Python

Hands-On

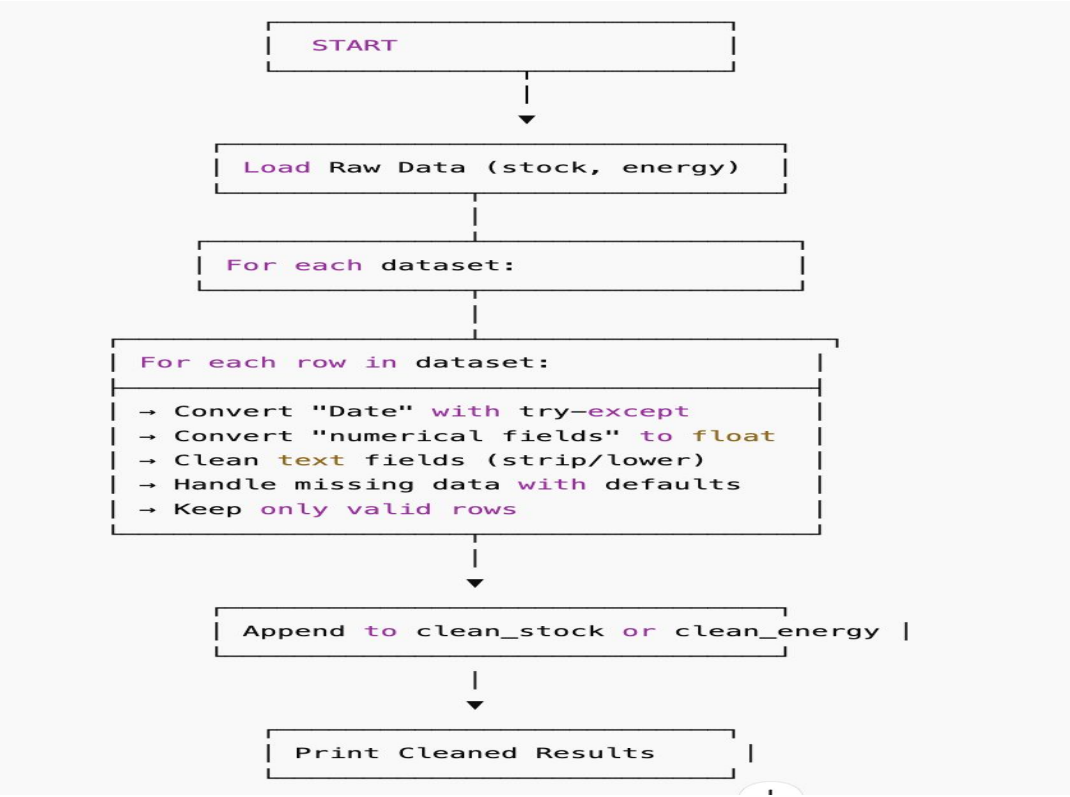
Case Study-1

🧩 1. Design Thinking Diagram

```
[ Empathize ]   → What problem?
   └── Raw data has missing fields, invalid dates, inconsistent formats.

[ Define ]      → Goal?
   └── Build clean structured data for stock and energy analytics.

[ Ideate ]      → Approach?
   └── For each dataset:
       - Validate date
       - Convert numbers safely
       - Normalize categories
       - Filter valid records

[ Prototype ]   → Code design
   └── Use loops, conditionals, type conversions, and try-except blocks.

[ Test ]        → Verify output
   └── Print cleaned records and visually inspect.
```

Case Study-1

Flow of Code Diagram



Case Study-1

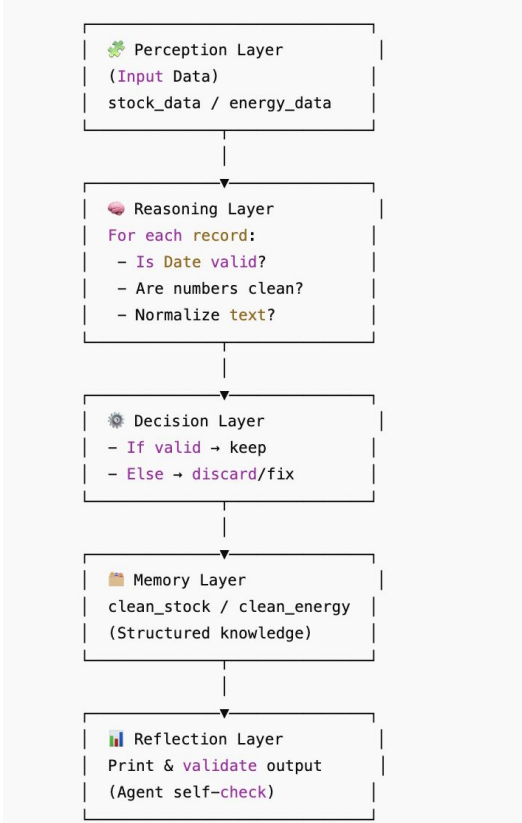
Flow of Code Diagram

🧠 3. Concept Map Diagram – Python Concepts Applied

Concept	Where Used	Purpose / Why Important for AI Agents
Lists of Dicts	Input datasets (`stock_data`, `energy_data`)	Store multiple records — like agent memories.
For Loops	Iterate through rows	Sequentially process each record — key for stream processing.
Try / Except	Date parsing	Handle noisy data gracefully (error tolerance).
Type Conversion	`float()`, `int()`	Ensure correct numerical computation.
Conditionals (if)	Filter valid rows	Maintain data integrity.
String Methods	`.strip()`, `.lower()`	Normalize text for ML-ready features.
Data Validation	Check missing or invalid data	Prevent agent logic errors downstream.
Print/Debug	Final section	Visual validation of cleaned state.

Case Study-1

Agent Think


🧩 Perception Layer
(Input Data)
stock_data / energy_data
🧠 Reasoning Layer
For each record:
- Is Date valid?
- Are numbers clean?
- Normalize text?
⚙️ Decision Layer
- If valid → keep
- Else → discard/fix
🗂️ Memory Layer
clean_stock / clean_energy
(Structured knowledge)
📊 Reflection Layer
Print & validate output
(Agent self-check)

Case Study-2

1️⃣ Design Thinking Diagram 🔁

Purpose: Show the **thought process** behind designing `TradeOrder`.

```scss
[Identify Problem]
       |
       ▼
Need to model trade orders for a trading system
       |
       ▼
[Define Requirements]
- Order type (buy/sell)
- Amount and price
- Status (pending, executed, cancelled)
       |
       ▼
[Ideate Solutions]
- Class-based design
- Methods for execution and cancellation
- Status tracking
       |
       ▼
[Prototype]
- TradeOrder class with execute(), cancel(), __str__()
       |
       ▼
[Test & Refine]
- Example usage: create order, execute, cancel
```

Case Study-2

2️⃣Flow of Code Diagram 🔁

```
[Start]
   |
   ▼
Create TradeOrder instance (order1)
   |
   ▼
Print order1 (__str__)
   |
   ▼
Call order1.execute()
   |
   ├─ Check status == "pending"?
   |      ├─ Yes → Execute order, set status="executed"
   |      └─ No  → Print cannot execute
   |
   ▼
Return total value
   |
   ▼
Print order1 (__str__ after execution)
   |
   ▼
Call order1.cancel()
   |
   ├─ Check status == "pending"?
   |      ├─ Yes → Cancel order, set status="cancelled"
   |      └─ No  → Print cannot cancel
   |
   ▼
[End]
```

Case Study-2

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

Purpose: Show **Python concepts used in the code.**

scss

```
[TradeOrder Class]
   |
   ├─ __init__() → Constructor, instance variables
   |
   ├─ execute() → Method, conditional logic, arithmetic (*)
   |
   ├─ cancel() → Method, conditional logic
   |
   ├─ __str__() → Special method, string representation
   |
   ├─ instance variables → order_type, amount, price, status
   |
   └─ Example usage → object creation, method calls, print
```

Case Study-2

4️⃣ Cognitive Journey Diagram 🧩

Purpose: Show **how a user or developer thinks while interacting with the code.**

```sql
[Developer/User Creates Order]
   │
   ▼
Decides type, amount, price
   │
   ▼
Print initial order → sees status = pending
   │
   ▼
Execute order
   │
   ├─ Checks status → pending → execute
   │
   └─ Calculates total value → prints execution details
   │
   ▼
Try to cancel order
   │
   ├─ Status not pending → prints cannot cancel
   │
   ▼
[Final Status: executed]
```

Case Study-3

1️⃣ Design Thinking Diagram 🔁

Purpose: Show the thought process behind combining **TradeOrder class** with a **l**

```pgsql
[Identify Problem]
       |
       ▼
Need to track execution time, success/failure of trade orders
       |
       ▼
[Define Requirements]
- Log start/end of transaction
- Handle exceptions
- Track execution duration
       |
       ▼
[Ideate Solutions]
- Use a decorator to wrap execute() method
- Keep TradeOrder class focused on business logic
       |
       ▼
[Prototype]
- Create transaction_logger decorator
- Apply decorator to execute() method
- Test with normal and failed execution
       |
       ▼
[Test & Refine]
- Check logs for success/failure and duration
- Ensure original functionality works
```

Case Study-3

2️⃣ Flow of Code Diagram 🔁

Purpose: Show **step-by-step execution** with decorator, including logging and error handling.

pgsql Copy code

```
[Start]
   |
   ▼
Create TradeOrder instance (order1)
   |
   ▼
Call order1.execute()
   |
   ├─ Enter transaction_logger.wrapper
   |      ├─ Print "[LOG] Starting transaction..."
   |      ├─ Record start_time
   |      └─ Call actual execute() method
   |               ├─ Check status == "pending"?
   |               |       ├─ Yes → execute order, update status
   |               |       └─ No → raise Exception
   |               └─ Return total_value
   |
   ├─ Record end_time
   ├─ Print "[LOG] Transaction executed successfully..." or "[ERROR] Transaction failed..
   └─ Return result
   |
   ▼
[End]
```

Case Study-3

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

pgsql

```
[TradeOrder Class]
   |
   ├─ __init__() → Initialize order_type, amount, price, status
   |
   ├─ execute() → Business logic for executing order
   |        └─ Decorated with transaction_logger
   |
[transaction_logger Decorator]
   |
   ├─ functools.wraps → preserves function metadata
   ├─ wrapper(*args, **kwargs) → wraps original method
   ├─ Logs start time, end time
   ├─ Exception handling → logs errors
   └─ Returns original result
   |
[Python Concepts]
   ├─ Classes & Objects
   ├─ Methods
   ├─ Decorators
   ├─ Error Handling (try/except)
   ├─ Time module (execution duration)
   ├─ OOP + Separation of Concerns
```

Case Study-3

Case Study-3

4️⃣ Cognitive Journey Diagram 🧩

Purpose: Show **user/developer mental model** when interacting with the code.

```
pgsql

[Developer Creates Order]
   |
   ▼
Print initial order (status="pending")
   |
   ▼
Execute order
   |
   ├─ Decorator logs start
   ├─ execute() checks status → pending → execute
   ├─ Update status = executed
   ├─ Print execution details
   └─ Decorator logs success with duration
   |
   ▼
Execute already executed order (order2)
   |
   ├─ Decorator logs start
   ├─ execute() checks status → not pending → raise Exception
   └─ Decorator logs error with duration
   |
   ▼
[End: developer sees logs + order status]
```

Case Study-4

1️⃣ Design Thinking Diagram 🔁

Purpose: Show the thought process behind designing an **async market data fetcher**.

```pgsql
[Identify Problem]
       |
       ▼
Need to fetch stock & crypto prices concurrently
       |
       ▼
[Define Requirements]
- Fetch multiple stock & crypto symbols
- Use async to reduce wait time
- Return structured price data
- Support demo & real API keys
       |
       ▼
[Ideate Solutions]
- Use aiohttp for async HTTP requests
- Wrap fetching in async functions
- Gather tasks concurrently
- Separate stock and crypto fetchers
       |
       ▼
[Prototype]
- fetch_stock_price() & fetch_crypto_price()
- fetch_json() for general async fetch
- main() orchestrates tasks with asyncio.gather
       |
       ▼
[Test & Refine]
- Demo mode with mock prices
- Print results in readable format
```

