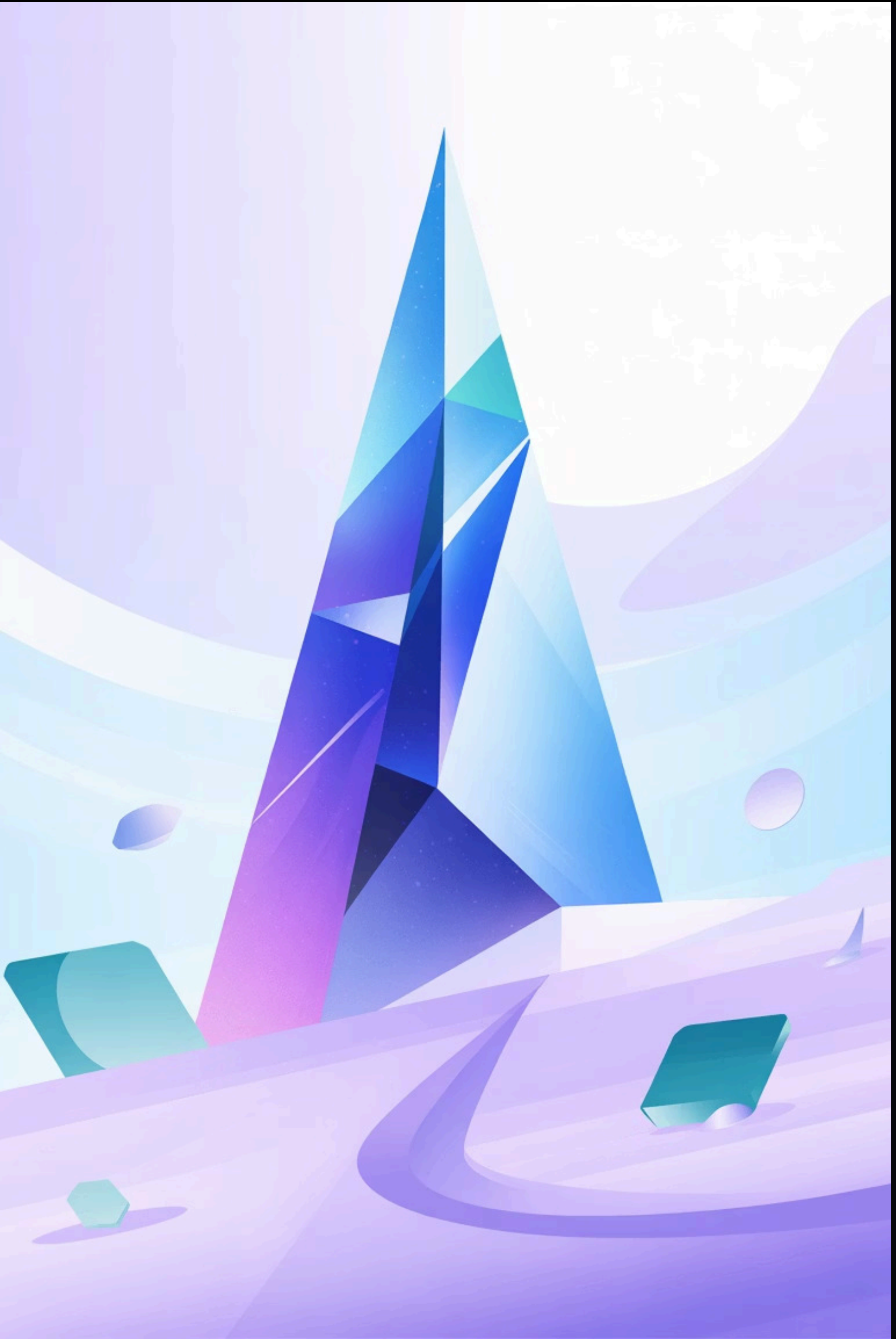


Agentic AI: Transforming LLMs into Autonomous Systems

A technical workshop on building intelligent AI agents with tool integration and state management

Workshop Agenda

1 The Problem & Solution

Understanding LLM limitations and how agentic AI addresses them

2 Technical Foundation

Exploring tool calling mechanisms and implementation patterns

3 LangGraph Deep Dive

Mastering orchestration, reasoning patterns, and state management

4 Hands-On Implementation

Building a fashion assistant agent with real-time capabilities

LLM Limitations: Why We Need Agents

Knowledge Cutoff

Static training data with no access to real-time information

No External Actions

Read-only systems that cannot interact with APIs or external tools

Stateless Nature

Limited memory between interactions, losing conversation context

Workflow Isolation

Inability to integrate with business systems and processes

What is Agentic AI?

