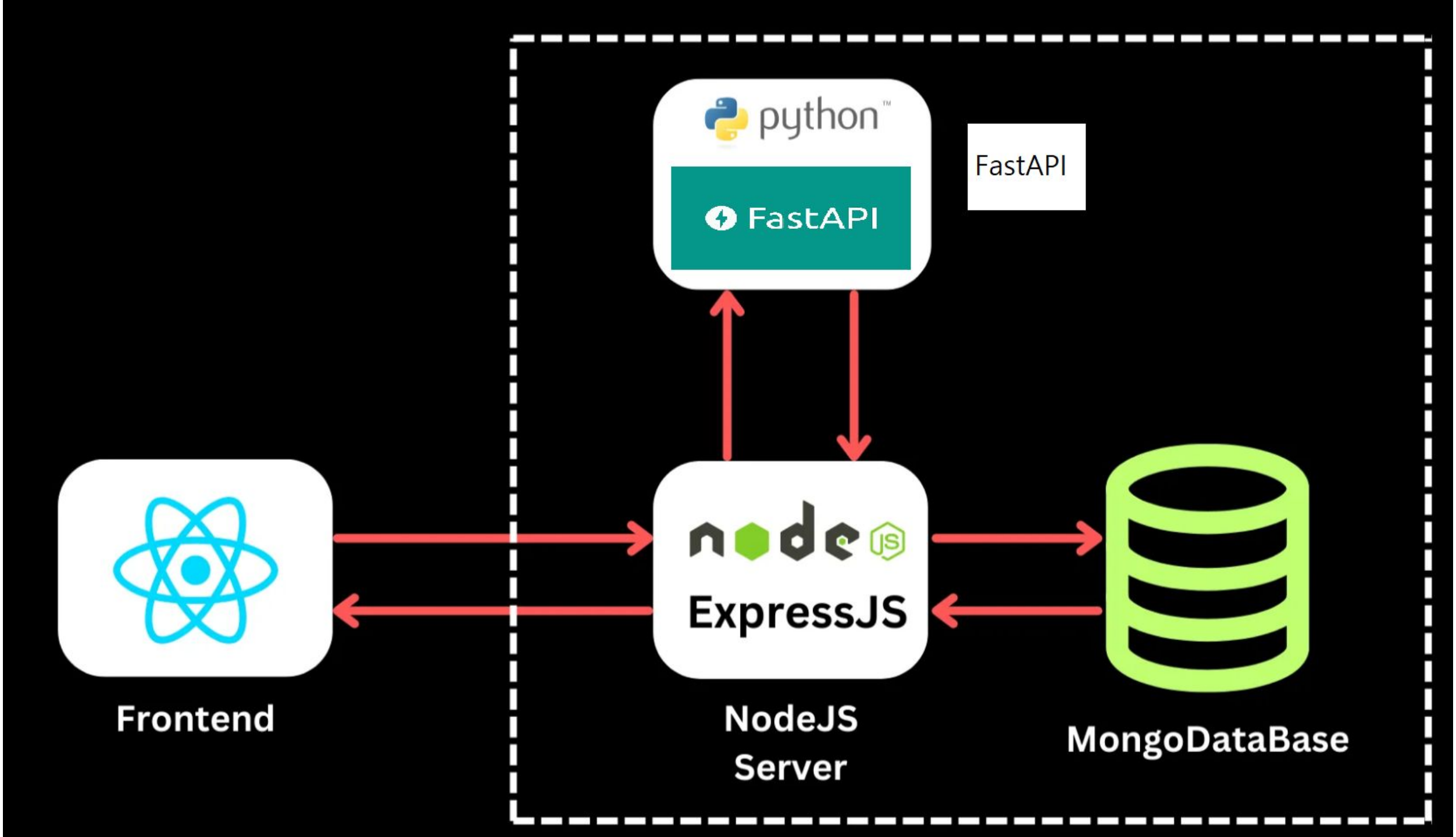


Why FastAPI for ML Projects

- Easy integration with ML models
- Supports background tasks
- Scalable from prototype to production
- Compatible with scikit-learn, TensorFlow, PyTorch, etc.
- Creating a separate Python ML service:
- Develop your ML model using Python and libraries such as TensorFlow or scikit-learn.
- Save the trained model in a format like .h5 or .pkl.