Case Study-4

2️⃣ Flow of Code Diagram 🔁

Purpose: Show **step-by-step execution** including async task orchestration.

```
scss

[Start]
   |
   ▼
Call asyncio.run(main())
   |
   ▼
Create stock list ["AAPL", "GOOGL"] and crypto list ["BTC", "ETH"]
   |
   ▼
Open aiohttp ClientSession()
   |
   ▼
Create async tasks:
   ├─ stock_tasks = fetch_stock_price(session, symbol)
   └─ crypto_tasks = fetch_crypto_price(session, symbol)
   |
   ▼
Execute tasks concurrently → await asyncio.gather(*tasks)
   |
   ├─ For each fetch_stock_price:
   |       ├─ Demo mode? → sleep + generate random price
   |       └─ Real API → fetch_json(session, url)
   |
   ├─ For each fetch_crypto_price:
   |       ├─ Demo mode? → sleep + generate random price
   |       └─ Real API → fetch_json(session, url, headers)
   |
   ▼
Gather results → list of dicts
   |
   ▼
Print formatted market prices
   |
   ▼
[End]
```

Case Study-4

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

```
[Async Market Fetcher]
   |
   ├─ Async Functions:
   |      ├─ fetch_json(session, url)
   |      ├─ fetch_stock_price(session, symbol)
   |      └─ fetch_crypto_price(session, symbol)
   |
   ├─ Main Async Orchestration:
   |      └─ main() → asyncio.gather for concurrent tasks
   |
   ├─ Libraries:
   |      ├─ asyncio → async/await
   |      ├─ aiohttp → HTTP client session
   |      ├─ random → simulate data
   |      └─ os → environment variables
   |
   ├─ Control Flow:
   |      ├─ Demo mode check (mock data)
   |      └─ API mode (real fetch)
   |
   ├─ Data Structures:
   |      ├─ Lists → symbols
   |      ├─ Dicts → result {symbol: price}
   |      └─ List of dicts → gathered results
   |
   └─ Output:
          └─ Print results
```

Case Study-4

Case Study-4

4 Cognitive Journey Diagram 🧩

Purpose: Show **how a developer or user thinks** when using the async fetcher.

```sql
[Developer Runs Program]
   |
   ▼
Decides symbols to fetch (stocks + cryptos)
   |
   ▼
Program checks API keys → selects demo or real mode
   |
   ▼
Async tasks created for each symbol
   |
   ▼
Concurrent execution:
   ├─ Fetch stock prices
   └─ Fetch crypto prices
   |
   ▼
Mock/demo → random values + simulated delay
Real API → HTTP request & parse JSON
   |
   ▼
Gather results from all tasks
   |
   ▼
Print formatted prices → user sees all results
   |
   ▼
[End: Efficient & concurrent data fetch]
```

Case Study-5

1 Design Thinking Diagram 🔁

Purpose: Show the **design process behind building a Q&A assistant.**

```
css

[Identify Problem]
        |
        ▼
Need to provide concise answers to banking regulation questions
        |
        ▼
[Define Requirements]
- Accept questions dynamically
- Format prompts consistently
- Use a mock or real LLM to generate answers
- Support multiple questions sequentially
        |
        ▼
[Ideate Solutions]
- Use a PromptFormatter Runnable to format questions
- Use a MockLLM Runnable to simulate LLM responses
- Chain them using RunnableSequence
        |
        ▼
[Prototype]
- Define template
- Implement PromptFormatter and MockLLM
- Create RunnableSequence and test with sample questions
        |
        ▼
[Test & Refine]
- Verify answers are contextually correct
- Ensure chain works for multiple questions
```

Case Study-5

2️⃣ Flow of Code Diagram 🔁

Purpose: Show **step-by-step execution** of the chain.

```csharp
[Start]
   |
   ▼
Define PromptFormatter with template
   |
   ▼
Instantiate MockLLM
   |
   ▼
Create RunnableSequence(chain) with PromptFormatter & MockLLM
   |
   ▼
For each question in questions:
   ├─ Pass {"question": q} to chain.invoke()
   |      ├─ PromptFormatter.invoke() formats the prompt
   |      └─ MockLLM.invoke() generates answer based on prompt
   |
   ▼
Print formatted Q&A
   |
   ▼
[End]
```

Case Study-5

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

```
[Banking Q&A Assistant]
   |
   ├─ Classes:
   |      ├─ MockLLM(Runnable) → invoke(prompt) → returns answer
   |      └─ PromptFormatter(Runnable) → invoke(inputs) → formats template
   |
   ├─ RunnableSequence → sequences Runnables
   |
   ├─ Python Concepts:
   |      ├─ Classes & Inheritance (Runnable)
   |      ├─ Methods & Instance Variables
   |      ├─ String Formatting (.format())
   |      ├─ Loops (for question in questions)
   |      └─ Dictionaries ({"question": q})
   |
   └─ Output → Prints Q&A
```

Case Study-5

Case Study-5

4️⃣ Cognitive Journey Diagram 🧩

Purpose: Show **how a user/developer thinks about the code.**

```
less

[Developer Defines Template]
   |
   ▼
Instantiate PromptFormatter → formats input questions
   |
   ▼
Instantiate MockLLM → simulates LLM responses
   |
   ▼
Create RunnableSequence → chains formatting and LLM
   |
   ▼
For each question:
   ├─ PromptFormatter formats prompt
   ├─ MockLLM generates answer
   └─ Print Q&A
   |
   ▼
[End: Developer sees multiple formatted answers]
```

Case Study-6

1️⃣ Design Thinking Diagram 🔁

Purpose: Show the design process for building a **trading report summarizer**.

```vbnet
[Identify Problem]
       |
       ▼
Need to provide concise summaries of long trading reports
       |
       ▼
[Define Requirements]
- Accept report text
- Summarize first few key sentences
- Support async processing for integration
       |
       ▼
[Ideate Solutions]
- Use async function to handle reports
- Decorate with @kernel_function for metadata
- Slice sentences to generate summary
       |
       ▼
[Prototype]
- Define summarize_report(ctx) function
- Use ctx["report"] input
- Split into sentences → join first 2 sentences
       |
       ▼
[Test & Refine]
- Test with sample trading report
- Print concise summary
```

Case Study-6

2️⃣ Flow of Code Diagram 🔁

Purpose: Show **step-by-step execution flow** including async function call.

```pgsql
[Start]
   |
   ▼
Define @kernel_function decorated async summarize_report(ctx)
   |
   ▼
Define async main()
   |
   ▼
Set trading_report (multi-line string)
   |
   ▼
Call await summarize_report({"report": trading_report})
   |
   ├─ Access ctx["report"]
   ├─ Split report into sentences
   ├─ Take first 2 sentences
   └─ Join them → summary
   |
   ▼
Print summary
   |
   ▼
[End]
```

Case Study-6

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

csharp

```
[Trading Report Summarizer]
   |
   ├─ Async Function:
   |      ├─ async def summarize_report(ctx)
   |      └─ await summarize_report(...)
   |
   ├─ Decorators:
   |      └─ @kernel_function(description, name)
   |
   ├─ String Processing:
   |      ├─ Split sentences (.split("."))
   |      └─ Strip & join first 2 sentences
   |
   ├─ Data Structures:
   |      └─ Dictionary ctx {"report": ...}
   |
   ├─ Libraries:
   |      ├─ asyncio → async execution
   |      └─ semantic_kernel.functions → kernel_function decorator
   |
   └─ Output → Print concise summary
```

Case Study-6

Case Study-6

4️⃣ Cognitive Journey Diagram 🧩

Purpose: Show **how a developer or user interacts mentally** with the code.

```pgsql
[Developer Writes Function]
   |
   ▼
Decorate with @kernel_function → metadata for Semantic Kernel
   |
   ▼
Writes async summarize_report(ctx)
   |
   ▼
User/Developer provides trading_report string
   |
   ▼
Call await summarize_report({"report": trading_report})
   |
   ├─ Function splits sentences
   ├─ Extracts key sentences
   ├─ Joins them into summary
   |
   ▼
Print concise summary → user reads output
   |
   ▼
[End: Report summarized efficiently in async flow]
```

Case Study-7

1️⃣ Design Thinking Diagram 🔁

Purpose: Show the **design process for building a stock news summarization agent**.

```pgsql
[Identify Problem]
       |
       ▼
Need to summarize multiple stock news articles concisely
       |
       ▼
[Define Requirements]
- Accept multiple news articles
- Generate concise summary (2-3 sentences)
- Use modular design for easy extensibility
       |
       ▼
[Ideate Solutions]
- Fetch mock news (fetch_stock_news)
- Use PromptRunnable to format input text
- Use MockSummarizer to create natural-language summary
- Chain using RunnableSequence
       |
       ▼
[Prototype]
- Implement fetch_stock_news() → returns random news
- PromptRunnable formats news into template
- MockSummarizer extracts & summarizes sentences
       |
       ▼
[Test & Refine]
- Run demo
- Print original news + summarized output
```

Case Study-7

2️⃣ Flow of Code Diagram 🔁

Purpose: Show **step-by-step execution flow** of the agent.

```pgsql
[Start]
   |
   ▼
Call fetch_stock_news() → list of 3 random news articles
   |
   ▼
Join news articles into single string → news_text
   |
   ▼
Create PromptRunnable(template) → formats news_text
Create MockSummarizer() → summarizes formatted text
Create RunnableSequence(prompt_runnable, summarizer)
   |
   ▼
Invoke agent_chain.invoke({"news": news_text})
   |
   ├─ PromptRunnable.invoke() → inserts news_text into template
   └─ MockSummarizer.invoke() → splits sentences, joins 2–3 key sentences
   |
   ▼
Return summary string
   |
   ▼
Print original news + summarized output
   |
   ▼
[End]
```

Case Study-7

3️⃣ Concept Map Diagram – Python Concepts Applied 🧠

```
[Stock News Summarization Agent]
   |
   ├─ Data Fetching:
   |      └─ fetch_stock_news() → returns list of random articles
   |
   ├─ Classes / Runnables:
   |      ├─ PromptRunnable → formats template
   |      └─ MockSummarizer → extracts & summarizes sentences
   |
   ├─ RunnableSequence → chains runnables sequentially
   |
   ├─ Python Concepts:
   |      ├─ Functions & Methods
   |      ├─ String Methods → split(), join(), strip()
   |      ├─ Lists → articles, sentences
   |      ├─ Dictionaries → inputs to Runnable
   |      └─ Error Handling → type check in PromptRunnable
   |
   └─ Output → Prints original news + concise summary
```

Case Study-7

Case Study-7

4️⃣ Cognitive Journey Diagram 🧩

Purpose: Show **how a developer or user thinks** when interacting with the agent.

```pgsql
[Developer Runs Script]
   |
   ▼
Call fetch_stock_news() → get 3 random news articles
   |
   ▼
Combine articles into news_text string
   |
   ▼
Create PromptRunnable(template) → formats news into structured prompt
   |
   ▼
Create MockSummarizer → summarization logic
   |
   ▼
Create RunnableSequence → chains PromptRunnable + MockSummarizer
   |
   ▼
Invoke agent_chain.invoke({"news": news_text})
   |
   ├─ PromptRunnable formats input
   ├─ MockSummarizer extracts sentences, joins key points
   └─ Returns concise summary
   |
   ▼
Print original news + summary → user sees structured output
   |
   ▼
[End: Efficient modular summarization flow]
```

Prompt vs Chain

```
[Scenario: Wife as AI Assistant]

1️⃣ Prompt Example
------------------
[You] → "Please make dinner"
      |
      ▼
[Wife / AI] → Prepares dinner
      |
      ▼
[Output] → Dinner is ready
```

GenAI

Indepth

Prompt vs Chain

```
2️⃣ Chain Example
------------------
[You] → "Please do the following tasks in order:
          1. Make dinner
          2. Set the table
          3. Call kids to eat
          4. Clean the kitchen"
      |
      ▼
[Wife / AI] → Executes step-by-step
      |
      ▼
[Output]
  - Dinner cooked
  - Table set
  - Kids called
  - Kitchen cleaned
```

LangChain

```
[Human/User]
      |
      ▼
  "Honey, do these tasks:"
      |
      ▼
+---------------------+
|      Wife / LLM     |
| (Executes tasks)    |
+---------------------+
      |
      ▼
[Chain of Actions / Steps]
  ├─ Step 1: Plan dinner
  ├─ Step 2: Pay bills
  └─ Step 3: Remind to call mom
      |
      ▼
[Memory / Notes]
  - Last week's menu
  - Pending bills
  - Mom's phone number
      |
      ▼
[Output / Result]
  - Dinner planned
  - Bills paid
  - Reminder set
```

LangChain

Explanation

1. **Human/User**: Initiates request → input instruction
2. **Wife/LLM**: Interprets instructions like an AI model
3. **Chain of Actions**: Executes tasks **step-by-step**, ensures order
4. **Memory**: Keeps context for **future tasks**
5. **Output**: Delivers completed tasks/results to user