AI systems that **autonomously take actions** to achieve specified goals

Tool Usage

Interacts with external systems, APIs, and databases

Multi-step Reasoning

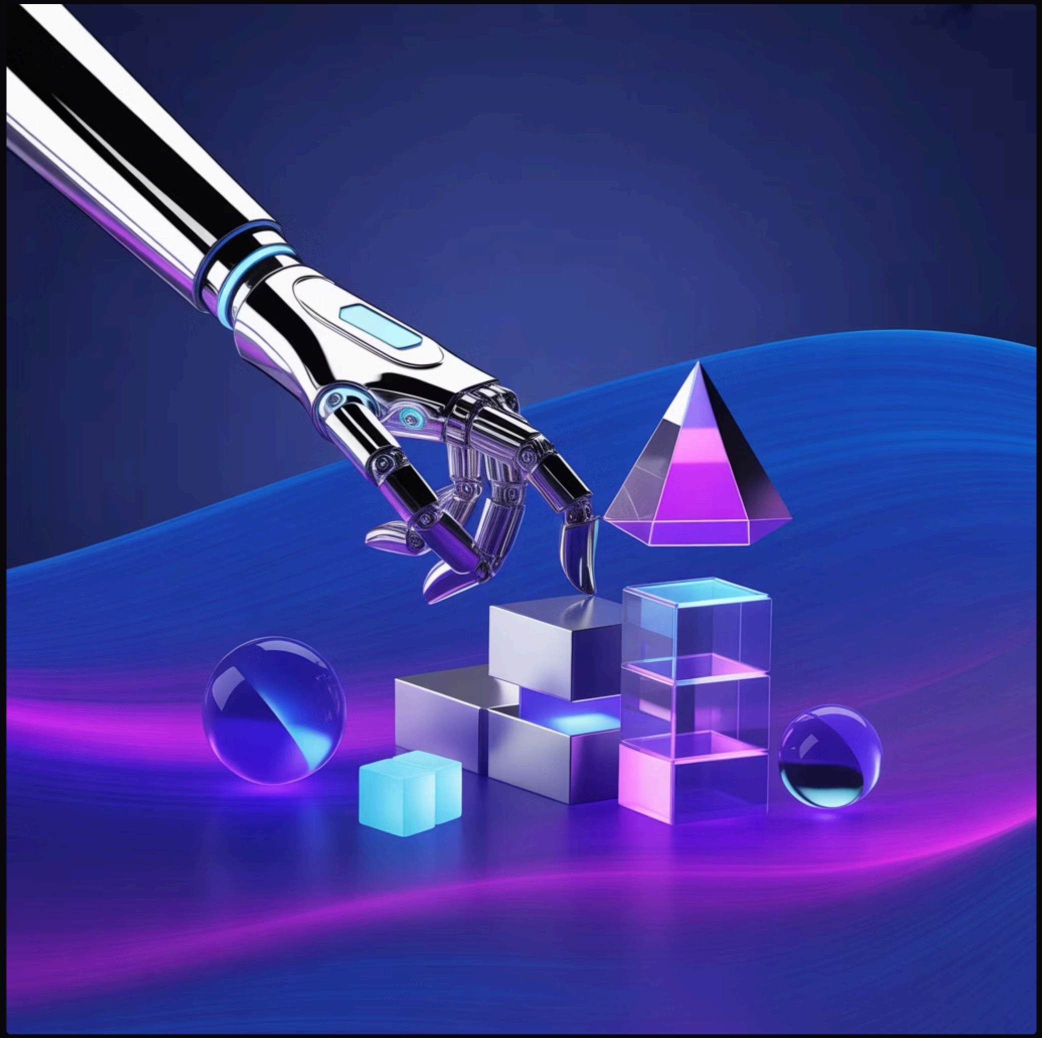
Plans and executes complex sequences of actions

