Session#1
Overview of Agentic AI

Palacode Narayana Iyer Anantharaman

24th Nov 2025

Beyond Parrots and Calculators

- Generative AI gave us impressive fluency — but limited autonomy.
- Early LLMs behave like parrots (mimicry) and calculators (logic): reactive, stateless, and prompt-bound.
- **Agentic AI introduces a shift** — from responding to prompts, to achieving goals through planning, memory, and action.
- We explore how to design, build, and evaluate AI agents that **think, act, and adapt**.
- Going beyond the chat interface, we'll cover structured reasoning, tool use, context management, and safety.

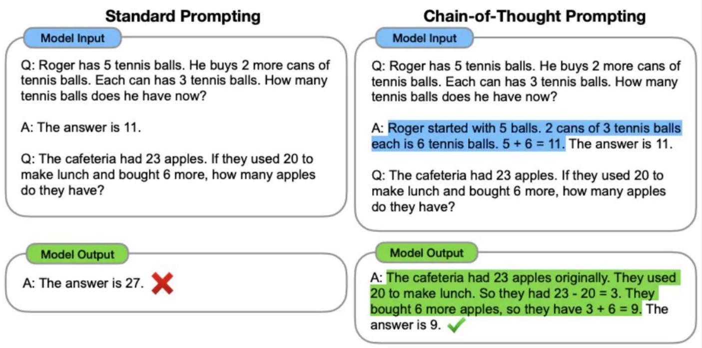
LLMs can mimic and compute—but agents must plan and act

AI, ML, DL, Generative and Agentic AI

- Goal of the AI is to build a machine that is as capable as a human
- Ability to think, reason and act like humans, *humanly and rationally* (Ref: Russel and Norvig)
- **Machine Learning** is about learning from data without being explicitly programmed
- **Deep learning** models are specialized ML architectures that rely on a large number of layers, complex models
- ML/DL form the foundation of **narrow AI** (also known as *weak AI*).
- **Generative AI**, moves beyond narrow functionality — creating novel content and enabling new interactions.

Rules → ML → DL → Gen AI → Agentic AI

Thinking Humanly	Acting Humanly
Thinking Rationally	Acting Rationally



The Next Frontier - Introducing Agency

- The next evolution in generative AI isn't just about better completions; it's about ***agency***.

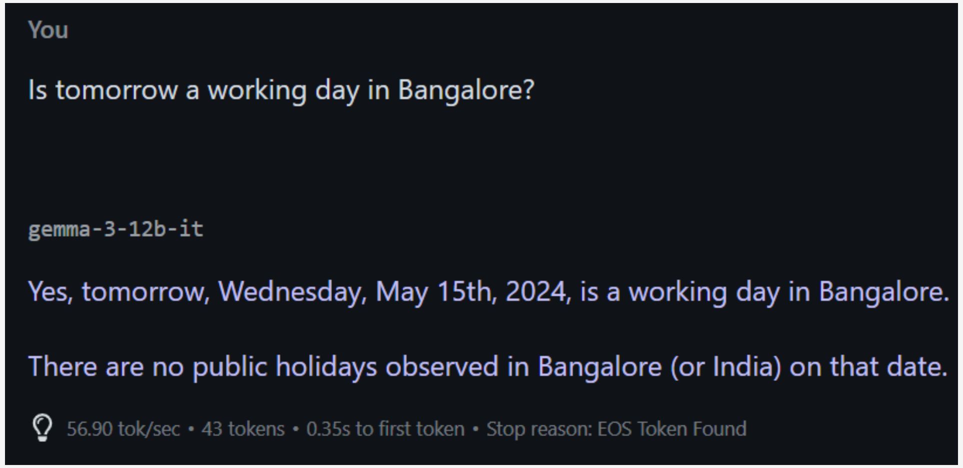
- Agency means autonomy, goal orientation, and adaptability.

