

1. LangGraph vs CrewAI

Question:

How would you describe the key difference between LangGraph and CrewAI in terms of how they model and structure agent workflows? What mental model do you use to distinguish the two?

Answer:

LangGraph is **graph-based workflow orchestration**, while CrewAI is **role-based multi-agent orchestration**.

In LangGraph, I explicitly define nodes, edges, state, branching, loops, retries, and human-in-the-loop steps. It's suitable when I need precise control over a production workflow.

CrewAI focuses more on multiple specialized agents with different roles and goals collaborating on a task.

My mental model is: **LangGraph is a workflow or state machine, while CrewAI is a team of specialized agents.**

2. AutoGen vs LangGraph

Question:

When would you prefer AutoGen's conversation-based multi-agent pattern over LangGraph's state-machine approach?

Answer:

I would choose AutoGen when the problem is **open-ended and benefits from dynamic agent-to-agent conversation**, such as brainstorming, research, planning, or code generation with multiple agents reviewing each other's work.

I would choose LangGraph when the workflow is **more deterministic and requires explicit state management, branching, checkpoints, retries, and human approval**.

The key characteristic is **predictability**. If I know the workflow and need strong execution control, I prefer LangGraph. If the solution emerges dynamically through agent collaboration, AutoGen can be a better fit.

3. What is a Tool?

Question:

In the context of LangChain agents, what best describes the role of a tool?

Answer:

A tool is a **callable function or API wrapper that an agent can invoke to interact with external systems, retrieve information, or perform actions beyond the LLM's native capabilities**.

Examples include APIs, databases, search engines, calculators, and Python functions.

4. Persistent State in Multi-Agent Workflows

Question:

How would you design persistent state management for several agents handing off context across asynchronous steps?

Answer:

I would use **durable persistent state rather than relying on in-memory state**.

Each workflow would have a unique workflow ID. Agent outputs and workflow metadata would be stored in a persistent database such as PostgreSQL, while large artifacts could be stored in object storage.

I would use an event-driven queue such as Kafka, RabbitMQ, or SQS to pass tasks between agents. After each agent completes its work, I would checkpoint the state.

If an agent fails, the workflow can resume from the last checkpoint instead of starting from the beginning. I would also make agent operations **idempotent** so retries don't create duplicate results.

This provides durability, fault tolerance, resumability, and scalability.

5. Reflection Loop

Question:

How would you implement a reflection loop where an agent generates a response, evaluates it, and revises it if necessary? How would you prevent infinite looping?

Answer:

I would use a **generator-reviewer pattern**. The generator creates the initial response, and a reviewer evaluates it for accuracy, completeness, relevance, safety, and formatting.

If problems are found, the reviewer produces structured feedback. The generator uses that feedback to revise the response, and the process repeats.

To prevent infinite loops, I would define clear termination conditions:

1. Maximum number of iterations, such as 3.
2. A minimum quality threshold.
3. No significant improvement between iterations.
4. No new actionable feedback from the reviewer.

This gives us a bounded reflection loop with predictable latency.

6. End-to-End RAG Pipeline

Question:

Can you walk through the complete RAG pipeline from raw document upload to a grounded LLM response?

Answer:

I divide the RAG pipeline into **indexing and retrieval** phases.

During indexing, the document is uploaded and parsed. I clean the extracted content and split it into meaningful chunks with appropriate overlap. I then generate embeddings for each chunk and store them in a vector database along with metadata such as document ID, page number, and source.

During retrieval, the user's query is processed and converted into an embedding. I perform vector or hybrid retrieval using vector similarity and keyword search such as BM25.

I retrieve the top-K candidates and optionally apply a **cross-encoder reranker** to improve relevance. The best chunks are then placed into the LLM prompt along with the user's question. The LLM generates a grounded response based on the retrieved context, and I return the answer with source citations.

If retrieval confidence is too low, I avoid generating an unsupported answer and return an appropriate fallback.

7. Reranking and HyDE

Question:

Have you encountered cases where retrieved chunks were semantically similar to the query but still produced poor LLM responses? What advanced technology would you use?

Answer:

Yes. Semantic similarity doesn't always mean that the retrieved chunk actually contains the information needed to answer the question.

I would use a **reranker**, typically a cross-encoder, after initial vector retrieval. The reranker evaluates the query and candidate document together and produces a more accurate relevance score.