Memory

Maintains context across interactions

Real-time Decisions

Adapts strategy based on current information



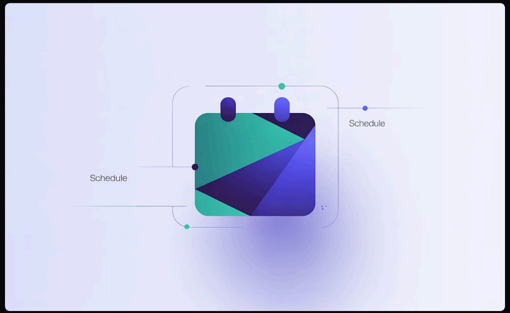
Chatbot vs. Agentic AI: A Comparison

Capability	Traditional Chatbot	Agentic AI
Information Access	Static (training data only)	Dynamic (real-time access)
Interaction Pattern	Single-turn responses	Multi-step reasoning & action
Output Capabilities	Text generation only	Text + external actions
System Integration	Limited or none	Deep integration with tools
Memory Management	Basic conversation history	Sophisticated state tracking
Problem Solving	Pattern matching	Goal-directed planning

Real-World Agent Applications

Customer Service

Order status tracking, automated refunds, intelligent escalation



Personal Assistants

Calendar management, travel booking, email prioritization

Business Automation

Data analysis, inventory management, lead qualification

Content Agents

Research, generation, publishing, A/B testing

Tool Calling: The Core Mechanism

The foundational capability that allows LLMs to interact with external systems

User Request

"What's the weather in Cluj-Napoca?"