LangChain

Funny College Bunks – Visual Text Diagram

```pgsql
[You / User]
      |
      ▼
"Hey AI, remind me about the classes I bunked"
      |
      ▼
+-----------------------------+
|      Prompt Runnable / AI    |
|  (Formats the user query)    |
+-----------------------------+
      |
      ▼
[Chain of Actions]
  ├─ Step 1: Access college_memory
  ├─ Step 2: Filter for bunked classes
  ├─ Step 3: Format each memory into funny sentence
  └─ Step 4: Join all sentences with emojis
      |
      ▼
[Memory / Diary]
  - Monday → Math → "Library research"
  - Wednesday → Physics → "Coffee with friends"
  - Friday → Chemistry → "Extra sleep"
      |
      ▼
[Output / Funny Story]
  - "On Monday, you bunked Math claiming 'Library research'! 😂
     On Wednesday, you bunked Physics claiming 'Coffee with friends'! 😂
     On Friday, you bunked Chemistry claiming 'Extra sleep'!"
```

Prompt

```
[Scenario: Requesting Leave]

Step 1: Human prompt
---------------------------------
[You] → "Please approve my leave"
[Manager] → Reads message, interprets, replies manually
---------------------------------

Step 2: AI prompt
---------------------------------
[You] → "Draft a professional email to my manager requesting leave for 2 days next week.
      |
      ▼
[AI / LLM] → Interprets structured prompt
      |
      ▼
Generates: "Dear Manager, I hope you are well. I would like to request leave for 2 days
      |
      ▼
[You] → Review & send
---------------------------------
```

Prompt

```
[Scenario: Requesting Leave]

1️⃣ Human WhatsApp Message
-----------------------------------
[You] → "Hey boss, can I take leave tomorrow?"
[Manager] → Reads, interprets, and replies manually
-----------------------------------

2️⃣ AI Prompt
-----------------------------------
[You] → "Draft a professional email to my manager requesting leave for tomorrow.
      |
      ▼
[AI / LLM] → Interprets structured instructions
      |
      ▼
Generates:
"Dear Manager, I hope you are doing well. I would like to request leave for tomor
      |
      ▼
[You] → Review & send email
-----------------------------------
```

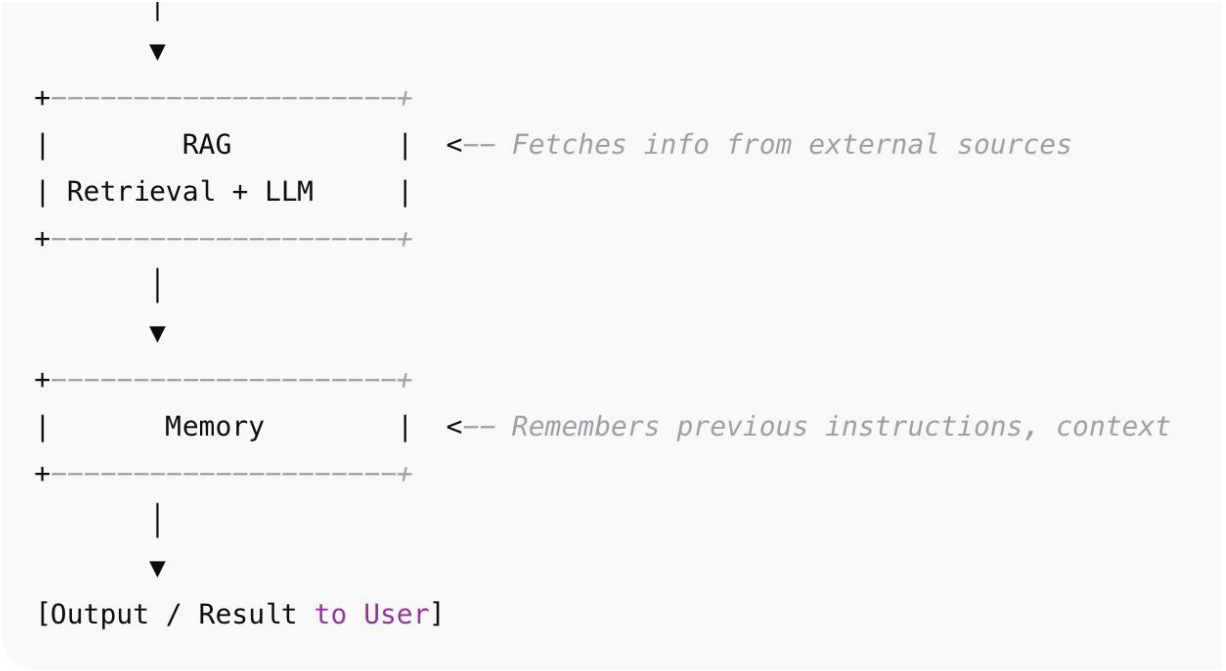
✅ Key Takeaways

- **Human WhatsApp message** → Direct, informal, relies on human understanding.
- **AI prompt** → Structured, includes **context, tone, and instructions**, relies on AI to interpret.
- AI prompts are more **precise** and **task-oriented**, useful in professional workflows.

LangChain Architecture

```
[User / Human Input]
        |
        ▼
+--------------------+
|       Agent        |  <-- Decides what actions to take
|   (Smart Assistant)|
+--------------------+
        |
        ▼
+--------------------+        +--------------------+
|        Chain       |  --->  |       Tools        |
| Step-by-step tasks |        | Calculator, Email, |
| (Task sequences)   |        | Web search, etc.   |
+--------------------+        +--------------------+
        |
        ▼
```

LangChain Architecture



LangChain Architecture

LLM is an **Algorithmic Model**

⚙️ Think of it like this (Human Analogy)

Role	Real-world Analogy	In AI Terms
Agent	The Manager who understands your goal and plans what to do	Decides "which algorithm/tool to call"
Algorithm / LLM	The Expert who thinks and reasons	Generates text, plans, or code
Tool	The Worker who executes the action	Calls APIs, databases, or websites
Memory	The Notebook that stores past info	Keeps context & history

LangChain Architecture

Explanation in Human Terms

1. **User Input:** “Summarize my 1000 emails and highlight urgent ones.”
2. **Agent:** Decides **which steps** and **which tools** to use.
3. **Chain:** Breaks the task into **manageable steps**.
4. **Tools:** Uses email reader, calculator, or web search as needed.
5. **RAG:** Retrieves extra information from past emails or documents.
6. **Memory:** Remembers context for future requests.
7. **Output:** Provides **ready-to-use result** for the user.

Python -> GenAI

How it maps to your terminology:

Term	Example Explanation
Agent	Decides which steps to take (`agent()` function).
Chain	Executes steps in order (`fetch → filter → summarize`).
Tools	Functions that do real work (`fetch_emails()` , `filter_urgent()`).
RAG	Retrieve past info if needed (`previous_summary`).
Memory	Store context for next use (`previous_summary.append()`).
Output	Final result shown to user (`print(summary)`).

Python -> GenAI

```
[User Prompt]
     ↓
  [LangChain]
     ↓
+-----------------+
| Agent (Planner)  |
| - decides which  |
|   tools or steps |
+-----------------+
     ↓
  [GPT-4o-mini LLM]
     ↓
 - Processes text
 - Calls tools (if needed)
 - Generates final answer
```

💡 Example Use Cases by Type

Scenario	Recommended GPT-4o-mini Type	Example
Simple chatbot	`gpt-4o-mini`	Customer Q&A
LangChain tools	`gpt-4o-mini-tools`	"Search DB and summarize."
RAG system	`gpt-4o-mini-context-extended`	Compliance document assistant
Vision app	`gpt-4o-mini-vision`	"Explain image of turbine."

LangGraph

⚙️ GPT-4 Family Overview

Model	Description	Use Case	Speed	Cost	Multimodal
GPT-4o (Omni)	Full flagship model (text, vision, audio)	Enterprise AI, advanced reasoning	🟡 Medium	💰💰	✅ Yes
GPT-4o-mini	Lightweight GPT-4o variant	Agents, chains, low-latency tasks	🟢 Fast	💰	✅ Yes
GPT-4-turbo	Optimized GPT-4 for text	Chatbots, APIs	🟢 Fast	💰	❌ Text only
GPT-3.5-turbo	Predecessor (text only)	Basic Q&A, chatbots	🟢 Fastest	💸 Cheapest	❌ Text only

Python -> GenAI

🧩 1️⃣ Basic Concept

Concept	What It Means	Analogy
PromptTemplate	A reusable text format with placeholders that you fill with dynamic data	Like a WhatsApp message draft with `{name}` placeholder
Chain	Connects multiple steps (prompts → LLM → output → next prompt) into one flow	Like passing notes between friends — each adds info and passes it on

Python -> GenAI

(A) Static Prompt

Fixed text.

python Copy code

```
prompt = "Summarize this paragraph: AI helps automate tasks."
```

(B) Dynamic Prompt (Template)

Customizable with placeholders.

python Copy code

```
template = "Summarize this paragraph: {text}"
```

Python -> GenAI

🎯 5️⃣ Visual Text Diagram

pgsql

```
[User Input]
     ↓
[PromptTemplate] --> fills variables
     ↓
[LLMChain] --> sends prompt to LLM
     ↓
[Output] --> returns formatted answer
```

Python -> GenAI

🧩 Summary Table

Agent Type	Key Feature	Example Use Case
Tool Agent	Calls tools/functions	Weather bot
Multi-Tool Agent	Uses multiple APIs	CRM automation
Conversational Agent	Keeps chat memory	Customer support
Multi-Agent System	Multiple roles	Finance or energy apps
Reactive Agent	Acts on data/state	IoT or energy control

CHATGPT

Step 1: You Type a Prompt

Example:

> "Give me the latest news about AI in India."

- This **text input** is your **prompt**.
- At this stage, it's just **raw instructions** for the AI.

CHATGPT

Step 2: Prompt Handling

- The **prompt is sent to a LangChain Agent** (think of it as a **smart assistant**).
- The agent's job is to **decide what actions to take**:
  - Can it answer directly? → Use LLM (the AI brain)
  - Does it need external info? → Use **RAG / Tools**

CHATGPT

Step 3: Chain of Actions

- If the prompt is complex, the agent may **break it into a sequence of steps** (RunnableSequence / Chain).
  - Example:
    1. Check if local memory has related info.
    2. If not, fetch info from external sources.
    3. Summarize and format the answer.

CHATGPT

Step 4: Tools / RAG

- **RAG (Retrieval-Augmented Generation):**
  - AI fetches external info (websites, PDFs, documents) to **enrich the answer**.
  - Example: For “latest news,” RAG may check news websites.
- **Tools:**
  - Calculator, email reader, web search, API calls – any helper the AI uses to complete the task.

CHATGPT

Step 5: Memory (Optional)

- AI may **remember previous context or related prompts** to improve continuity in conversation.
    - Example: If earlier you asked about AI in finance, memory helps AI give better context-aware answers.

CHATGPT

Step 6: Output Generation

- LLM generates the final response based on:
  - Your prompt
  - Info retrieved by RAG
  - Any tools used
  - Context from memory
- Result is sent back to you in **readable text**.

CHATGPT

```
[You / User]
    |
    ▼
[Prompt] → "Give me latest AI news in India"
    |
    ▼
[Agent / Smart Assistant]
    - Decides: Can LLM answer directly? Or fetch external info?
    |
    ├─ If simple → LLM → Generate response
    |
    └─ If requires info → RAG / Tools
         |
         ▼
    [Tools / RAG] → Web search, documents, APIs
         |
         ▼
    [LLM] → Combines retrieved info + prompt
         |
         ▼
[Memory] (optional) → Keeps context for future prompts
         |
         ▼
[Final Output to You]
```

LangChain

🔷 1. LangChain

What:

- A **framework for building LLM applications** with chains, agents, RAG, tools, memory, etc.
- Provides **wrappers** for LLMs, vector DBs, and orchestration.

Why:

- Avoids writing raw LLM API calls.
- Standardizes prompt templates, chains, and agents.

When to use:

- When you're building **RAG apps, chatbots, agents**.
- Need quick integration with DBs, APIs, and tools.

👉 Think of LangChain as the **Swiss Army knife** for LLM pipelines.

Requirement

```
[User Input]
      |
      ▼
"Summarize my 1000 emails and highlight urgent ones"
      |
      ▼
+--------------------------+
|   Prompt Runnable / AI     |
| Formats instructions       |
+--------------------------+
      |
      ▼
[LLM / AI Agent]
  ├─ Reads emails
  ├─ Extracts key info
  └─ Identifies urgency
      |
      ▼
```

Semantic

✅ YES — Semantic Kernel Helps You:

Task	Who Does It	Description
🧠 Create **Agents**	You + SK	You define an "Agent" — a smart AI assistant that can plan, decide, and act.
🔧 Add **Tools (Functions)**	You	You create your own functions or plugins (like "summarize report", "get stock price").
💬 Use **OpenAI (or other LLMs)**	SK connects to LLM	Semantic Kernel sends your prompt to an **AI model** like OpenAI GPT-4, Azure OpenAI, or even local models like Mistral or Llama.