- **Hybrid**: Keep Node.js for the web layer/realtime, call a FastAPI microservice for model inference via gRPC/HTTP.
- Use frameworks like Flask or FastAPI to create an API around your model.
- Call the Python API from your Node.js application to get predictions.

- **FastAPI**: Handles the **ML model serving**, since it's great for:
  - High-performance APIs
  - Python ecosystem access (NumPy, pandas, TensorFlow, PyTorch, etc.).
  - Lightweight deployment of trained models via REST endpoints.
- **Node.js**: Acts as the **frontend/backend server** (or API gateway), handling:
  - Web server logic, authentication, sessions.
  - Serving the frontend (e.g., React, Vue, etc.).
  - Interacting with databases, logging, and other services.
  - Making requests to FastAPI endpoints to retrieve predictions.


python
FastAPI
FastAPI
node
JS
ExpressJS
Frontend
NodeJS
Server
MongoDataBase

Code Demo (Hello World)

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hello")
def read_hello():
  return {"message": "Hello, world!"}
```

Why FastAPI + Agents

What you gain

Async-first (ASGI) → concurrent tool calls & non-blocking I/O.

Typed contracts with Pydantic → strict input/output for tools, safer prompts.

Auto OpenAPI docs (Swagger UI / ReDoc) for quick testing & sharing.

WebSockets/streaming → token-by-token updates to your UI.

FastAPI vs http.server & low-level HTTP

Why not http.server for agents?

- Not recommended for production; only basic security checks.
- No request validation or type-safe models; you hand-roll JSON parsing/errors.
- No automatic API docs / schema → harder to test tools.
- No built-in WebSockets, DI, or background tasks.

ASGI > WSGI for Agents (real-time & concurrency)

ASGI supports long-lived connections (WebSockets) and true async I/O — ideal for streaming tokens/tools.

WSGI (e.g., classic Flask setups) was designed pre-async; async often runs in a thread and doesn't unlock full concurrency.

With FastAPI+ASGI you can multiplex tool calls and stream results.

Easier to manage agent runtimes

Developer superpowers

Dependency Injection for keys, vector clients, tool registries.

BackgroundTasks for logging, telemetry, callbacks after response.

Middleware & CORS for auth/rate-limits and front-end apps.

Pydantic models & validation

```python
from pydantic import BaseModel, Field

class AskRequest(BaseModel):
    question: str = Field(min_length=3, examples=["What is attention?"])

class AskResponse(BaseModel):
    answer: str
```

```python
@app.post("/ask", response_model=AskResponse)
async def ask(req: AskRequest):
    return {"answer": f"You asked: {req.question}"}
```

FastAPI auto-generates request/response schemas and examples.

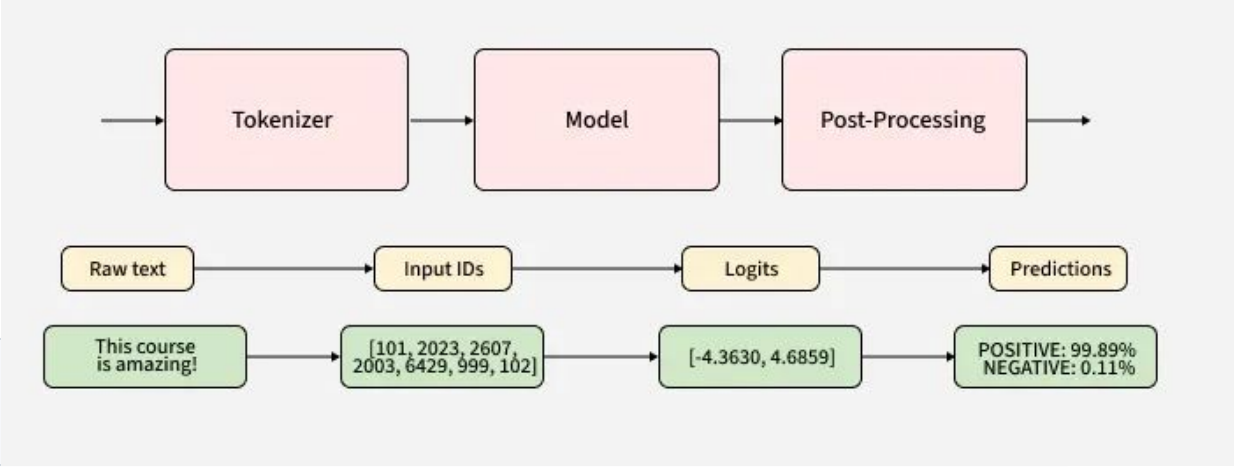
Async & running with Uvicorn

