🤖 Technical Challenge – AI Engineer Junior

Artefact Assistant | Data & AI to Drive Impact

An intelligent AI assistant that automatically decides when to use external tools (calculator, FX, crypto) or respond using its own knowledge.
The assistant supports **tool routing**, **multi-tool chaining**, **optional conversational memory**, and an **interactive CLI**, implemented using **LangGraph** and **LangChain**.

📂 Project Structure

Backend

```
backend/
├── src/
│   ├── llm/
│   │   ├── models/
│   │   │   └── models.py        # Typed agent state definition
│   │   ├── assistant.py         # Assistant logic (with and without memory)
│   │   ├── graph.py             # LangGraph execution graph
│   │   └── llm_call.py          # LLM invocation and configuration
│   ├── observability/
│   │   └── context.py           # Tracing and tool-usage context
│   └── tools/
│       └── tools.py             # External tools (calculator, FX, crypto)
├── .env                         # Environment variables (API keys, config)
├── main.py                      # Backend entry point (FastAPI / CLI)
└── requirements.txt             # Python dependencies
```

Frontend

```
frontend/
└── src/
    ├── app/
    │   ├── page.tsx            # Main application entry point (state + layout
orchestration)
    │   ├── page.module.css     # Global page layout and container styles
    │   ├── chat.module.css     # Chat area, message list, scrolling, empty state
styles
    │   ├── controls.module.css # Input field, buttons, and interaction controls
styles
    │   └── globals.css         # Global CSS reset, fonts, and base styles
    │
    ├── components/
    │   ├── Header.tsx          # Header with branding (logo + title)
    │   ├── ChatBox.tsx         # Scrollable chat container with auto-scroll logic
    │   ├── MessageBubble.tsx   # Single message renderer (user vs assistant)
    │   └── TypingIndicator.tsx # Animated typing indicator while LLM responds
    │
    └── lib/
        ├── api.ts              # Backend API client (chat, reset, etc.)
        └── session.ts          # Session ID management for conversational memory
```

🚀 How to Run

1. Installation

```
git clone <your-repository>
cd source/backend/src
pip install -r requirements.txt
```

2. Configuration

Create a `.env` file with your OpenAI API key:

```
OPENAI_API_KEY=sk-proj-xxxxxxxxxxxxxx
```

How to obtain an OpenAI API key:

- Go to https://platform.openai.com/api-keys
- Create a new API key
- Copy it into the `.env` file

---

3. Run the Backend

```
uvicorn main:app --reload
```

4. Run the Frontend

```
# Navigate to the Frontend Directory
cd source/frontend
```

```
# Install next.js dependencies
npm start
```

```
# Activate frontend application
npm run dev
```

Now, you can open the application in `https://localhost:3000/`

🧠 Architecture

LangGraph Execution Flow

```
      ┌─────────────┐
      │   User      │
      │  Question   │
      └──────┬──────┘
             │
             ▼
      ┌──────────────┐
      │    Agent     │◄──────────┐
      │  (LLM + AI)  │           │
      └──────┬───────┘           │
             │                   │
       ┌─────▼─────┐             │
       │ Needs a   │             │
       │  tool?    │             │
       └─────┬─────┘             │
             │                   │
       ┌─────┴─────┐             │
      Yes          No            │
       │            │            │
       ▼            ▼            │
  ┌──────────┐ ┌────────┐        │
  │  Tools   │ │  END   │        │
  │  Node    │ └────────┘        │
  └────┬─────┘                   │
       │                         │
       └─────────────────────────┘
```

🧩 Core Components

`./tools/`

Defines all **external tools** available to the assistant. Each tool is deterministic, isolated, and observable.

- `calculator()` Performs deterministic arithmetic operations (used for exact calculations and post-processing).
- `fx_convert()` Converts fiat currencies using a public FX API, returning precise exchange rates.

- `crypto_convert()` Retrieves cryptocurrency prices and conversions using the CoinGecko public API (no API key required).

Each tool explicitly reports when it is used, enabling clear observability of routing decisions.

`./graph/`

Defines the **LangGraph execution graph** and the agent control flow.

- `AgentState` Typed state object that holds the full message history across nodes.
- `call_model()` Invokes the LLM with tool binding enabled, allowing the model to decide whether a tool is required.
- `should_continue()` Conditional routing logic that checks for `tool_calls` and decides whether to:
    - route execution to the Tools node, or
    - terminate the graph.
- `create_graph()` Assembles and compiles the LangGraph workflow, connecting agent and tool nodes into a deterministic loop.

`./assistant/`

High-level assistant interface layer.