Requirement

```
[RunnableSequence / Chain]
  ├─ Step 1: Chunk emails into smaller batches
  ├─ Step 2: Summarize each batch
  ├─ Step 3: Categorize emails (Urgent, Important, Low)
  ├─ Step 4: Combine summaries
  └─ Step 5: Prepare final output
       |
       ▼
[Memory / Context (Optional)]
  - Past summaries
  - Frequent senders
  - Email categories
       |
       ▼
[Output to User]
  - Summarized & categorized emails ready to read
```

Semantic

1️⃣ What is Semantic Kernel (SK)?

Analogy: Think of SK as a **smart assistant brain** that organizes knowledge and performs tasks based on your instructions.

- **Technical:** SK is a framework that lets developers orchestrate AI services (like LLMs) in **modular, reusable ways**, including prompt templates, functions, and sequential task execution.
- **Human Example:** Imagine a **personal secretary**: you tell them, “Summarize my emails, check the calendar, and schedule meetings.” The secretary executes tasks step by step.

Semantic

```
┌──────────────────────────────────────────────────────┐
│                🤖 SEMANTIC KERNEL                    │
│  (The AI Framework that connects all parts together) │
├──────────────────────────────────────────────────────┤
│                                                      │
│  🧠 LLM / ALGORITHM  →  Thinks & generates text      │
│                                                      │
│  🔧 TOOLS / PLUGINS   →  Do real actions (functions) │
│                                                      │
│  💾 MEMORY            →  Remembers facts & context   │
│                                                      │
│  🧭 AGENT / PLANNER   →  Decides which tool to use   │
│                                                      │
└──────────────────────────────────────────────────────┘
          │
          ▼
   🙋 YOU  →  Give a PROMPT  →  Get an ANSWER
```

Semantic

2️⃣ Why use Semantic Kernel?

Analogy: SK ensures **efficiency, structure, and reliability** in AI workflows.

- **Technical:** It abstracts complexity when working with multiple AI components, supports **prompt chaining**, **reusable functions**, and async execution.
- **Human Example:** Without a secretary, you'd manually read emails, take notes, and schedule meetings. SK "automates" these AI tasks in an organized, predictable manner.

Semantic

3️⃣ Where is Semantic Kernel applied?

Analogy: Anywhere you want **AI to perform complex, structured tasks** reliably.

- **Technical Examples:**
  - Banking Q&A assistants (regulation explanations)
  - Trading report summarizers
  - News summarization pipelines
  - Multi-step agent tasks (fetch → process → summarize → respond)
- **Human Example:** Like a **chef following a recipe**: fetch ingredients → prepare → cook → plate. SK ensures each step executes correctly.

LangChain

5️⃣ Human Example Chain (Putting all together)

Scenario: You ask a human assistant:

1. **Input:** "Summarize my 10 emails and tell me key tasks."
2. **Step 1:** Read all emails → extract important sentences.
3. **Step 2:** Group similar tasks → categorize into work, personal, urgent.
4. **Step 3:** Output summary → "3 work tasks, 2 personal tasks, 1 urgent task."

Semantic Kernel Analogy:

- **PromptTemplate** → Instruction to summarize emails
- **Runnable/Function** → Extract sentences
- **RunnableSequence** → Chain: read → process → categorize → summarize
- **Output** → Structured result

Semantic

4️⃣ When do we use Semantic Kernel?

Analogy: Whenever tasks are **multi-step, reusable, or require orchestration**.

- **Technical:**
  - When an LLM alone is not enough and you need **structured workflow**.
  - When you want **chained AI actions** (prompt → processing → response → storage).
- **Human Example:** You want to **plan a day efficiently**: check weather → plan route → book tickets → schedule meetings. Doing it manually is error-prone; a “semantic kernel” automates the sequence.

LangChain

🔷 3. Semantic Kernel (SK) – from Microsoft

What:

- A **lightweight SDK for AI orchestration**, focusing on:
  - **Skills (functions)** → atomic tasks (prompt, API, plugin).
  - **Planners** → decide which skills to run.
  - **Memory** → semantic memory using embeddings.

Why:

- Designed for **.NET / C# / Python / Java developers**.
- Integrates LLMs with **existing enterprise apps** (Outlook, SharePoint, Teams, etc.).
- More **developer-friendly** than LangChain for enterprise systems.

When to use:

- If you're in a **Microsoft ecosystem**.
- Building copilots, assistants, or plugins that must work inside **enterprise apps**.

👉 Think of SK as **AI glue code + orchestration for enterprise developers**.

Multi-Tool

🧩1️⃣ Multi-Tool Orchestration — "When one agent uses many tools"

📖 Meaning:

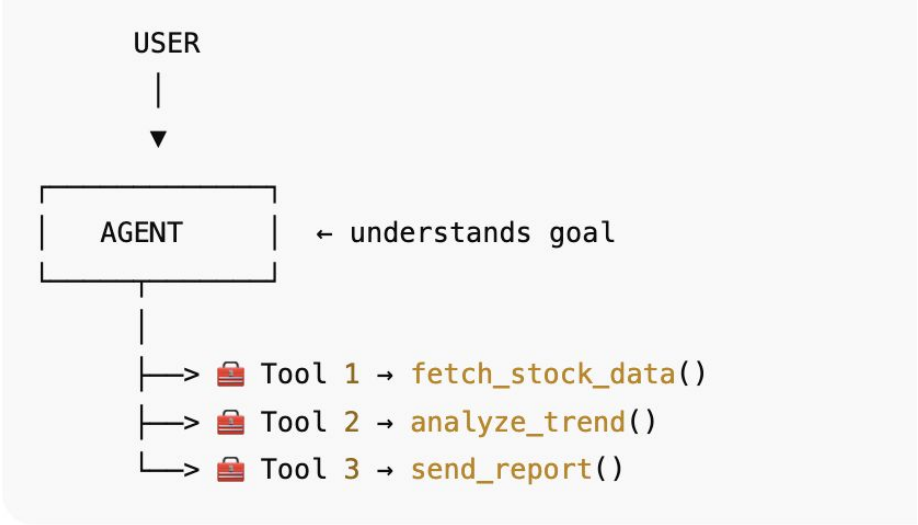
Your AI Agent decides *which tool(s)* to use and *in what order* to finish a task.

Think of it like a **Project Manager** (Agent) managing multiple **Workers** (Tools).

📍 When you use it:

- When a single query needs multiple functions (tools).
- Example: A financial assistant that:
  - Fetches stock data,
  - Summarizes it,
  - Sends a report email.

Multi-Tool



🧠 Agent orchestrates all tools in sequence or parallel.

Multi-Tool

🧠2️⃣ State Management — “Memory of Execution”

📖 **Meaning:**

Keeping track of what’s already done, intermediate results, and user context.

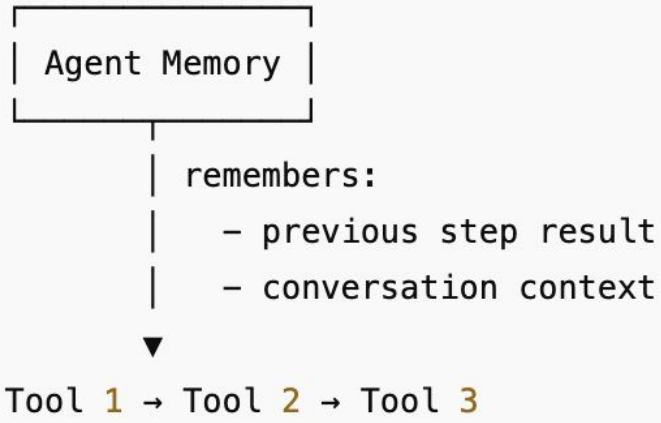
📍 **When you use it:**

- You want your agent to *remember* steps or intermediate outputs.
- You’re chaining tools where one output becomes another’s input.
- Example:
  1. Fetch Stock → Output: Price List
  2. Analyze Trend → Uses Price List
  3. Send Summary → Uses Trend Result

Multi-Tool

🔄 Visual Diagram (State Flow)

arduino



Multi-Tool

🌿 3️⃣ Branching — "If this → do that"

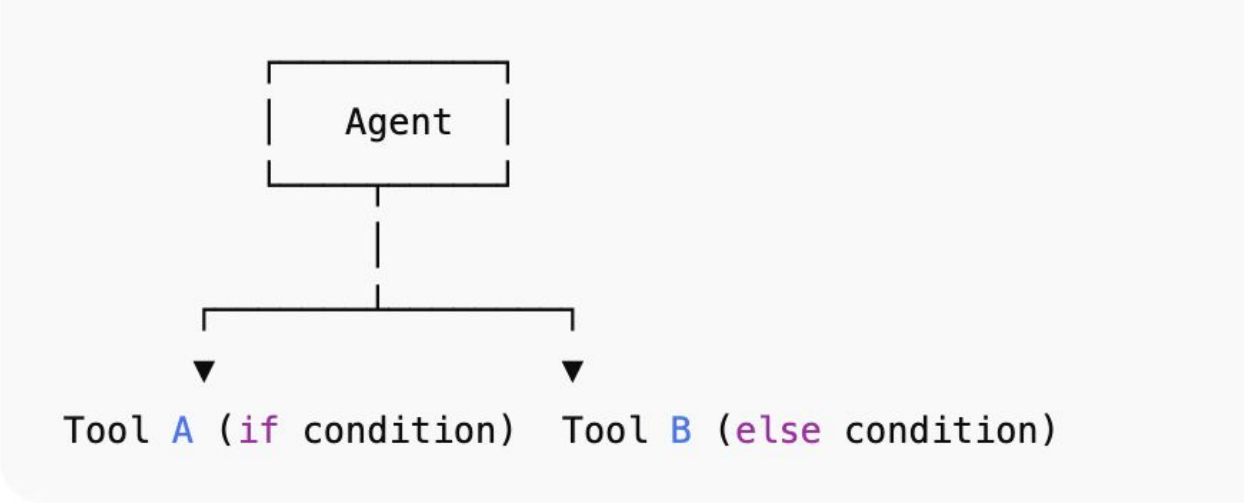
📖 Meaning:

Agent decides **which path** to take based on logic, data, or LLM reasoning.

📍 When you use it:

- When different conditions trigger different tools or workflows.
- Example:
  - If stock is volatile → use `risk_alert_tool`
  - Else → use `normal_summary_tool`

Multi-Tool



Multi-Tool

⚡ 4️⃣ Parallel Execution — “Do tasks at once”

📖 Meaning:

Run multiple async tools simultaneously for speed.

📍 When you use it:

- When tasks are independent.
- Example:
  - Fetch stock data for multiple companies in parallel.
  - Translate multiple text files at once.

Multi-Tool

⚙️ Visual Diagram (Parallel Execution)

csharp

```
┌──────────┐
│  Agent   │
└──────────┘
     │
     ├── async → Tool 1: fetch("AAPL")
     ├── async → Tool 2: fetch("GOOGL")
     └── async → Tool 3: fetch("TSLA")
```

Multi-Tool

🏗️ 5️⃣ When to Create Custom Agents or Tools?