LLM Analysis

Recognizes need for real-time data

Tool Selection

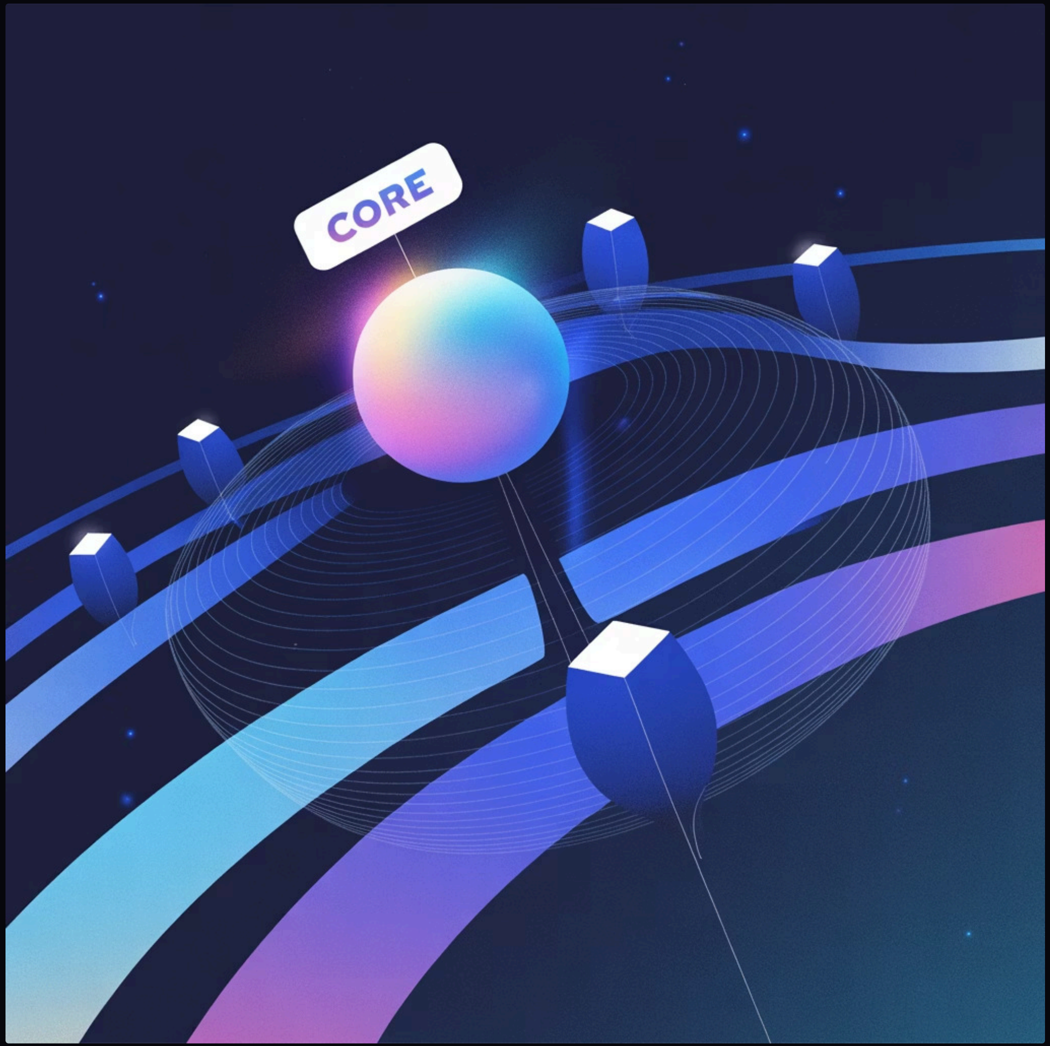
Chooses weather API tool

API Call

Retrieves current weather data

Response

Delivers accurate, real-time information



OpenAI Function Calling Implementation

Tool Definition

JSON schema describing function name, parameters, and types

Parameter Extraction

Automatic parsing of natural language into structured inputs

Execution Options

Parallel or sequential tool invocation based on needs

Error Handling

Built-in retry mechanisms and graceful failure modes

```json
{
  "name": "get_weather",
  "description": "Gets current weather for a given location.",
  "parameters": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "The name of the city"
      }
    },
    "required": ["location"]
  }
}
```

How Tool Calling Works: Under the Hood

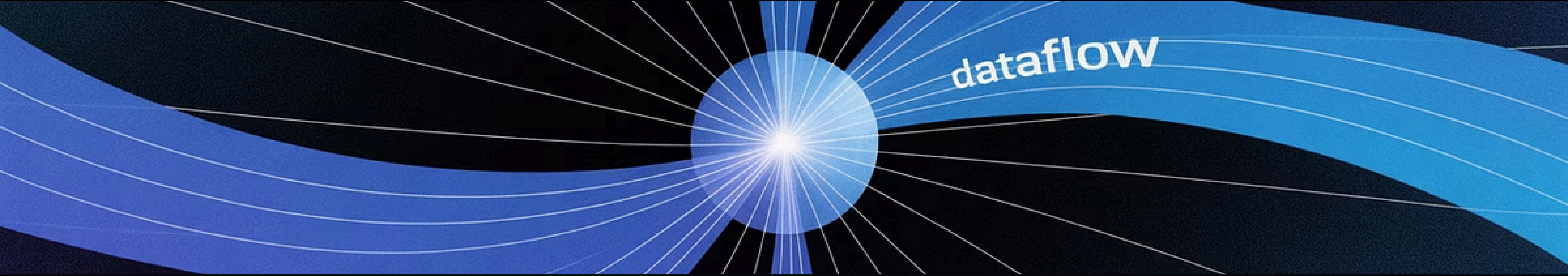
At its core, tool calling enables LLMs to bridge the gap between language understanding and external action. This is achieved through a specific internal mechanism: **https://platform.openai.com/docs/guides/function-calling?api-mode=responses**

Tool Definitions in the System Prompt

Unlike regular chat, an LLM capable of tool calling receives detailed descriptions (schemas) of available tools, their functions, and required parameters. These definitions are encoded directly into the system prompt, giving the LLM the context needed to understand its capabilities.

Structured JSON Response

When the LLM determines a tool is needed to fulfill a user's request, it doesn't generate a natural language reply. Instead, it responds with a structured JSON object that specifies the name of the tool to be called and the arguments to pass to it. An external orchestrator then intercepts this JSON, executes the tool, and feeds the result back to the LLM for further processing.



Introduction to LangGraph

Problems with Basic Tool Binding:

- No workflow control between tools
- Limited state management capabilities
- Inability to handle complex routing