```
pip install "fastapi[standard]" uvicorn
uvicorn main:app --host 0.0.0.0 --port 8000
```

Tips

- Prefer async def for IO-bound work
- Configure timeouts & retries

Serve a Hugging Face model




```
pip install transformers torch
```

```
from transformers import pipeline
gen = pipeline("text-generation", model="gpt2")

@app.post("/generate")

async def generate(prompt: str):
    out = gen(prompt, max_length=40, num_return_sequences=1)[0]["generated_text"]
    return {"text": out}
```

request to try yourself:

```
curl -X POST localhost:8000/generate -H 'Content-Type: application/json' -d '"hello"'
```

Streaming responses (SSE)

Server
*:*from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import asyncio
app = FastAPI()
# Async generator that yields Server-Sent Event (SSE) chunks
async def stream_tokens(prompt: str):
    for token in ["This ", "is ", "streamed ", "text."]:
        # Each SSE message must end with a blank line (\n\n)
        yield f"data: {token}\n\n"
        await asyncio.sleep(0.2)

@app.get("/stream")
async def stream(prompt: str):
    return StreamingResponse(stream_tokens(prompt), media_type="text/event-stream"   )

Client (browser):

```
<script>
const es = new EventSource('/stream?prompt=hi');
es.onmessage = (e) => console.log(e.data);
</script>
```

Output

```
{"token": "This "}
{"token": "is "}
{"token": "streamed "}
{"token": "text."}
```

What is an Agent? (Revision)

A system that **uses an LLM to decide actions** (tools) toward a goal

Tools can be: search, DB queries, code execution, APIs …

ReAct pattern: interleave **Reasoning** and **Acting loop**