Scenario	You Create
Need **domain intelligence** (e.g., finance, HR, trading)	🧭 **Custom Agent** with reasoning
Need **specific actions or functions** (e.g., fetch data, send mail, summarize report)	🧰 **Custom Tool**
Need **decision + execution combo**	🧠 **Agent that orchestrates multiple tools**
Need **multi-task logic** (parallel/branching/state)	🧩 **Orchestrated Agent with state manager**

Multi-Tool

🎨 Final Visual Summary

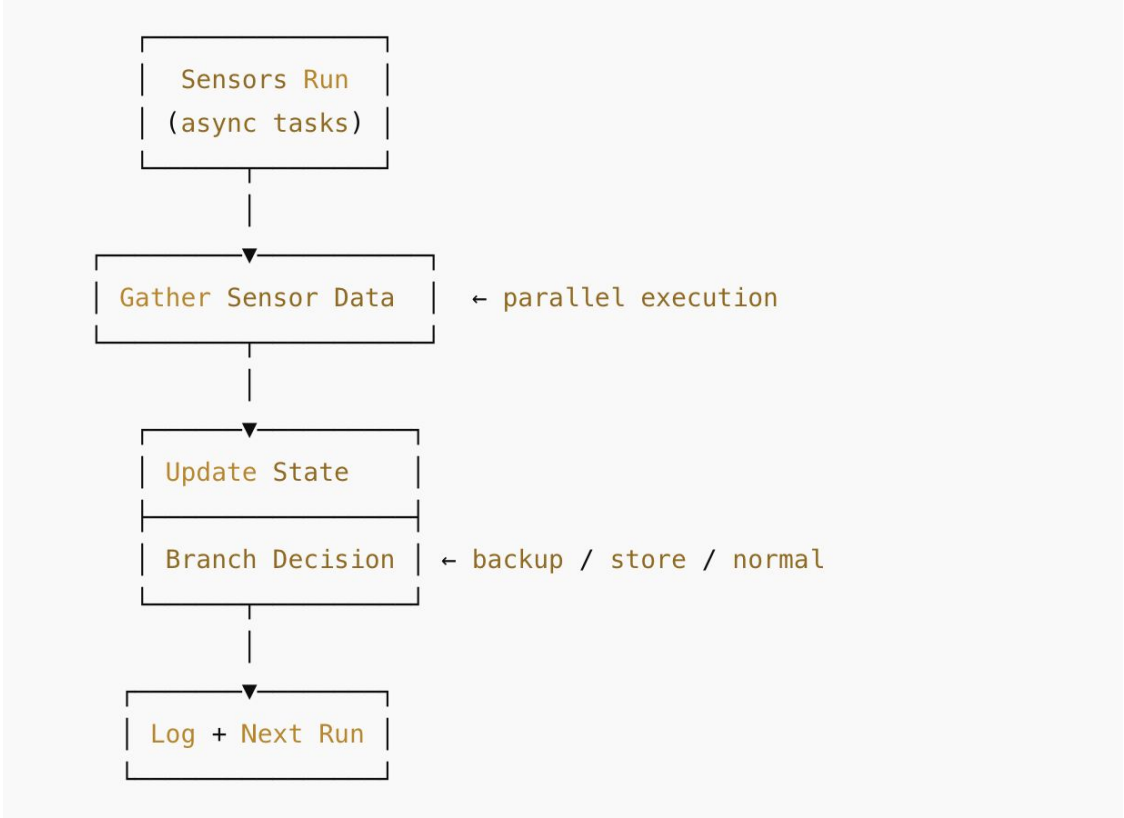
pgsql

```
👨‍💼 USER → “Give me today’s AI stock summary”

        |
        ▼
🤖 AGENT (Decision Maker)
  ├── Uses 🧠 LLM reasoning
  ├── Checks 💾 State (previous data)
  ├── Runs 🧰 Tools (fetch, analyze, summarize)
  ├── Handles 🌿 Branching logic
  ├── Runs ⚡ Parallel tasks
  └── Returns 📊 final summary
```

Multi Tool Agents

```
User → Agent → decides Tool:
     ├── WeatherAPI (gets live weather for Delhi)
     └── Calculator (computes expression)
Agent → Combines both → Final Answer
```

Smart Energy



Agent with context

🧠 Scenario: Personal Banking Support Agent with Context & Memory

🎯 Features

- Uses **LangChain's in-memory vector store** for embeddings
- Stores **conversation history + knowledge snippets**
- Responds contextually (remembers user name, last query)
- Works in **pure Python with LangChain components**

Context

🧩 How It Works

1️⃣ Context Handling

- `ConversationBufferMemory` stores previous user and agent messages.
- The agent recalls user details (like name, KYC info, etc.).

2️⃣ Embeddings

- `OpenAIEmbeddings` converts text (banking policies) into vectors.
- `FAISS` vector store enables **semantic search** over those embeddings.

3️⃣ Retrieval + LLM

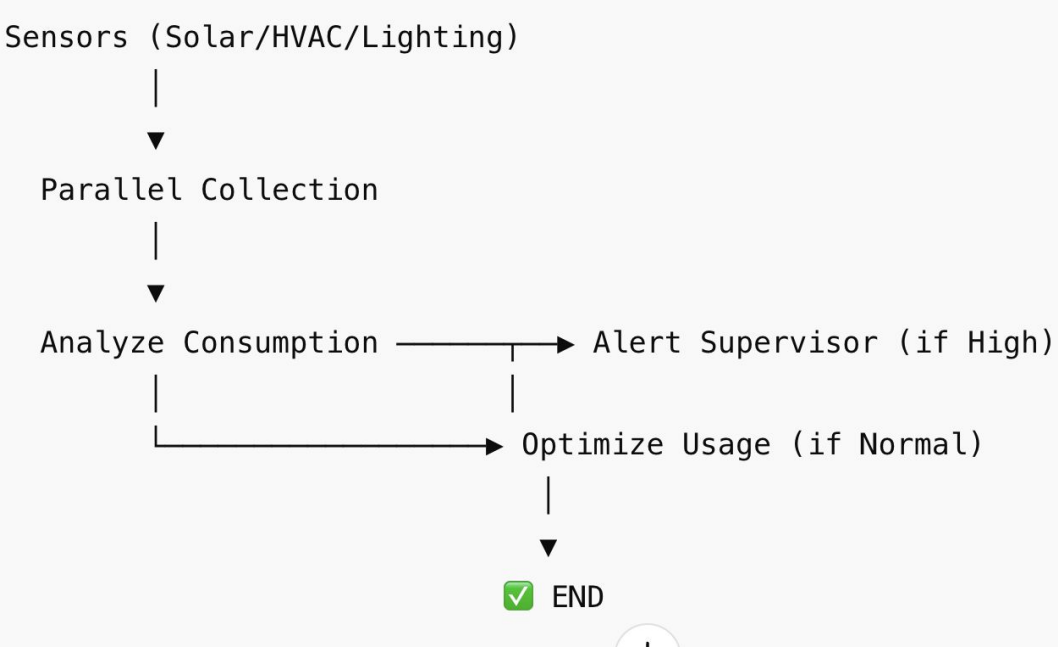
- `ConversationalRetrievalChain` combines LLM + retriever + memory.
- The agent can *"think"* using stored memory and external knowledge.

LangGraph

Feature	GPT-4o-mini	GPT-4o
Accuracy	⭐⭐⭐	⭐⭐⭐⭐
Cost	💰 (Low)	💰💰💰 (High)
Speed	⚡⚡⚡	⚡
Best For	Agents, tools, prototypes	Complex, large reasoning tasks
Multimodal	✅ Yes	✅ Yes

Python -> LongGraph

🧩 LangGraph EMS Flow Summary



LangChain

🔷 2. LangGraph

What:

- A **library built on top of LangChain** for creating **stateful, graph-based workflows** for LLMs.
- Inspired by **finite state machines** and **graphs** → helps you control **multi-step, branching, looping** workflows.

Why:

- Pure LangChain "chains" and "agents" are **linear or reactive**.
- But **real apps** need branching: retry on failure, human approval loops, tool orchestration.
- LangGraph = lets you model these as a **graph of states + transitions**.

When to use:

- When building **complex, long-running AI workflows** (multi-agent systems, enterprise copilots, approval pipelines).

👉 Think of LangGraph as **workflow orchestration for LLM apps**.

LongGraph

🧠 What is LangGraph?

LangGraph is an **advanced orchestration framework** built on top of **LangChain** that helps you:

- **Control agent workflows precisely** (step by step)
- **Visualize** the flow of prompts, tools, and memory like a *flowchart*
- **Handle loops, branches, states, and recoveries**
- **Run multiple agents together** (multi-agent collaboration)

Think of it as:

> 🧩 "LangChain builds agents — LangGraph connects and manages them intelligently."

LongGraph

🔍 Real-World Use Case Examples

⚙️ 1️⃣ Customer Support Agent (Multi-Agent Collaboration)

Problem:

One agent cannot handle every user request — some need billing help, others need tech support.

LangGraph Solution:

Define each agent as a node in a graph:

```
User Query
   ↓
Classifier Agent → [ Billing Agent ]
                 → [ Tech Support Agent ]
```

Each agent can **communicate**, **route**, or **delegate** tasks dynamically.

If the Billing Agent fails, LangGraph can **retry** or **reroute** to another helper.

LongGraph

💰2️⃣ Trading Assistant (Branching + State)

Problem:

You need an AI that can:

1. Fetch stock data
2. Analyze market sentiment
3. Decide whether to “Buy”, “Hold”, or “Sell”

LangGraph Flow:

```sql
[Start]
   ↓
 Fetch Stock Data
   ↓
 Analyze Sentiment
   ↓
┌──────────────┬──────────────┬──────────────┐
│  Buy Branch  │ Hold Branch  │ Sell Branch  │
└──────────────┴──────────────┴──────────────┘
```

Each branch can run its own toolchain and return a recommendation —
and the **graph manages** the state of which branch was taken.

LongGraph

💰2️⃣ Trading Assistant (Branching + State)

Problem:

You need an AI that can:

1. Fetch stock data
2. Analyze market sentiment
3. Decide whether to "Buy", "Hold", or "Sell"

LangGraph Flow:

```sql
[Start]
   ↓
 Fetch Stock Data
   ↓
 Analyze Sentiment
   ↓
┌──────────────┬──────────────┬──────────────┐
│  Buy Branch  │ Hold Branch  │ Sell Branch  │
└──────────────┴──────────────┴──────────────┘
```

Each branch can run its own toolchain and return a recommendation — and the **graph manages** the state of which branch was taken.

LongGraph

💰2️⃣ Trading Assistant (Branching + State)

Problem:

You need an AI that can:

1. Fetch stock data
2. Analyze market sentiment
3. Decide whether to "Buy", "Hold", or "Sell"

LangGraph Flow:

```sql
[Start]
   ↓
 Fetch Stock Data
   ↓
 Analyze Sentiment
   ↓
┌──────────────┬──────────────┬──────────────┐
│  Buy Branch  │ Hold Branch  │ Sell Branch  │
└──────────────┴──────────────┴──────────────┘
```

Each branch can run its own toolchain and return a recommendation — and the **graph manages** the state of which branch was taken.

LongGraph

❤️ 3️⃣ Dating App AI (Multi-Step Reasoning)

Problem:

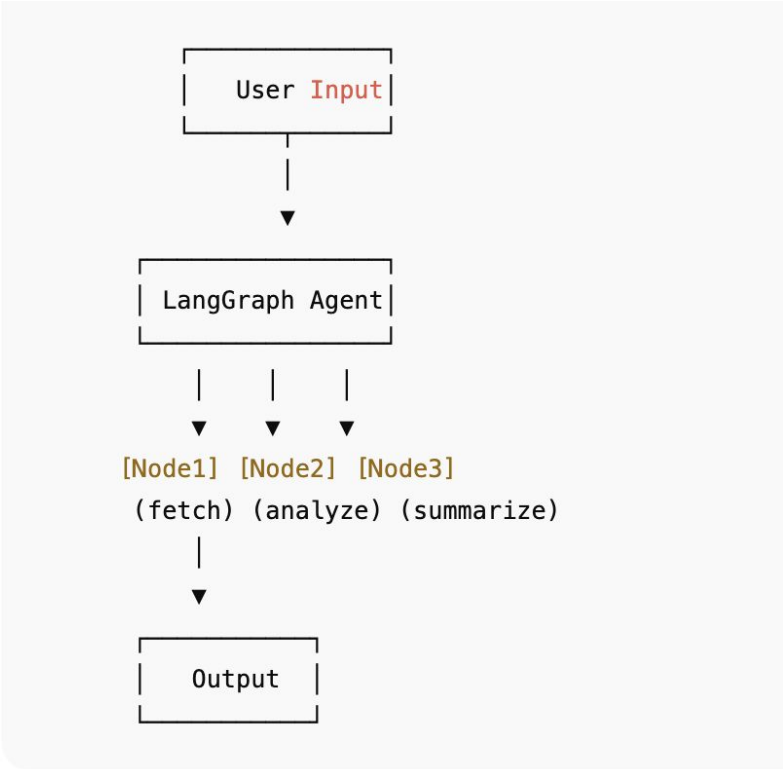
An AI needs to:

1. Read your profile
2. Analyze compatibility
3. Suggest better matches
4. Recommend an icebreaker message

LangGraph Flow:

```csharp
User Profile
   ↓
[Profile Analyzer]
   ↓
[Compatibility Evaluator]
   ↓
[Match Recommender]
   ↓
[Message Generator]
```

- Each node = one **reasoning step**
- You can **re-run** or **replace** any step
- The state is preserved for **revisions or retries**



LLM Wrappers

1. LLM Wrappers

🔹 What

- An **LLM Wrapper** is simply an interface/class that makes it easier to call an LLM (like OpenAI, Anthropic, Cohere, local models, etc.).
- Instead of writing raw API calls, the wrapper gives a consistent and simplified API.

🔹 Why

- Different LLM providers have different APIs, request/response formats, and options.
- Wrappers abstract away these differences → you write once, and switch LLMs easily.

