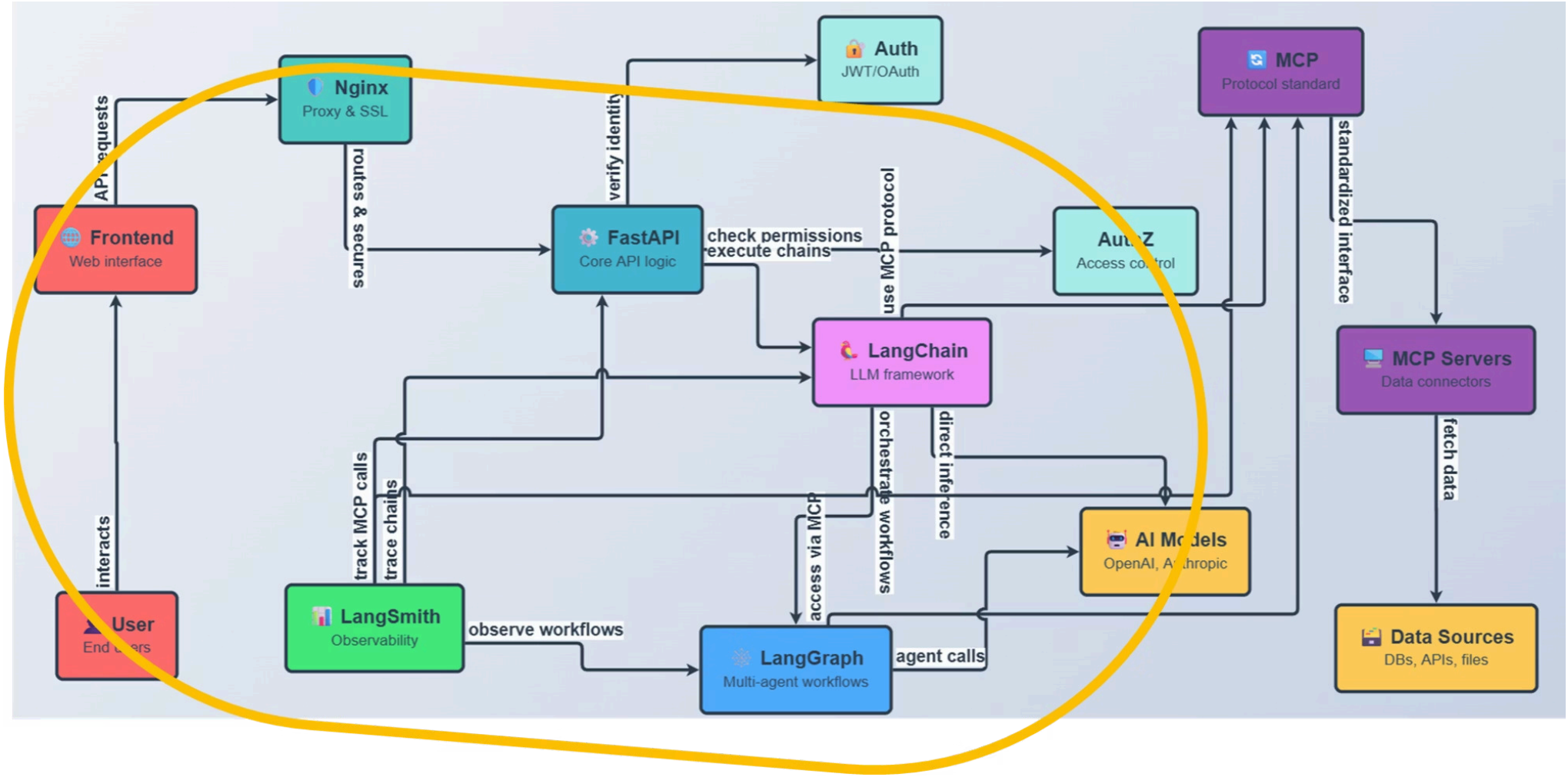
AI Agent Design II: from LangChain to LangGraph