```
Thought → Action(tool=search, input="...") → Observation → (repeat) → Final Answer
```

Mini example (toy)

User goal: “Find the count of ‘refund’ tickets last week and give a one-line insight.”

- **Thought:** Need last week’s tickets; then filter by “refund”; then summarize.
- **Action:** `tool=sql_query, input="SELECT created_at, title FROM tickets WHERE created_at BETWEEN '2025-09-15' AND '2025-09-21';"`
  Observation: 3,412 rows returned.
- **Thought:** Count titles containing “refund”.
- **Action:** `tool=code_exec, input="count = sum('refund' in t.lower() for t in titles); print(count)"`
- **Observation:** `987`
- **Final Answer:** “987 refund-related tickets last week; spike followed a pricing change—consider adding clearer refund terms on checkout.”

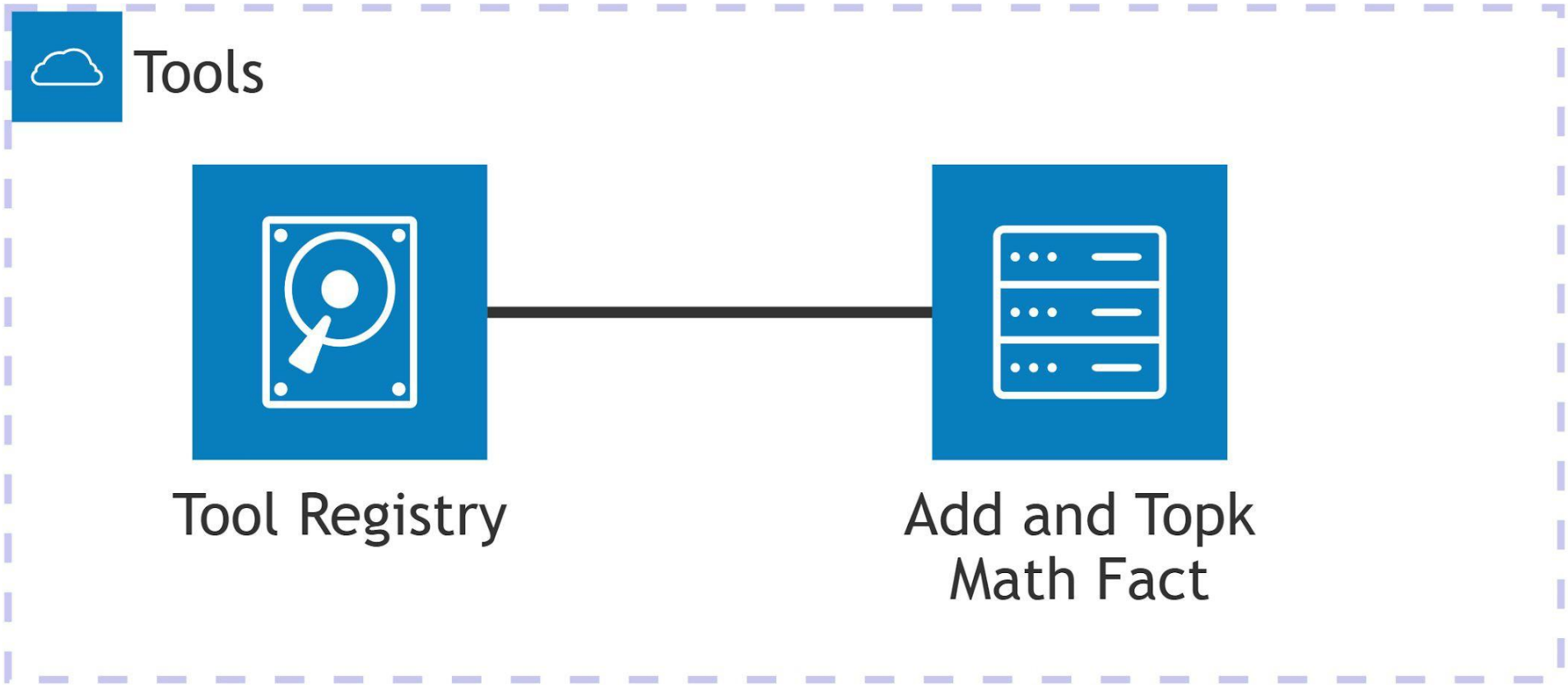
Tool calling basics (schema)

- Define tools (name, description, JSON schema for args)
- Let the model select & call tools; your server executes them
- Return tool **observations** to the model for next step