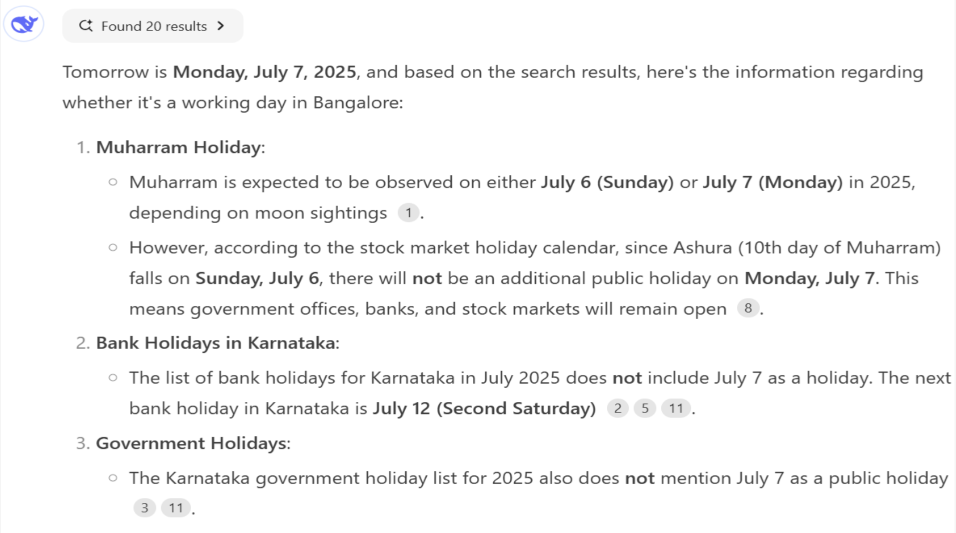
- New models hint at this trend, they have an important piece that supports agency: "**Reasoning**"

    - **DeepSeek R1:** Separates planning & reasoning from text generation for structured problem-solving.

    - **GPT-4o:** Integrates vision, text, and audio – enabling interaction with the world beyond text.

- These models still require architectural frameworks to achieve true agency. That is: Design Patterns, Frameworks and Tools, etc.





Democratization – the world of open source models

The rise of open-source models (Llama 3.x, gemma, Mistral AI) democratizes access to powerful LLMs.

Technical Details: Quantization reduces memory footprint and improves speed (GGUF, bitsandbytes).

Hardware Requirements: Significant RAM & GPU VRAM are needed for local execution.

Software Tools: llama.cpp, LMStudio, ollama simplify deployment.

Benefits: Enhanced privacy, customization, offline functionality, reduced reliance on external APIs.

From Models to Agents — What Makes an AI Agentic?

Autonomy: Operates with minimal human intervention to complete tasks.

Memory: Remembers past interactions, tasks, and state.

Goal-Directed Behavior: Acts with purpose, not just prompt-response.

Tool Use: Can use APIs, search, or other tools to complete objectives.

Context Awareness: Understands and reacts to surrounding context (task, history, environment).

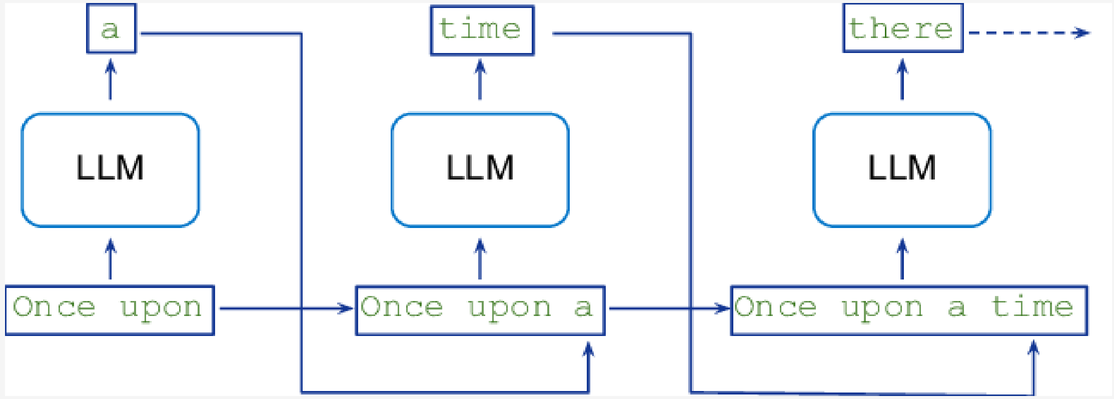
Reflection & Adaptation: Can revise plans or strategies mid-process.