For poorly worded or ambiguous queries, I could also use **HyDE — Hypothetical Document Embeddings**. It generates a hypothetical answer, embeds that answer, and uses the embedding for retrieval.

I can combine this with hybrid search, query rewriting, metadata filtering, and context compression depending on the use case.

The mental model is:

- **Hybrid search → better candidate recall**
- **Reranking → better candidate precision**
- **HyDE → better query representation**

8. OpenAI API vs Azure OpenAI

Question:

When switching an application from the standard OpenAI API to Azure OpenAI, what differences did you encounter in authentication, endpoint configuration, and rate limiting?

Answer:

The main differences are authentication, endpoint configuration, and quota management.

With OpenAI, I typically use an API key and specify the model. With Azure OpenAI, I can use an API key or Microsoft Entra ID authentication.

Azure uses a **resource endpoint and deployment name**, rather than simply specifying the model name. The API version is also part of the Azure configuration.

For rate limiting, Azure quotas are managed around the Azure resource and deployments, including metrics such as TPM and RPM. I handle HTTP 429 responses using exponential backoff and retry strategies.

Azure also provides enterprise capabilities such as RBAC, private networking, regional deployment, and integration with the Azure ecosystem.

9. Chain-of-Thought Problem

Question:

Can you describe a situation where Chain-of-Thought prompting made things worse? What went wrong and how did you fix it?

Answer:

In a RAG application, I initially used Chain-of-Thought prompting for every query. It increased the response length, latency, and token consumption, and in some cases the model started reasoning beyond the retrieved context.

I diagnosed the problem by comparing **answer quality, latency, token usage, and hallucination rate** across different prompt strategies.

I found that simple factual retrieval questions didn't benefit from explicit reasoning instructions. I changed the approach to use **task-specific prompts**. For factual RAG queries, I instructed the model to answer only from the supplied context and explicitly state when the information wasn't available.

I reserved complex reasoning approaches for tasks that actually required multi-step reasoning. The result was lower latency, lower cost, and better grounding.

10. OpenAI Function-Calling Schema

Question:

Write a Python dictionary representing an OpenAI-compatible function-calling schema for a tool that searches a product catalog. The schema should support filtering by category, price range, and availability.

Answer:

```
search_product_tool = {  
    "type": "function",  
    "function": {  
        "name": "search_product_catalog",
```

<https://www.instagram.com/rawhul213>

```
"description": "Search a product catalog by category, price range, and availability.",
"parameters": {
  "type": "object",
  "properties": {
    "category": {
      "type": "string",
      "description": "Product category"
    },
    "min_price": {
      "type": "number",
      "description": "Minimum price"
    },
    "max_price": {
      "type": "number",
      "description": "Maximum price"
    },
    "available": {
      "type": "boolean",
      "description": "Whether the product is in stock"
    }
  },
  "required": ["category"],
  "additionalProperties": False
}
```

Important: This is a **Python dictionary**, so don't write:

```
def search_product_tool
```

That would cause the `SyntaxError: expected '('` shown in your screenshot.

11. FastAPI Long-Running Agent Task

Question:

Suppose a FastAPI endpoint triggers an AI agent task that takes 30–120 seconds. How would you design it so the HTTP connection isn't blocked?

Answer:

I would decouple the HTTP request from the long-running AI task.

The client sends a request, and FastAPI immediately creates a unique **job ID**, stores the job as PENDING, and puts the task into a background processing system such as Celery with Redis/RabbitMQ or another message queue.

The API immediately returns **HTTP 202 Accepted** with the job ID.

A background worker processes the AI task and updates the job status to RUNNING, COMPLETED, or FAILED.

The client can then either poll `/jobs/{job_id}` or use **SSE/WebSockets** to receive the result in real time.

This avoids HTTP timeouts and allows the workers to scale independently from the API layer.

12. SSE Streaming with FastAPI

Question:

How would you implement Server-Sent Events to stream output from a FastAPI backend to a browser in real time? What recovery strategy would you use if the system is interrupted?

Answer:

I would use **SSE with FastAPI's streaming response mechanism**. The backend generates output incrementally and sends each chunk as an SSE event. The browser consumes it using the JavaScript EventSource API.

Each event should have a unique **event ID or sequence number**.

For recovery, the client keeps track of the last received event. If the connection drops, SSE can reconnect and send the Last-Event-ID header. The server can then resume from the next event instead of starting from the beginning.