```
{ "type": "function",
  "function": {
    "name": "search_web",
    "description": "Search the web and return a snippet",
    "parameters": {
      "type": "object",
      "properties": {"query": {"type": "string"}},
      "required": ["query"]
    }
  }
```

Build a tiny Agent (tools)

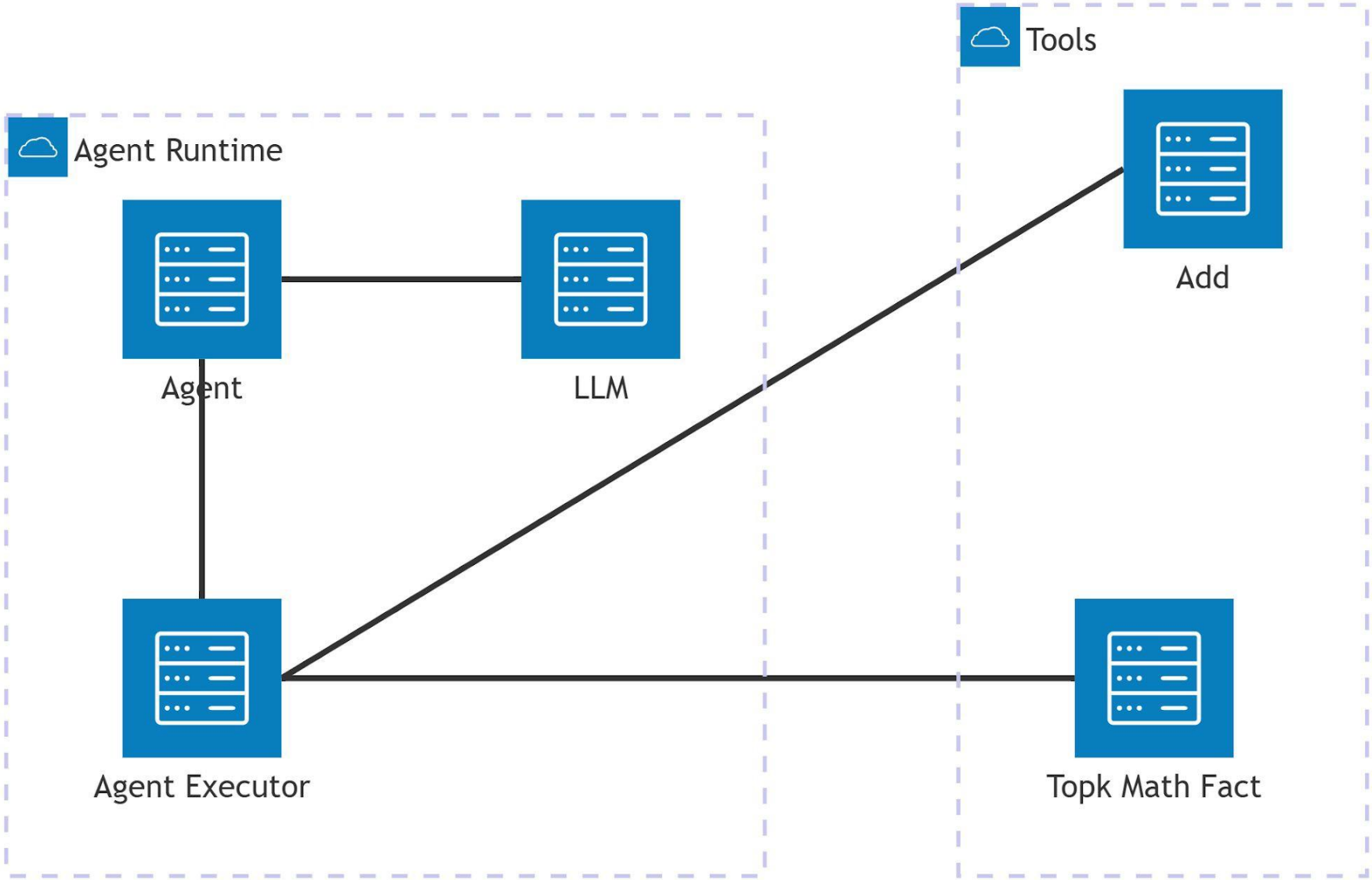
```
from langchain_core.tools import tool

@tool
def add(x: int, y: int) -> int: """Add two integers.""" return x + y

@tool
def topk_math_fact(topic: str) -> str: """Return a fake 'fact' for demo."""
    return f"Fun fact about {topic}: 42"

TOOLS = [add, topk_math_fact]
```


Tools
Tool Registry
Add and Topk
Math Fact

Create the Agent executor