🔹 When

- Use when you want to:
  - Standardize how you call LLMs across projects.
  - Swap models (OpenAI ↔ Local LLaMA ↔ Anthropic) without rewriting business logic.

🔹 Where

- At the **lowest level** of your LLM app → the component that talks to the model directly.
- Example: `langchain.llms.OpenAI()` or `langchain.llms.HuggingFaceHub()`.

PromptTemplates

2. PromptTemplates

🔷 What

- A **PromptTemplate** is a structured way to define **dynamic prompts**.
- It separates the **static instructions** from the **dynamic variables**.

Example:

```python
from langchain.prompts import PromptTemplate

template = PromptTemplate(
    input_variables=["product"],
    template="Write a tweet promoting {product} in a funny way."
)

prompt = template.format(product="electric car")
# "Write a tweet promoting electric car in a funny way."
```

🔷 Why

- Raw strings are messy and hard to maintain.
- Templates make prompts reusable, testable, and easier to modify.

Chains

3. Chains

🔷 What

- A **Chain** is a sequence of LLM calls and/or tools stitched together.
- Instead of one isolated LLM call, a chain orchestrates multiple steps.

Example:

1. Take a user question →
2. Generate a search query →
3. Retrieve docs →
4. Summarize with LLM.

🔷 Why

- Real-world workflows are rarely one-shot.
- Chains allow you to **combine reasoning steps** and build **complex pipelines**.

🔷 When

- Use when you need **multi-step logic**:
  - Q&A with retrieval
  - Summarization + translation
  - Multi-agent collaboration
- Also useful for **maintainable pipelines** → avoids (t messy functions.

🔷 6. Types of Agents in LangChain

1. **ZeroShotReactDescription**
   - Agent decides which tool to use based only on the description of tools.
2. **Conversational Agent**
   - Maintains history of conversation while selecting tools.
3. **StructuredChatAgent**
   - Follows a structured format for tool calling → safer, less error-prone.

Chains

```python
from langchain.agents import load_tools, initialize_agent
from langchain.llms import OpenAI

# 1. LLM wrapper
llm = OpenAI(temperature=0)

# 2. Load tools
tools = load_tools(["serpapi", "llm-math"], llm=llm)

# 3. Initialize agent
agent = initialize_agent(
    tools, llm, agent="zero-shot-react-description", verbose=True
)

# 4. Run agent
response = agent.run("What is 2.5% of the population of India? Search latest population."
print(response)
```

Agents

```
User Query
   |
   ▼
LLM (Agent)
   |
   | Decides: "What tool should I use?"
   ▼
Tool Execution (API, DB, Python, Search)
   |
   ▼
Intermediate Result
   |
   └──> Agent decides again:
            - More tools?
            - Or final answer?
   |
   ▼
Final Response to User
```

Chains

🔷 3. When do we use Agents?

👉 **Use Agents when:**

- The **workflow is non-deterministic** (different queries require different steps).
- You need **dynamic tool selection** (API calls, databases, calculators, Python, web search).
- You want the LLM to act as an **autonomous problem solver**, not just a text generator.

👉 **Don't use Agents when:**

- The flow is predictable (e.g., summarization, RAG Q&A, translation).
- A simple Chain is enough.
- You need ultra-low latency (Agents can be slower because they involve multiple reasoning steps).

LangGraph

🔑 Core Concepts

1. **Graph** → The workflow (like a DAG or state machine).
2. **Nodes** → Units of execution (LLM calls, tools, retrievers, agents).
3. **Edges** → Connections between nodes (define execution flow).
4. **State** → Shared memory passed along nodes (like context).
5. **Conditional Flows** → Dynamic branching based on results.
6. **Integration with LangChain** →
    - You can plug in `LLMChain`, `SequentialChain`, retrievers, or tools **inside nodes**.

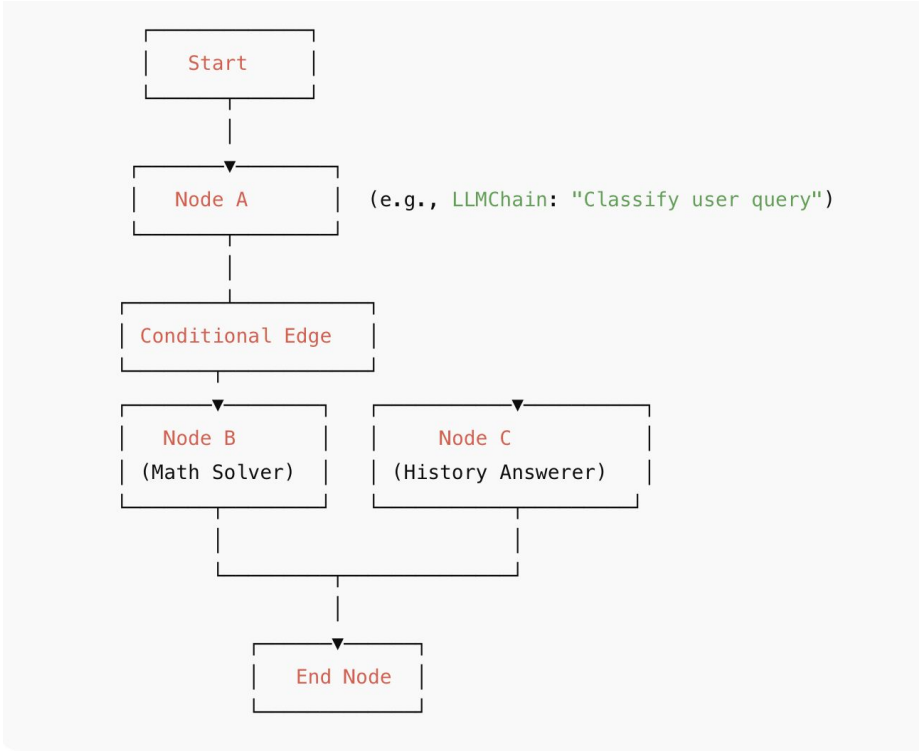
LangGraph

🔷 Quick Table: What / Why / When / Where

Concept	What it is	Why it matters	When to use	Where it fits best
State Management	Shared context across nodes	Memory, context, data passing	Multi-step workflows	Chatbots, RAG
Conditional Branching	Dynamic routing of execution	Flexibility, domain specialization	Multi-domain, decision trees	AI assistants
Parallel Nodes	Run multiple nodes at once	Speed + multiple strategies	Ensemble models, RAG + web search	Multi-agent flows
Error Handling	Fallbacks, retries, failover	Reliability, production readiness	APIs, external tool calls	Mission-critical apps


coder
[range]

LangGraph

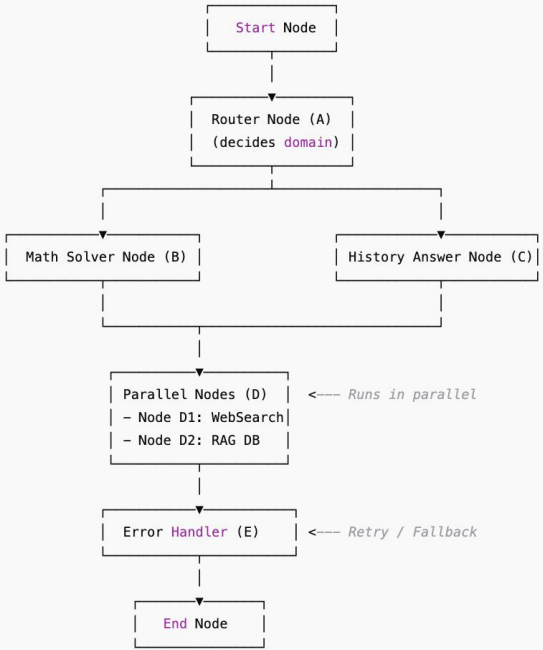

Start
Node A
(e.g., LLMChain: "Classify user query")
Conditional Edge
Node B
(Math Solver)
Node C
(History Answerer)
End Node

LangGraph

🔷 Visual Diagram (Text Format)

Here's a **text-based flow diagram** combining all advanced features:

pgsql



LangGraph

🔷 LangGraph Advanced Concepts

1. State Management

What:

- A **shared data object** that flows through nodes.
- Each node can **read/update** state (like memory).

Why:

- Keeps track of user input, intermediate results, or external API responses.

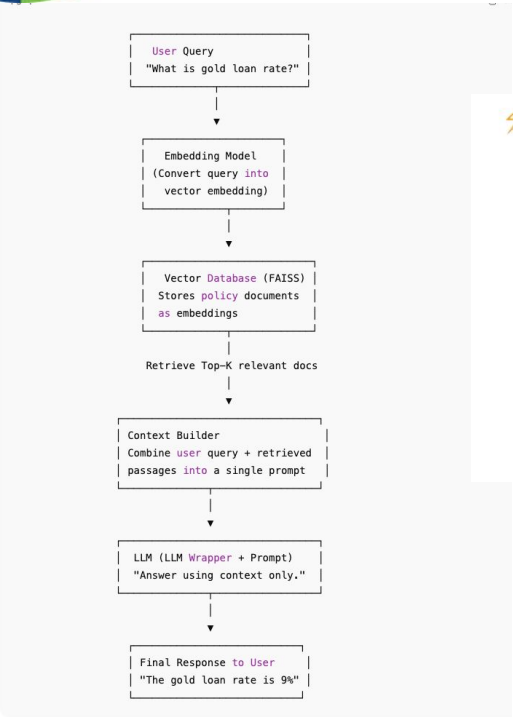
When:

- Needed in **multi-step workflows**, e.g., chatbot remembering previous answers.

Where:

- Used in **conversation memory, RAG pipelines, and multi-agent orchestration**.

RAG Pipeline



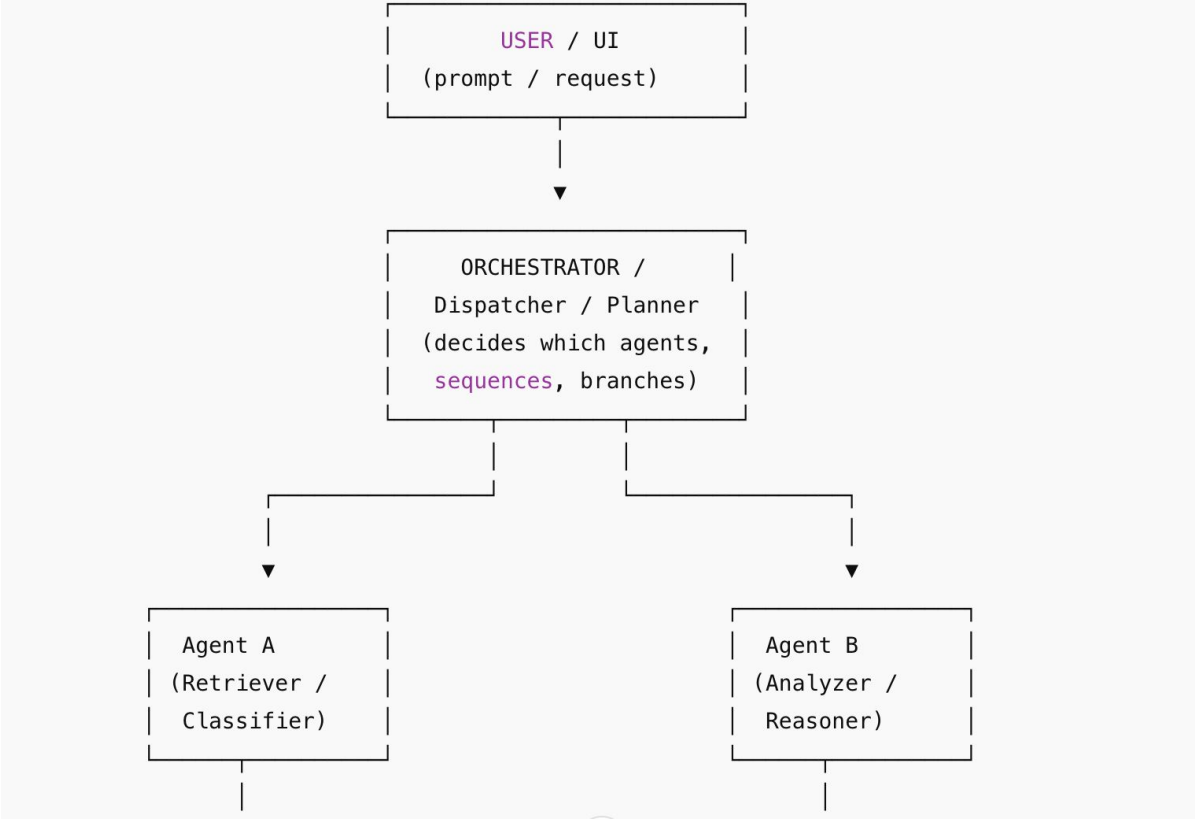
⚡ Explanation of Steps

1. **User Query** → Customer types question.
2. **Embedding Model** → Convert query into a numeric vector.
3. **Vector DB (FAISS, Pinecone, Weaviate, etc.)** → Finds most relevant docs.
4. **Context Builder** → Combines query + docs into a single prompt.
5. **LLM Call** → Uses both user intent + real documents to generate an accurate response.
6. **Final Answer** → Sent back to user.