- `AssistantWithMemory` Wraps the execution graph with conversational memory, enabling multi-turn interactions while keeping the graph itself stateless.
- `run_assistant()` Stateless, single-shot compatibility function that runs the assistant without memory (useful for scripts, tests, or demos).

`./observability/` (or equivalent telemetry utilities)

Provides **lightweight observability** for runtime execution.

- Tracks **which tools were used** during a request
- Attaches a **trace_id** to each interaction
- Preserves **context propagation** across turns when memory is enabled

This makes agent behavior transparent, debuggable, and aligned with production-grade AI system expectations.

💡 How It Works

1. **User submits a question** -> converted to a `HumanMessage`
2. **Agent node (LLM)** analyzes the context

3. **Routing decision**:
   - Math-related -> calls `calculator`
   - Currency conversion -> calls `fx_convert`
   - Crypto price/conversion -> calls `crypto_convert`
   - General knowledge -> answers directly

4. **Tools node** executes the requested tool

5. **Agent loops back** to generate the final response

6. **Conversation memory** (optional) preserves context across turns

---

📊 Examples

```
>>> run_assistant("What does Artefact do in a few words?")
"Artefact is a data and digital consulting company that helps organizations
accelerate AI and data adoption to drive business impact."

>>> run_assistant("How much is 0.1 BTC in BRL?")
"[TOOL] Using CRYPTO Converter (CoinGecko)"
"🤖 Artefact Assistant: 0.1 BTC is approximately 49,031.40 BRL."

>>> run_assistant("How much is 1 USD in BRL?")
"[TOOL] Using FX CONVERTER"
"🤖 Artefact Assistant: 1 USD is approximately 5.39 BRL."

>>> run_assistant("Using the current BTC price in BRL and the USD/BRL exchange
rate, what is the price of 0.1 BTC in USD?")
"[TOOL] Using CRYPTO CONVERTER (CoinGecko)"
"[TOOL] Using FX CONVERTER"
"[TOOL] Using local CALCULATOR"
"🤖 Artefact Assistant: The price of 0.1 BTC is approximately 9,103.65 USD."
```

These examples demonstrate how the assistant **combines multiple tools across turns** while preserving conversational context. The assistant retrieves external data, performs deterministic calculations, and chains results logically to solve multi-step user requests in a transparent and explainable way.

🧠 Implementation Logic (Summary)

- Built a **minimal, deterministic agent loop** using LangGraph
- Used an **Agent (LLM) node** and a **Tools node**
- Enabled **tool calling** via `bind_tools`
- Routed execution based on the presence of `tool_calls`
- Continued looping until no tool was required
- Used **deterministic tools** for calculations and conversions to ensure correctness and trust
- Added **observability logs** to clearly show when tools are used
- Implemented **memory as a thin wrapper**, without modifying the core graph

---

✨ User Experience & Product Considerations

Beyond the core technical requirements, several UX-oriented decisions were intentionally added to make the assistant feel closer to a real product:

- Typing indicator while the LLM is processing, reducing perceived latency
- Clear visibility of when external tools are used
- Deterministic tool outputs for reliability and explainability
- Optional conversational memory for natural multi-turn interactions

These small details improve usability while keeping the system simple and maintainable.

🚀 Conversational Memory (Optional Enhancement)

Although **conversational memory was not a required feature** for this challenge, it was included as an optional enhancement to demonstrate how the assistant could evolve in a real-world scenario.

To keep the core solution aligned with the challenge scope:

- The **LangGraph execution graph remains stateless**
- Memory is implemented as a **thin wrapper (`AssistantWithMemory`)**
- The same graph can be reused in **single-shot** or **conversational** modes

This preserves clarity and extensibility without increasing system complexity.

🧠 Learnings & Next Steps

What I learned This project deepened my understanding of how to orchestrate LLMs with tools using execution graphs, and how small decisions around state and routing significantly impact clarity, reliability, and scalability.

What I'd do with more time I would further improve routing robustness, expand testing and observability, and refine long-term memory handling for extended conversations.

🔧 Technologies

- **LangGraph** – agent orchestration and control flow
- **LangChain** – tools and LLM integration
- **OpenAI GPT-4o-mini** – language model
- **FastAPI** – backend API for chat sessions and tool execution
- **Next.js (React)** – interactive frontend chat interface
- **Python 3.9+**

📚 References

- https://langchain-ai.github.io/langgraph/

- https://python.langchain.com/docs/modules/tools/
- https://platform.openai.com/docs/guides/function-calling

Author: Guilherme Indiciate **Created on**: January 7th, 2026 **Role applied for**: AI Engineer – Full-Stack for Generative AI Applications **Context**: Technical Challenge | AI Engineer Junior Position