```
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    ("user", "{input}"),
    MessagesPlaceholder("agent_scratchpad"),
])
llm_with_tools = llm.bind_tools(TOOLS)

agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x: format_to_openai_tool_messages(x["intermediate_steps"]),
    }   | prompt | llm_with_tools | OpenAIToolsAgentOutputParser()
)

agent_executor = AgentExecutor(agent=agent, tools=TOOLS, verbose=True)
```


agent_executor.invoke({"input": "Add 2 and 3, then give me a fun math fact."})


Tools
Agent Runtime
Add
Agent
LLM
Agent Executor
Topk Math Fact

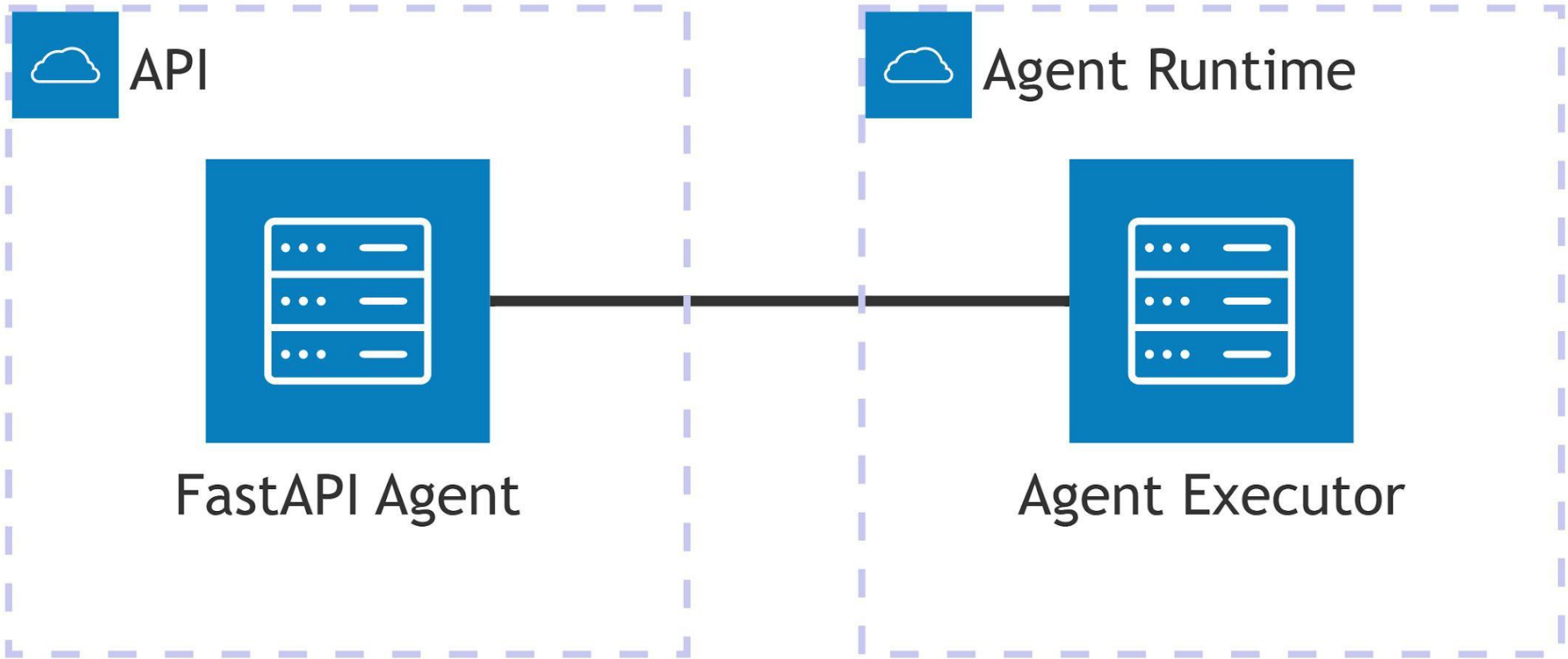
Wrap the Agent in FastAPI