Nginx
Proxy & SSL
Auth
JWT/OAuth
MCP
Protocol standard
Frontend
Web interface
FastAPI
Core API logic
check permissions
execute chains
LangChain
LLM framework
MCP Servers
Data connectors
LangSmith
Observability
observe workflows
LangGraph
Multi-agent workflows
agent calls
Data Sources
DBs, APIs, files
routes & secures
verify identity
standardized interface
fetch data
track MCP calls
trace chains
orchestrate workflows
direct inference
interacts

What is LangChain Expression Language?

LangChain Expression Language (LCEL) is a revolutionary way to chain AI components together using the intuitive pipe (|) operator. It transforms complex AI application development into modular, readable code that flows naturally from one component to the next.

Intuitive Design

Uses familiar pipe operator for chaining components together seamlessly

Modular Architecture

Break complex workflows into manageable, reusable components

Readable Code

Clear visual flow from prompts through models to parsed outputs

colab.research.google.com

Google Colab

LCEL (LangChain Expression Language)

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_01_lcel.ipynb

LangChain Agent

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_02_00_agent.ipynb

Core LCEL Components

Prompts

Templates that format your input data into structured requests for AI models. They handle variable substitution and ensure consistent formatting across different use cases.

Models

AI models like GPT-4 that process the formatted prompts and generate intelligent responses. The core reasoning engine of your application.

Parsers

Format the model's raw output into usable, structured data that your application can work with effectively.

Tools

External functions and APIs that the AI can call to perform actions beyond text generation, extending capabilities significantly.

The Runnable Protocol

Every LangChain component implements a standard set of methods, ensuring consistency and predictability across your entire application architecture.

invoke()

Process a single input synchronously

batch()

Process multiple inputs simultaneously for efficiency

stream()

Get real-time responses as they're generated

Each method also has asynchronous versions (ainvoke(), abatch(), astream()) for non-blocking operations in concurrent applications.

Building Your First Chain

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Create components
prompt = ChatPromptTemplate.from_template("Tell me a joke about {topic}")
model = ChatOpenAI()
parser = StrOutputParser()

# Chain them together
chain = prompt | model | parser

# Use it
result = chain.invoke({"topic": "cats"})
```

This elegant syntax demonstrates the power of LCEL - complex AI workflows become as simple as connecting pipes, with each component handling its specific responsibility in the processing pipeline.

How the Pipe Operator Works

Normal Python Usage

In standard Python, the pipe operator performs bitwise OR operations:

```
5 | 3 = 7  # bitwise OR
```

LangChain Magic

LangChain overloads the __or__ method to create intelligent chains:

```
prompt | model = chain
```

This operator overloading transforms the familiar pipe symbol into a powerful tool for building AI workflows, making the code both intuitive and expressive for developers.

Advanced LCEL Features

1

Parallel Execution

Run multiple chains simultaneously using RunnableParallel for improved performance and comprehensive responses.

```
parallel_chain = RunnableParallel(
    joke=joke_chain,
    fact=fact_chain
)
```

2

Fallback Systems

Implement robust error handling with automatic fallbacks to backup models when primary systems fail.

```
chain_with_fallback = primary_model.with_fallbacks([backup_model])
```

3

RAG Pattern

Retrieval-Augmented Generation combines document retrieval with generation for informed, contextual responses.