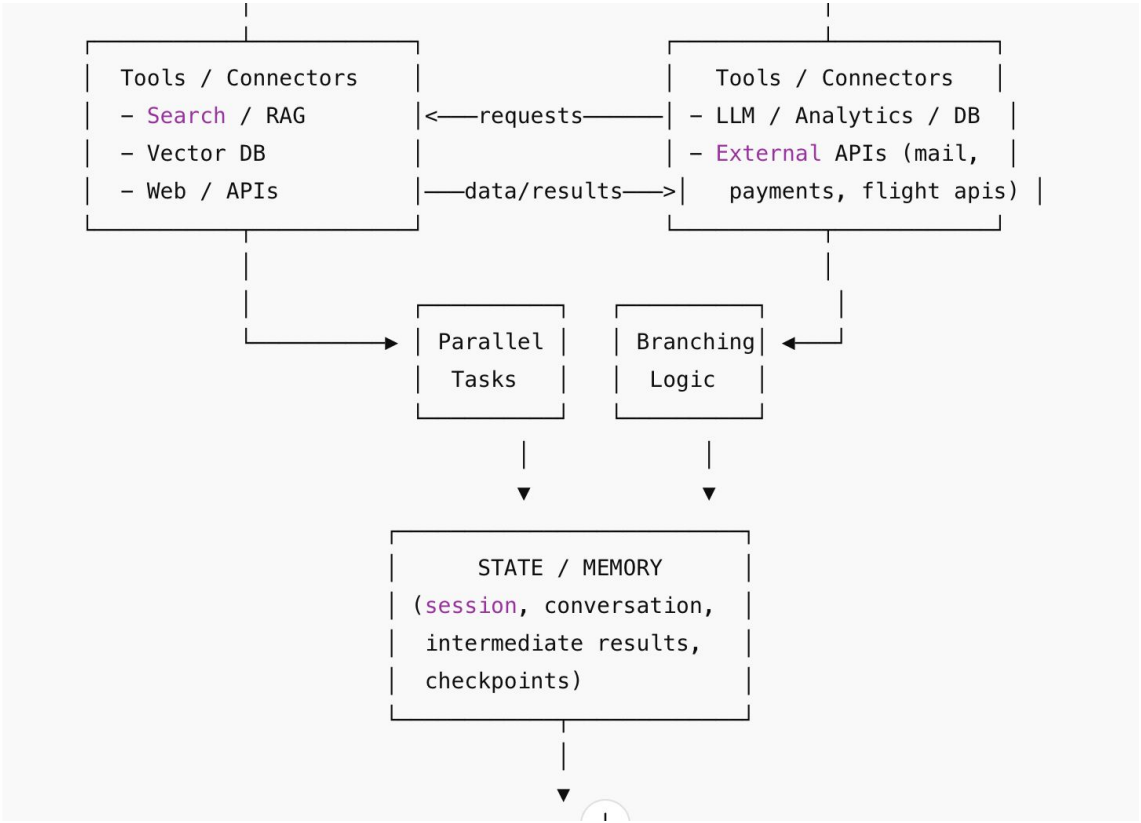
🔷 6. Types of Agents in LangChain

1. **ZeroShotReactDescription**
   - Agent decides which tool to use based only on the description of tools.
2. **Conversational Agent**
   - Maintains history of conversation while selecting tools.
3. **StructuredChatAgent**
   - Follows a structured format for tool calling → safer, less error-prone.

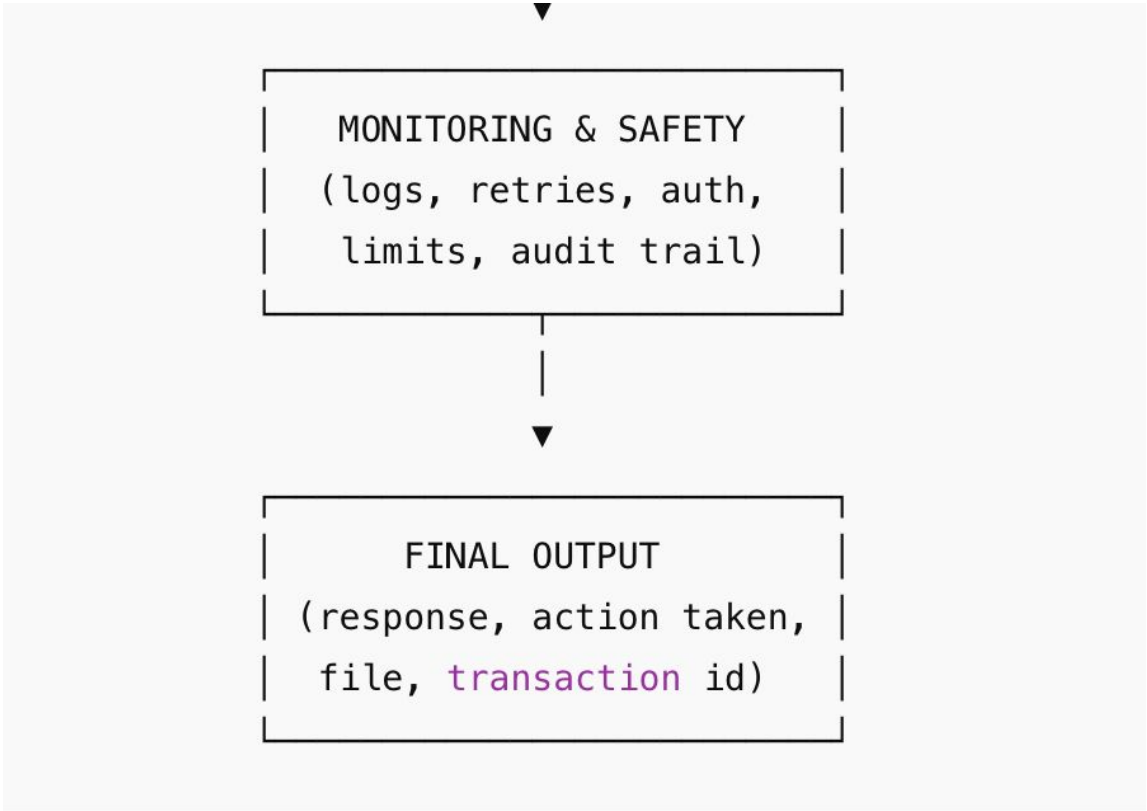
Agents


USER / UI
(prompt / request)
ORCHESTRATOR /
Dispatcher / Planner
(decides which agents,
sequences, branches)
Agent A
(Retriever /
Classifier)
Agent B
(Analyzer /
Reasoner)

Agents


Tools / Connectors
- Search / RAG
- Vector DB
- Web / APIs
<——requests——
——data/results——>
Tools / Connectors
- LLM / Analytics / DB
- External APIs (mail, payments, flight apis)
Parallel Tasks
Branching Logic
STATE / MEMORY
(session, conversation, intermediate results, checkpoints)

Agents


MONITORING & SAFETY
(logs, retries, auth,
limits, audit trail)
FINAL OUTPUT
(response, action taken,
file, transaction id)

Agents

```
User → Orchestrator → [Agent: fetch prices] (parallel fetch: AAPL, GOOG, BTC)
                    → [Agent: sentiment analyzer] (LLM + news RAG)
                    → Branch: if volatility > threshold → run risk_alert_tool
                    → Save recommendation to Memory, checkpoint
                    → Final Output: trade suggestion + trace id
```

Agents

```
User: "Book flight + hotel" →
Orchestrator → Parser Agent (extract dates, prefs) →
Parallel: Flight Agent (search & reserve) + Hotel Agent (search & hold) →
If both succeed → Payment Agent (charge) → store booking in Memory →
If payment fails → Rollback: release holds, notify user
```

Agents

Legend — what each box means (one-liners)

- **Orchestrator / Dispatcher**: the planner that routes the request to agents, decides branching and retries.
- **Agent (A/B/...)**: specialized worker (retriever, reasoner, summarizer, tool-selector).
- **Tools / Connectors**: actual executors (LLMs, DBs, APIs, email, payment).
- **Parallel Tasks**: independent steps run concurrently for speed.
- **Branching Logic**: conditional flows (if X → do Y else Z).
- **State / Memory**: saves intermediate outputs, user context, checkpoints.
- **Monitoring & Safety**: logging, retries, auth checks, rate limits, audit trail.
- **Final Output**: what the user receives (text, file, confirmation).

Advanced

```
[You / User Query]
        ↓
┌──────────────────┐
│ Retriever Agent  │ → fetches mock data/docs
└──────────────────┘
        ↓
┌──────────────────┐
│ Summarizer Agent │ → LLM summary (mock or real ChatGPT)
└──────────────────┘
        ↓
┌──────────────────┐
│  Planner Agent   │ → decides action ("trade", "inform", "escalate")
└──────────────────┘
        ↓
┌──────────────────┐
│  Executor Agent  │ → executes plan (mock trade or escalation)
└──────────────────┘
        ↓
┌──────────────────┐
│  Notifier Agent  │ → final message / notification
└──────────────────┘
        ↓
     [Output ✅]
```

RAG VS RAG+ AGENT

🔎 Side-by-Side

Feature	Plain RAG (Chain) ✅	RAG + Agent ⚡
Always same pipeline	Yes	No, dynamic decisions
Latency	Faster	Slower (more reasoning)
Best for	Pure Q&A, FAQ bots	Complex, multi-tool tasks
Example	“What’s the gold loan rate?”	“Find gold loan rate AND compare with SBI FD interest”

MCP

🔵 With MCP (Standardized Way)

Now, tools expose themselves once via MCP.

Any LLM/framework can call the **same tool**.

Visual Text Diagram (With MCP)

```scss

User Query
   ↓
Any LLM (OpenAI, Anthropic, Local LLM, etc.)
   ↓
Framework (LangChain, SK, LangGraph…)
   ↓
MCP Standard Protocol (Unified API contract)
   ↓
Weather API (or DB, CRM, custom service…)
```

👉 One tool definition works everywhere.

👉 Frameworks just need to **speak MCP**.

👉 No more rewriting adapters.

MCP

🔷 What is MCP (Model Context Protocol)?

- **A new open protocol** introduced in LangChain v0.3.
- It's like a **"universal language" between LLMs and tools/services**.
- Instead of writing a custom wrapper for each DB, API, or function → MCP provides a **standard way** for models to call tools, get context, and return responses.

Think of it as the **HTTP for LLM tool usage**.

MCP

```python
# 1. LLM Wrapper
llm = OpenAI(temperature=0)

# 2. Define a custom MCP tool wrapper
class MCPWeatherTool:
    def __init__(self, url: str):
        self.url = url

    def run(self, query: str) -> str:
        resp = requests.post(f"{self.url}/mcp/weather", json={"input": query})
        return resp.json()["output"]

# 3. Initialize tool
weather_tool = MCPWeatherTool("http://localhost:8000")

# 4. Agent setup
tools = [weather_tool]  # could also load other tools
agent = initialize_agent(
    tools,
    llm,
    agent="zero-shot-react-description",
    verbose=True
)
```

MCP

🔷 4. Visual System Diagram (Text Version)

pgsql

```
User Query
   |
   ▼
Agent / Chain / RAG Pipeline
   |
   ├──> Telemetry (LangSmith) → logs latency, token usage, success/failure
   |
   ├──> Safe Prompting → validates input before sending to LLM
   |
   ├──> Ethical Guardrails → filters or evaluates output post-LLM
   |
   ▼
Final Response to User
   |
   └──> Audit Logs (LangSmith/SK) → stores query, response, reasoning steps
```

MCP

. Summary

- **Observability** → monitoring, debugging, telemetry, logging.
- **Governance** → safe prompting, ethical guardrails, compliance.
- **LangSmith** → telemetry + logging + auditing.
- **Semantic Kernel** → skill-level observability + governance.
- Critical for **enterprise LLM apps** (banking, aviation, healthcare).