```python
from pydantic import BaseModel

class AgentRequest(BaseModel): input: str

@app.post("/agent")
async def run_agent(req: AgentRequest):
    result = await agent_executor.ainvoke({"input": req.input})
    return {"output": result["output"]}
```

Request → `{ "input": "What is 2+3? Use add." }`

Response →

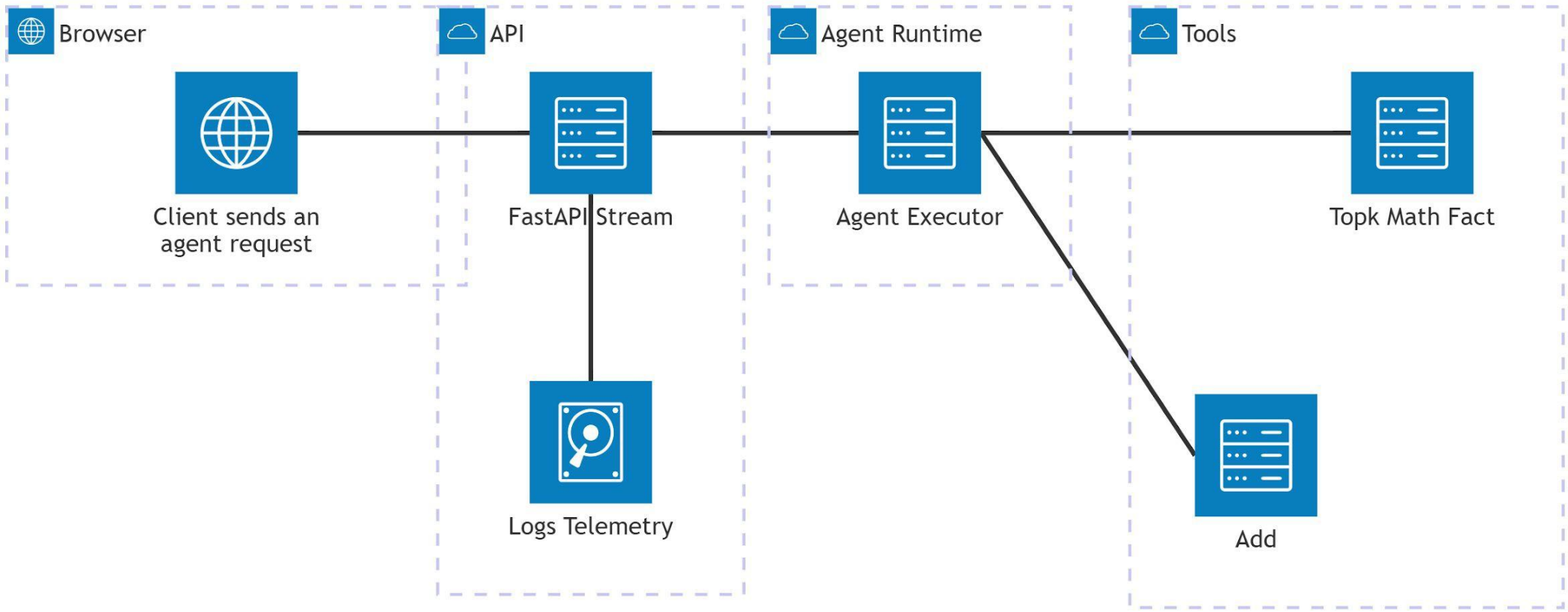
```
{"output": "2 + 3 = 5"}
```


API
Agent Runtime
FastAPI Agent
Agent Executor

Stream the Agent (SSE)

```
from typing import AsyncIterable

async def agent_events(prompt: str) -> AsyncIterable[str]:
    async for ev in agent_executor.astream_events({"input": prompt}):
        yield f"data: {ev}\n\n"

@app.get("/agent/stream")
async def agent_stream(input: str):
    return StreamingResponse(agent_events(input), media_type="text/event-stream")
```

Client receives intermediate tool calls & final answer incrementally.


Browser
API
Agent Runtime
Tools
Client sends an agent request
FastAPI Stream
Agent Executor
Topk Math Fact
Logs Telemetry
Add

Testing & docs

Interactive docs at **/docs** (Swagger UI) and **/redoc**

Auto-generated schemas & examples from Pydantic

Use pytest + httpx.AsyncLient for endpoint tests

```
from httpx import AsyncClient

async def test_root():
    from main import app
    async with AsyncClient(app=app, base_url="http://test") as ac:
        r = await ac.get("/")
    assert r.json()["message"] == "Hello, FastAPI"
```