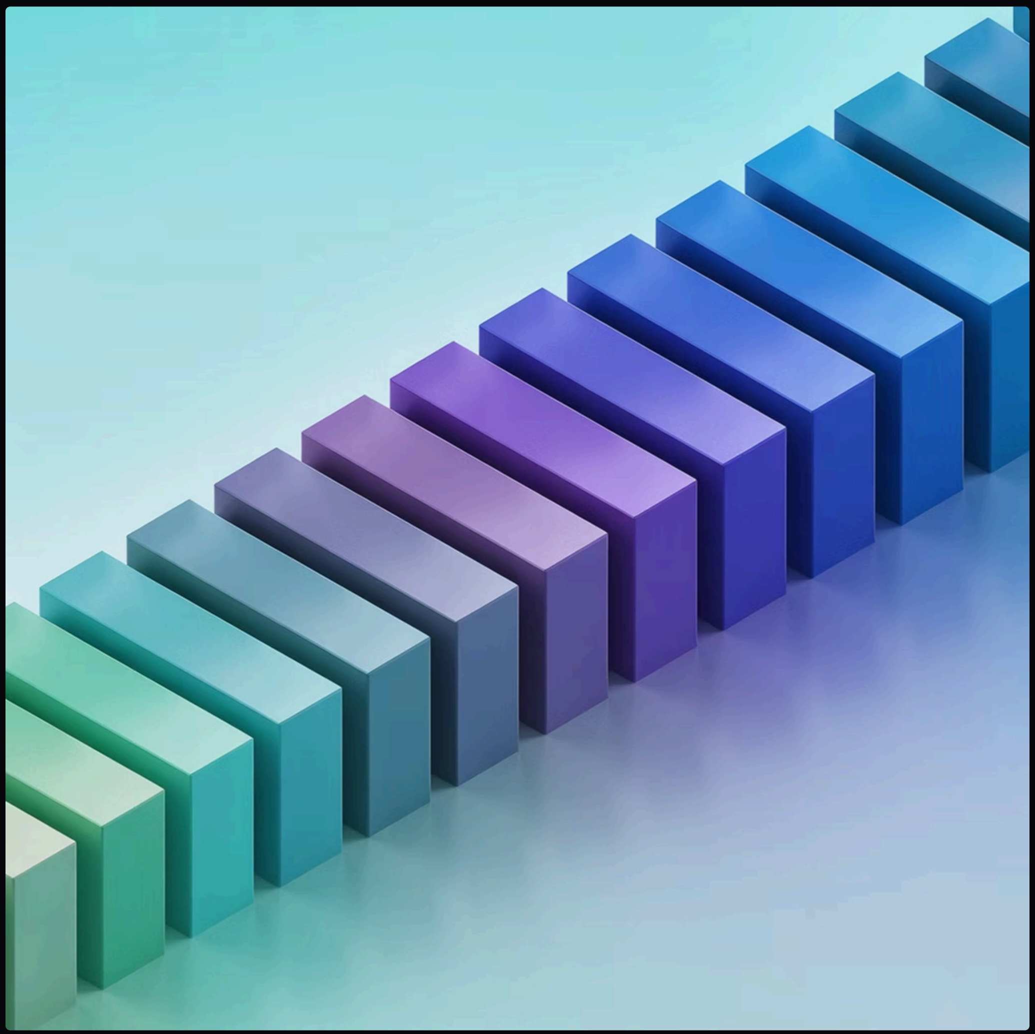
LangGraph Solutions:

- Explicit control flow between components
- Persistent state across interactions
- Conditional routing based on content
- Sophisticated memory management

Core Concepts: Nodes (processing units), Edges (control flow), State (shared data), Checkpoints (memory persistence)