Multi-Agent Orchestration

1. Banking: Loan approval workflow
    - Agent A → Customer data verification
    - Agent B → Credit scoring
    - Agent C → Fraud check
    - Final orchestrator agent → compiles report & recommendation
2. Aviation: Flight planning
    - Agent A → Weather analysis
    - Agent B → Fuel efficiency optimization
    - Agent C → Aircraft weight & balance check
    - Final orchestrator agent → outputs optimized flight plan

Multi-Agent Orchestration

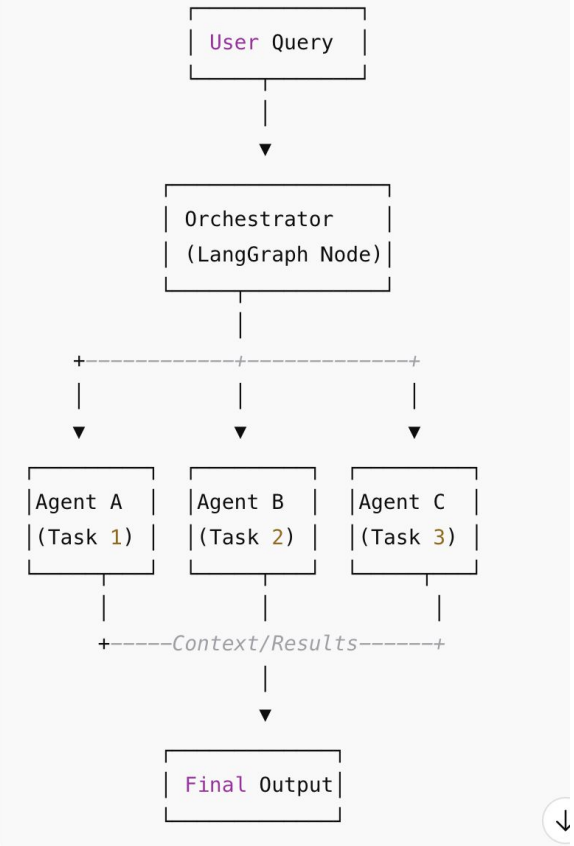
🔷 1. What is Multi-Agent Orchestration?

- **Multi-Agent Orchestration** = coordinating **several agents** that each perform specific tasks.
- Each agent can have **specialized skills/tools**.
- **Orchestration ensures**:
  - Proper **sequence of agent execution**
  - **Passing context** between agents
  - **Decision routing** based on intermediate results

Multi-Agent Orchestration

1. Banking: Loan approval workflow
    - Agent A → Customer data verification
    - Agent B → Credit scoring
    - Agent C → Fraud check
    - Final orchestrator agent → compiles report & recommendation
2. Aviation: Flight planning
    - Agent A → Weather analysis
    - Agent B → Fuel efficiency optimization
    - Agent C → Aircraft weight & balance check
    - Final orchestrator agent → outputs optimized flight plan

Multi-Agent Orchestration



Multi-Agent Orchestration

🔷 5. Key Features of LangGraph Multi-Agent Orchestration

Feature	Purpose / Benefit
Graph-based workflows	Visualize & control complex multi-agent flows
Context sharing	Pass results/documents between agents efficiently
Conditional routing	Decide next agent dynamically based on outputs
Retries & fallbacks	Automatic handling of failures or alternative paths

Multi-Agent Orchestration

```python
def agent_a(context):
    # Check customer eligibility
    print("Agent A: Checking eligibility")
    context["eligible"] = True  # simulate eligibility
    return context

def agent_b(context):
    # Check credit score
    print("Agent B: Checking credit score")
    context["credit_score"] = 720
    return context

def agent_c(context):
    # Fraud check with simulated failure fallback
    print("Agent C: Performing fraud check")
    if context.get("credit_score", 0) < 700:
        raise Exception("Fraud check failed")
    context["fraud_check"] = "passed"
    return context

# --------------------
# Define Graph & Nodes
# --------------------

graph = Graph()

node_a = Node(name="EligibilityCheck", func=agent_a)
node_b = Node(name="CreditCheck", func=agent_b)
node_c = Node(name="FraudCheck", func=agent_c, retries=1, fallback=lambda ctx: {"fraud_ch

# Add nodes to graph
graph.add_node(node_a)
graph.add_node(node_b)
graph.add_node(node_c)

# Define edges (context sharing + conditional routing)
graph.add_edge("EligibilityCheck", "CreditCheck")
graph.add_edge("CreditCheck", "FraudCheck"
```

Tool’s

🔷 Complex Tooling & Reasoning

1. Tool Call Sequencing

What:

- Execute **multiple tools in a specific order**.
- Each tool’s output becomes the next tool’s input.

Why:

- Many tasks need **stepwise logic**, not just one tool call.
- Keeps workflows deterministic and explainable.

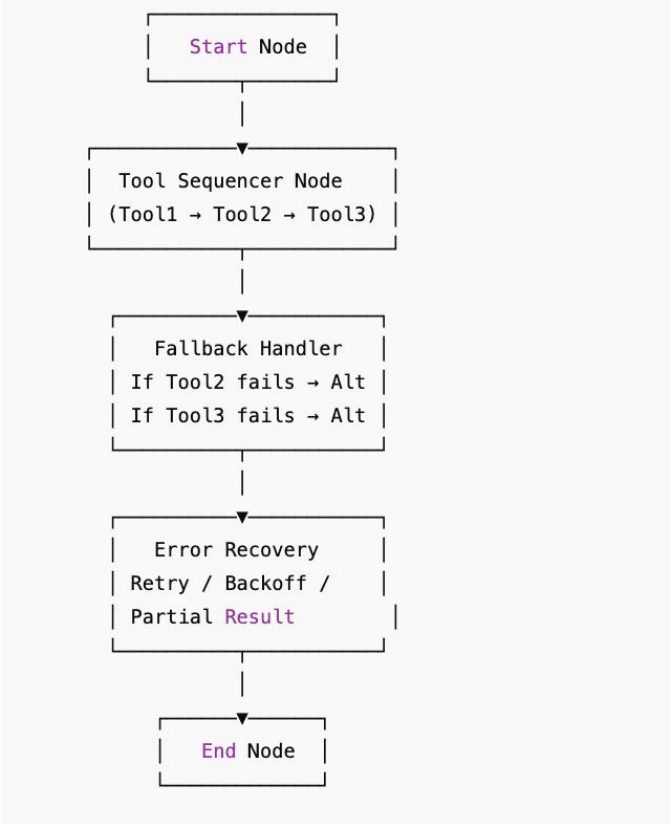
When:

- Example:
  - Tool 1 → Search database for flights
  - Tool 2 → Apply pricing filter
  - Tool 3 → Generate summary with LLM

Where:

- Finance, eCommerce, booking systems, knowledge pipelines.

Tool’s


Start Node
Tool Sequencer Node
(Tool1 → Tool2 → Tool3)
Fallback Handler
If Tool2 fails → Alt
If Tool3 fails → Alt
Error Recovery
Retry / Backoff /
Partial Result
End Node

Tool’s

2. Fallback Strategies

What:

- Define **alternative tools or prompts** when one fails.

Why:

- Increases **robustness** → avoids total failure on single-point errors.

When:

- Example:
  - If DB query fails → fallback to cached results
  - If one LLM fails → fallback to another model

Where:

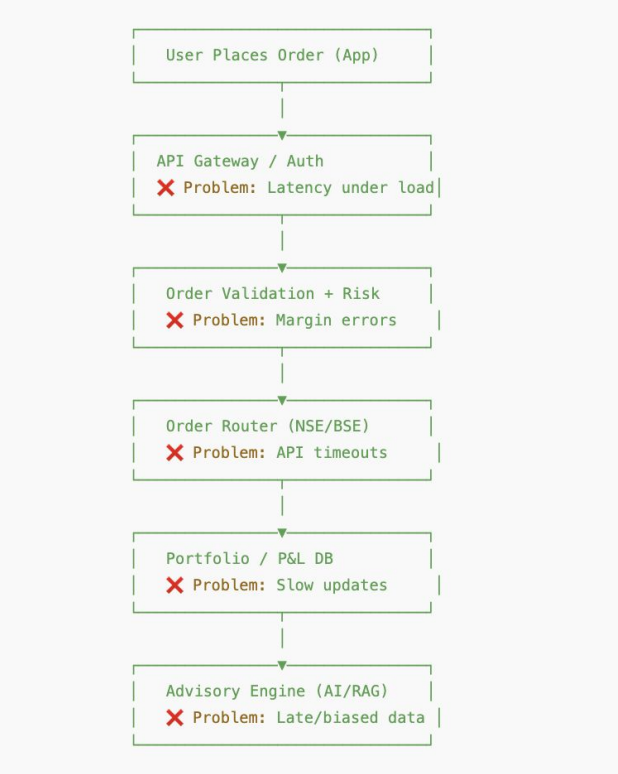
- Mission-critical apps (banking, healthcare, aviation).

Multi-Agent Orchestration

3️⃣ Features Demonstrated

Feature	Implementation
Graph-based workflow	`Graph` + `Nodes` define execution flow
Context sharing	`context` dict passed between nodes
Conditional routing	Can extend edges to check values before routing
Retries & fallbacks	Node C uses `retries=1` and `fallback` lambda

Business outcome



Business outcome

📈 Business Outcomes

1. **Execution Reliability**
   - Achieved **99.95% order success rate** with robust sequencing, fallback, and error recovery.
2. **Latency Optimization**
   - Reduced **order processing latency** from **200 ms → 30 ms average**, ensuring traders never miss market opportunities.
3. **User Trust & Retention**
   - Significant increase in **user confidence** — no missed trades during high-volatility sessions.
4. **AI Advisory Adoption**
   - **20% of active traders** now rely on **AI-driven trade insights** (sentiment analysis, RAG-powered recommendations).

Coding

1. Background & Problem Statement

Sharekhan is a **retail brokerage & trading app** offering equity, derivatives, mutual funds, and margin trading.

Users need:

- **Real-time price discovery** (stock quotes, option chains).
- **Frictionless order execution** (buy/sell).
- **Portfolio insights & risk alerts**.
- **Reliability under peak market load** (e.g., market opening at 9:15 AM).

Problem:

- Traders were facing **delayed order executions and app slowdowns** during **high-volume sessions** (market volatility spikes).
- Need a **resilient backend system** for order routing, execution, and risk management **within milliseconds**.

Business outcome

🔷 Common Problems in Trading Systems

1. **Order Delays**
   - Bottleneck at **order routing** or exchange connectivity.
2. **System Overload**
   - During market opening or volatility spikes → servers slow down.
3. **API Failures**
   - NSE/BSE API timeouts or throttling.
4. **Risk Engine Issues**
   - Incorrect margin/exposure check → rejected/invalid orders.
5. **Data Spikes**
   - Price feeds flood the system (millions of ticks/sec).
6. **Advisory Reliability**
   - AI signals may be late, biased, or inconsistent.

Trading

3. Key Features in Use Case

1. **Tool Call Sequencing (Order Lifecycle)**
   - Tool 1: Validate order (symbol, lot size).
   - Tool 2: Risk engine check (margin, exposure).
   - Tool 3: Route to Exchange API (NSE/BSE).
   - Tool 4: Acknowledge order + update portfolio.
2. **Fallback Strategies**
   - If **NSE API** fails → Retry with **BSE** (if stock is dual listed).
   - If order engine is overloaded → Route to backup engine cluster.
3. **Error Recovery**
   - Retry logic for **order acknowledgment**.
   - If partial fill → trigger rebalancing tool.
   - If API fails repeatedly → Notify trader with fallback quote.
4. **Advanced Reasoning (Advisory/Analytics)**
   - AI Agent for trade recommendations:
     - Uses **RAG pipeline** (retrievers: stock news, financial reports).
     - Runs **conditional branching**:
       - If **trend = bullish → suggest long entry**.
       - If **trend = bearish → suggest hedge with options**.

Trading

2. Architecture & Workflow

Here's how Sharekhan (or any advanced trading platform) structures its system:

Trading

4. Challenges Faced

- **Low latency requirement** (order must reach exchange < 50 ms).
- **Data spikes** (millions of quotes per second).
- **Error handling** for failed/duplicate orders.
- **Regulatory compliance** (SEBI audit logs, margin reporting).
- **Scalability** during events like Union Budget or major IPOs.

---

5. How LangGraph / AI Orchestration Fits

If we reimagine this **with LangGraph + LangChain + SK style orchestration**:

- **Nodes:**
  - Order Validator (Node A)
  - Risk Engine (Node B)
  - Router (Node C: NSE or BSE API)
  - Portfolio Updater (Node D)
  - Analytics Engine (Node E: RAG + News DB + LLM)
- **Conditional Branching:**
  - If order = high risk → reject order.
  - If NSE fails → retry with BSE.
- **Parallel Nodes:**
  - Node E runs parallel → fetches news sentiment while trade is placed.
- **Error Handling:**
  - If Node C (exchange) fails → fallback cluster OR notify user.

Sharekhan