```
rag_chain = retriever | prompt | model | parser
```

Agent Core trick: json and pydantic

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_02_01_jsonValidation.ipynb

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_02_02_JsonHuggingFace.ipynb

Introducing LangGraph

Whilst LangChain excels at simple chains, LangGraph revolutionises AI applications by adding sophisticated state management and complex workflow capabilities that enable truly intelligent, context-aware systems.

Memory

Remembers past conversations and context across interactions

Multi-step Workflows

Complex decision making with branching logic and conditional paths

Human Approval

Pause workflows for user input and oversight when needed

Persistence

Save and resume conversations across sessions and system restarts

LangGraph

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_03_langgraph.ipynb

Understanding LangGraph with Phone Analogies

THREAD = Individual Chat

Like texting with different friends - each conversation is separate and isolated, identified by a unique thread_id for perfect organisation.

STATE = Conversation History

Like your entire chat history with one person - remembers what was discussed before and can store custom data like user preferences.

PERSISTENCE = Saving Messages

Like your phone saving messages when you close the app - MemorySaver for temporary storage, SqliteSaver for permanent database storage.

STREAMING = Real-time Updates

Like seeing typing indicators - watch the AI think step-by-step as it processes and responds to your requests.

Building a Basic LangGraph Agent

```
from langgraph.graph import StateGraph, MessagesState, START, END
from langgraph.prebuilt import ToolNode

def should_continue(state):
 last_message = state["messages"][-1]
 if last_message.tool_calls:
 return "tools" # Use tools
 return END # Done

def call_model(state):
 response = model.bind_tools(tools).invoke(state["messages"])
 return {"messages": [response]}

# Build the workflow
workflow = StateGraph(MessagesState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", ToolNode(tools))
workflow.add_edge(START, "agent")
workflow.add_conditional_edges("agent", should_continue, ["tools", END])
workflow.add_edge("tools", "agent")

agent = workflow.compile()
```

Managing Multiple Users with Threads

Separate Conversations

```
# Different users, different threads
config_sarah = {"configurable": {"thread_id": "user_sarah"}}
config_john = {"configurable": {"thread_id": "user_john"}}

# Sarah's Paris conversation
agent.invoke({"messages": [HumanMessage("Plan a trip to
Paris")]}, config_sarah)

# John's separate Tokyo conversation
agent.invoke({"messages": [HumanMessage("Plan a trip to
Tokyo")]}, config_john)
```

Perfect Isolation

Each user maintains their own conversation history and context. Sarah's Paris planning never interferes with John's Tokyo research, ensuring privacy and relevance.

Thread management scales effortlessly from individual users to enterprise applications with thousands of concurrent conversations.

Human-in-the-Loop Control

Add human oversight to AI workflows by pausing execution at critical decision points, ensuring safety and maintaining control over automated processes.

```
# Pause before using tools for approval
agent_with_approval = workflow.compile(
    checkpointer=memory,
    interrupt_before=["tools"]
)

# Execute and pause
response = agent_with_approval.invoke(input_data, config)
state = agent_with_approval.get_state(config)

if state.next:  # Paused for approval
    print(f"Agent wants to use: {state.values['messages'][-1].tool_calls}")
    approval = input("Approve? (y/n): ")
    if approval == 'y':
        agent_with_approval.stream(None, config)  # Continue
```

This pattern is essential for sensitive operations like financial transactions, data modifications, or external API calls requiring human validation.

When to Use Each Framework

Use LangChain (LCEL) when:

- Simple prompt → model → response flows
- Stateless operations
- Quick prototypes and experiments
- RAG applications with document retrieval
- Linear processing pipelines

Use LangGraph when:

- Multi-turn conversations with memory
- Complex decision workflows
- Human approval requirements
- Long-running processes
- State management across sessions

The frameworks complement each other perfectly - start simple with LangChain, then add LangGraph features as your application requirements become more sophisticated.

Real-World Application Patterns

1

Customer Support Bot

Classify issues → attempt resolution → escalate if needed. Maintains conversation history and tracks issue severity across multiple interactions.

2

Travel Planning Agent

Extract preferences → search flights → find hotels → create itinerary. Remembers budget constraints and dates throughout the planning process.

3

Content Creation Pipeline

Generate outline → conduct research → draft content → review quality → format output. Human approval at each critical step ensures quality control.

Graph State

https://github.com/enoch-sit/project-1-ipynb/blob/main/Tutorials/Tutorial_04_GraphState.ipynb