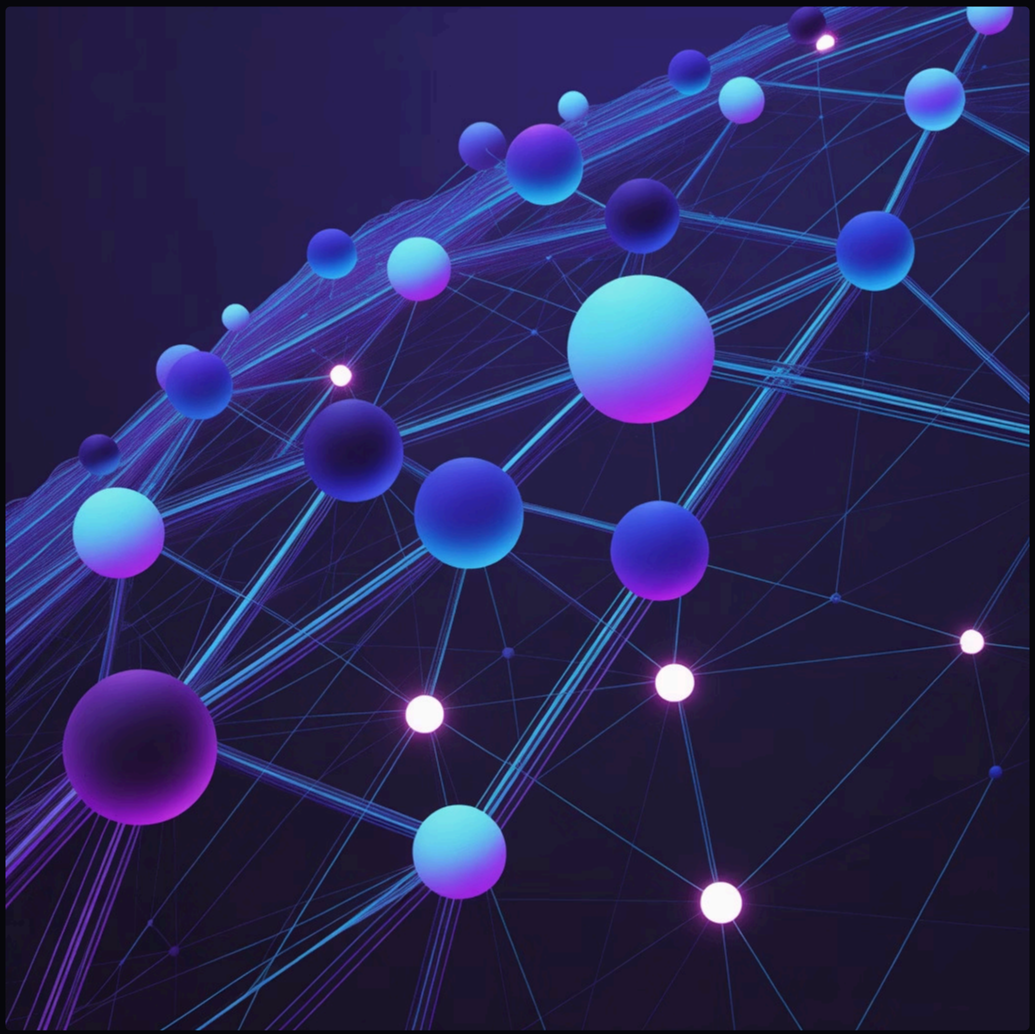
Graph Architecture vs Linear Flow

Linear Flow



- Simple implementation
- Limited to sequential execution
- No branching or complex workflows
- Minimal state management

Graph Architecture



- Multi-step reasoning capabilities
- Sophisticated state management
- Error recovery pathways
- Conditional branching logic
- Scalable design patterns

Nodes vs Tool Calling: Best Practices

Tool Calling (External APIs)

- Automatic parameter extraction
- Built-in error handling
- OpenAI optimized schemas
- Parallel execution capabilities

Example: Weather API, database queries, third-party services

Nodes (Complex Logic)

- Multi-step data processing
- Custom business logic implementation
- Complex conditional workflows
- Fine-grained control over execution

Example: Approval workflows, data transformation, routing logic

Choose the right pattern based on your integration complexity and control needs

The ReAct Pattern: Reasoning + Acting

Reasoning

Analyze the situation and determine the next action

Acting

Execute the appropriate tool or operation

Continuing

Repeat the cycle until task completion

Observing

Process results and update understanding

Benefits: Multi-step capability, dynamic decisions, error recovery, transparent reasoning

Memory and State Management

Thread-based Management

Separate conversation threads for each user session

Memory Types

- Short-term (conversation)
- Long-term (preferences)
- Working (processing)

langchain-ai.github.io

Overview

Build reliable, stateful AI systems, without giving up...

Checkpointing Options:

MemorySaver

In-memory storage for development

SQLite

Local persistence for testing

Redis

Distributed storage for production

What We'll Build: Fashion Assistant Agent

Real-time Weather Integration

Fetch current conditions to inform outfit recommendations

Wardrobe Database Querying

Search personal clothing inventory by type, color, and season

Multi-step Outfit Planning

Combine weather data with wardrobe options for contextual suggestions

Persistent Conversation Memory

Remember style preferences and previous recommendations

Technical Stack: LangChain, LangGraph, Azure OpenAI, Python

Workshop Learning Path & Key Takeaways

1. **Tool Creation**

   Weather API and wardrobe database functions

2. **Basic Tool Binding**

   Connect tools to LLM for simple interactions

3. **Graph Implementation**

   Build ReAct pattern with multi-step reasoning

4. **Memory Integration**

   Add persistent state across conversations

Key Takeaways

- Agents solve LLM limitations through tool integration and state management
- LangGraph enables production-ready agent systems with sophisticated workflows
- Best practices: Tool calling for APIs, nodes for complex logic
- Production needs: Authentication, monitoring, error handling, scalability
- Advanced horizons: Multi-agent systems, human-in-the-loop, structured outputs