I would also persist the job state and generated chunks in Redis or a database. This allows the stream to recover even after a worker or server restart.

So the important concepts are **SSE, event IDs, Last-Event-ID, checkpointing, persistence, and resumability**.

Simple implementation

```
from fastapi import FastAPI
from sse_starlette.sse import EventSourceResponse
import asyncio

app = FastAPI()

@app.get("/stream")
```

<https://www.instagram.com/rawhul213>

```
async def stream():
    async def event_generator():
        for i in range(5):
            yield {
                "id": str(i),
                "event": "message",
                "data": f"Token {i}"
            }
        await asyncio.sleep(1)
    return EventSourceResponse(event_generator())
```

13. AI Safety Guardrail Failure

Question:

How do you design safe fallback behavior when an agent guardrail is triggered midway through a multi-step workflow, without exposing partial or unsafe state?

Answer:

I would design the workflow using a **fail-safe guardrail architecture**.

Each critical agent step passes through a safety validation before its output is committed or exposed.

If a safety check is triggered, I immediately stop downstream execution and mark the workflow as BLOCKED or FAILED_SAFE.

Any unsafe intermediate output is quarantined and is never returned to the user. I persist the workflow state and log the safety event for auditing.

The user receives a safe fallback response, such as:

"I can't complete this request because it violates safety policies."

For transient technical failures, the workflow can resume from the last checkpoint. But for a genuine safety violation, the workflow must **terminate and cannot resume past the unsafe step**.

This ensures **no unsafe or partial state reaches the user while maintaining auditability and fault tolerance**.

Most important keywords to remember for this interview

RAG:

Chunking → Embeddings → Vector DB → Hybrid Search → Top-K → Reranking → Context → LLM → Grounded Answer

Agents:

State → Nodes → Tools → Checkpoints → Routing → Reflection → Termination

Production:

Async → Queue → Worker → Job ID → Persistence → Retry → Idempotency

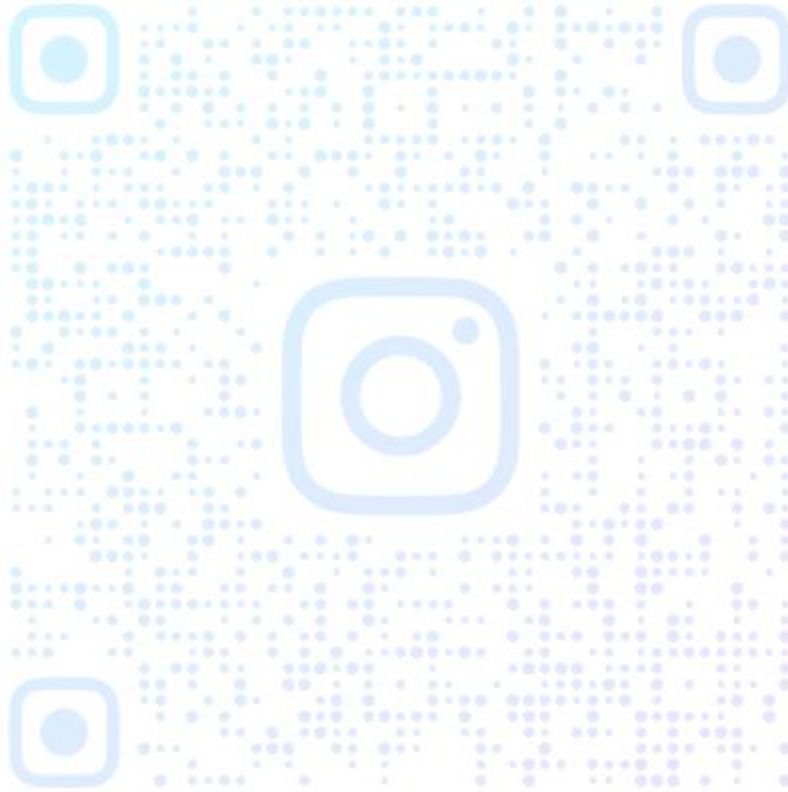
<https://www.instagram.com/rawhul213>

Streaming:

SSE → Event ID → Last-Event-ID → Reconnect → Checkpoint → Resume

Safety:

Guardrail → Validate → Block → Quarantine → Audit → Safe Fallback



@RAWHUL213