Human-in-the-Loop (HITL): Allows feedback and oversight when needed.

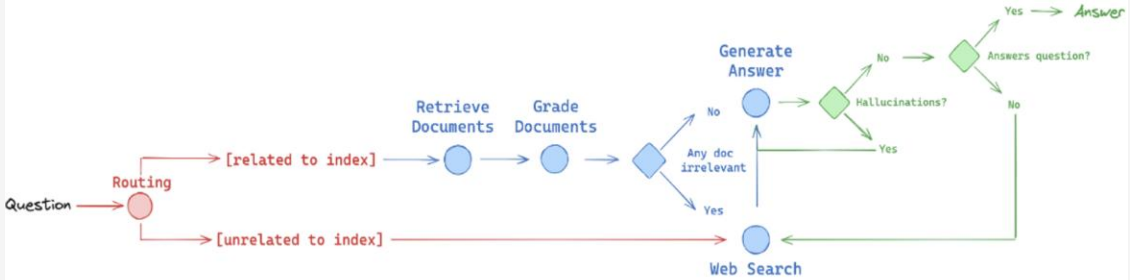
Architecture-Driven: Built using patterns like ReAct, AutoGen, LangGraph, CrewAI.

Traditional LLM applications versus Agentic

Traditional LLM Apps	**Agentic Workflows**
One-shot completions	Multi-step reasoning & memory
Reactive	Proactive & autonomous
No tool use	Tool orchestration
Stateless	Stateful with memory



LLM: Sequence (tokens) in Sequence (tokens) out



Agentic Workflows may involve feedback loops

Hands On – “Hello World” LLM!

- LMStudio
- Ollama
- OpenAI Protocol
- Custom Libraries

From Chatbots to Agents: Your First Steps in LLM-Powered Apps

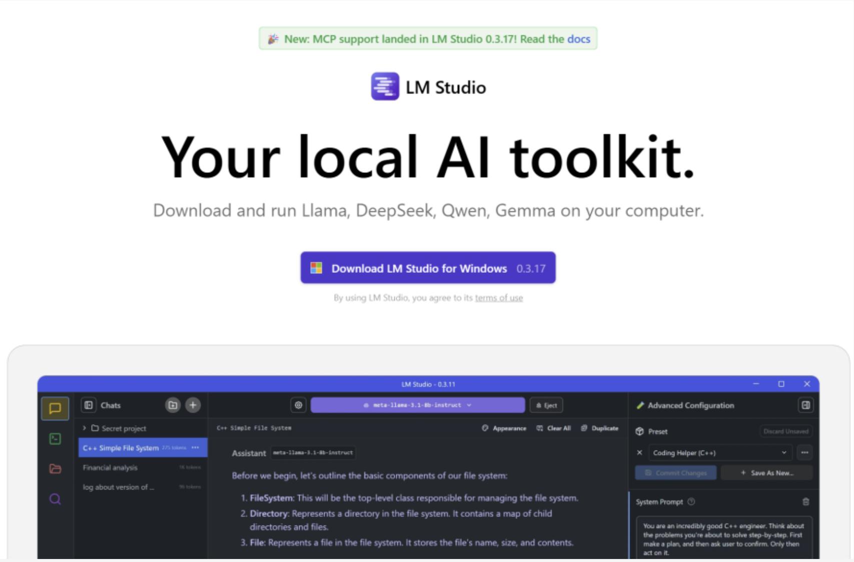
Getting Started

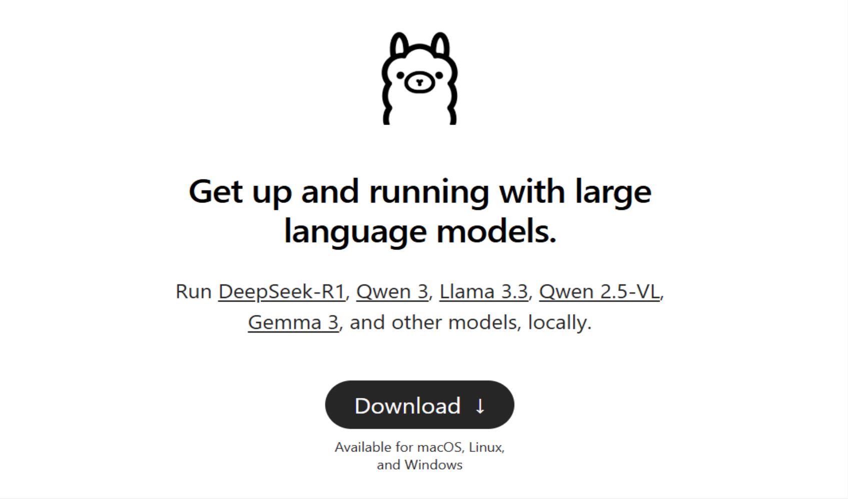
- Download LMStudio (or Ollama)
- Load an open source LLM like gemma 3 12B and run as a server
- Write a client code
- Test it with your prompts

Building your first Agentic AI

- Install OpenAI Agent SDK
- Refer the code and explanation from this session

Optionally, test both with a Streamlit based frontend (See demo)





LM Studio SDK

```
pip install lmstudio

import os
import lmstudio as lms

model = lms.llm()



def image_chat(image_path, prompt):
    chat = lms.Chat()
    image_handle = lms.prepare_image(image_path)
    chat.add_user_message(prompt, images=[image_handle])
    prediction = model.respond(chat)
    return prediction.content
```

Prompt based applications

No inherent memory

No native ability to interact with the external world

Very limited adaptability, context awareness

Prompt driven as opposed to goal driven: e.g. AI research assistant

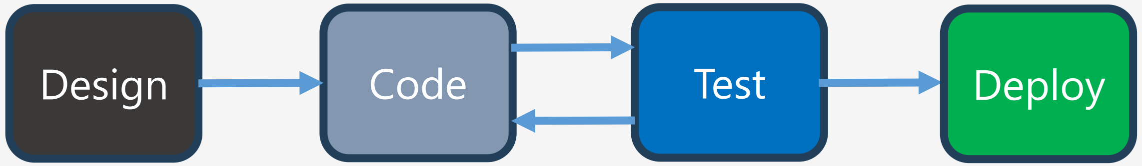
Creating Your First Agent using OpenAI Agents SDK

Refer the demos: demo*.py

Ensure OpenAI Agent SDK is installed

Workflow agents versus Autonomous Agents: Agentic Workflows

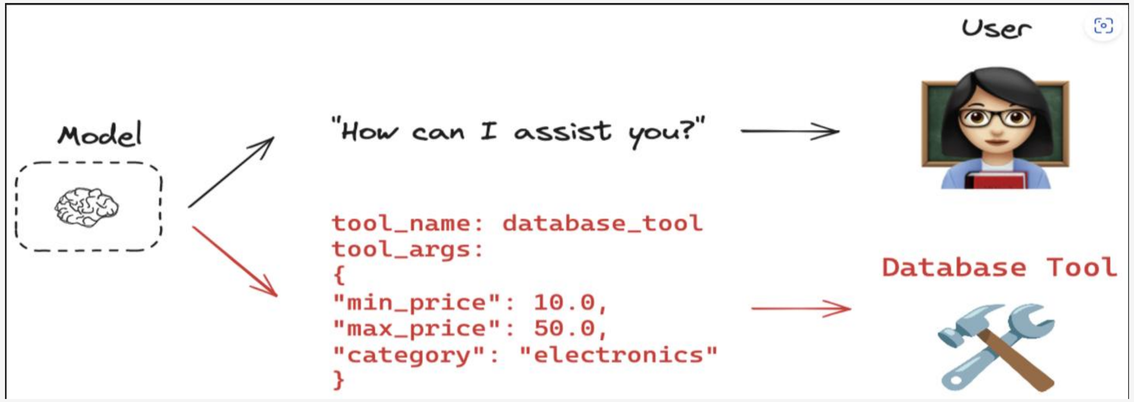
- **Linear, pre-scripted sequences**
  - Example: Generate Code → Run Tests → Manual Review → Deploy

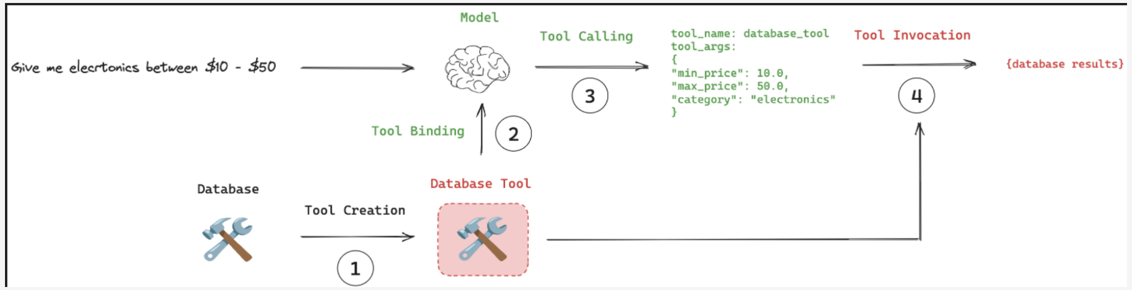


- **Optimized for**:
  - ✅ Predictability
  - ✅ Repeatability

- **Limitations:**
  - ❌ Break under unexpected scenarios
  - ❌ No decision-making or context awareness

- **Strength:**
  - ⚙️ Reliable for known, repetitive tasks





Agentic AI: Context Aware and Adaptive

Goal-driven and dynamic, not step-bound

Agent actions depend on real-time context:

- 👁 Observe → 🧠 Interpret → 🛠 Act → 🔄 Reflect → Adapt

Key strengths:

- ✅ Handle ambiguity
- 🔁 Retry or replan on failure
- 🧠 Learn from feedback and evolve

Examples: Reflection loops, ReAct, Tree-of-Thought

User Goal: "Notify me when there are highly rated electronics between $10 and $50 — purchase if my preferences are met."

- The LLM understands the request as a complete process: monitor inventory, filter by price/rating, and auto-purchase matching items
- Planner module creates a multi-step plan: Monitor inventory, Filter by Price and rating, cross check with user preferences, Trigger purchase and notify the user.
- Tool usage and reasoning loop
- Reflection: If no items are found after 3 days, revise the goal
- Trigger Action: Once criteria met, agent calls purchase API and confirms with the user
- Memory Update: Logs purchase history, refines preferences for future autonomy

Why this is Agentic AI?

Capability	Present in Agentic Flow?	Example
Autonomy	✅	Agent monitors & acts without prompting
Adaptability	✅	Adjusts strategy if no match found
Tool Use	✅	Product API, Purchase API
Planning & Retry	✅	Iterative search & fallback logic
Memory	✅	Remembers preferences & purchase history

Case Study: A real world problem

Problem Statement

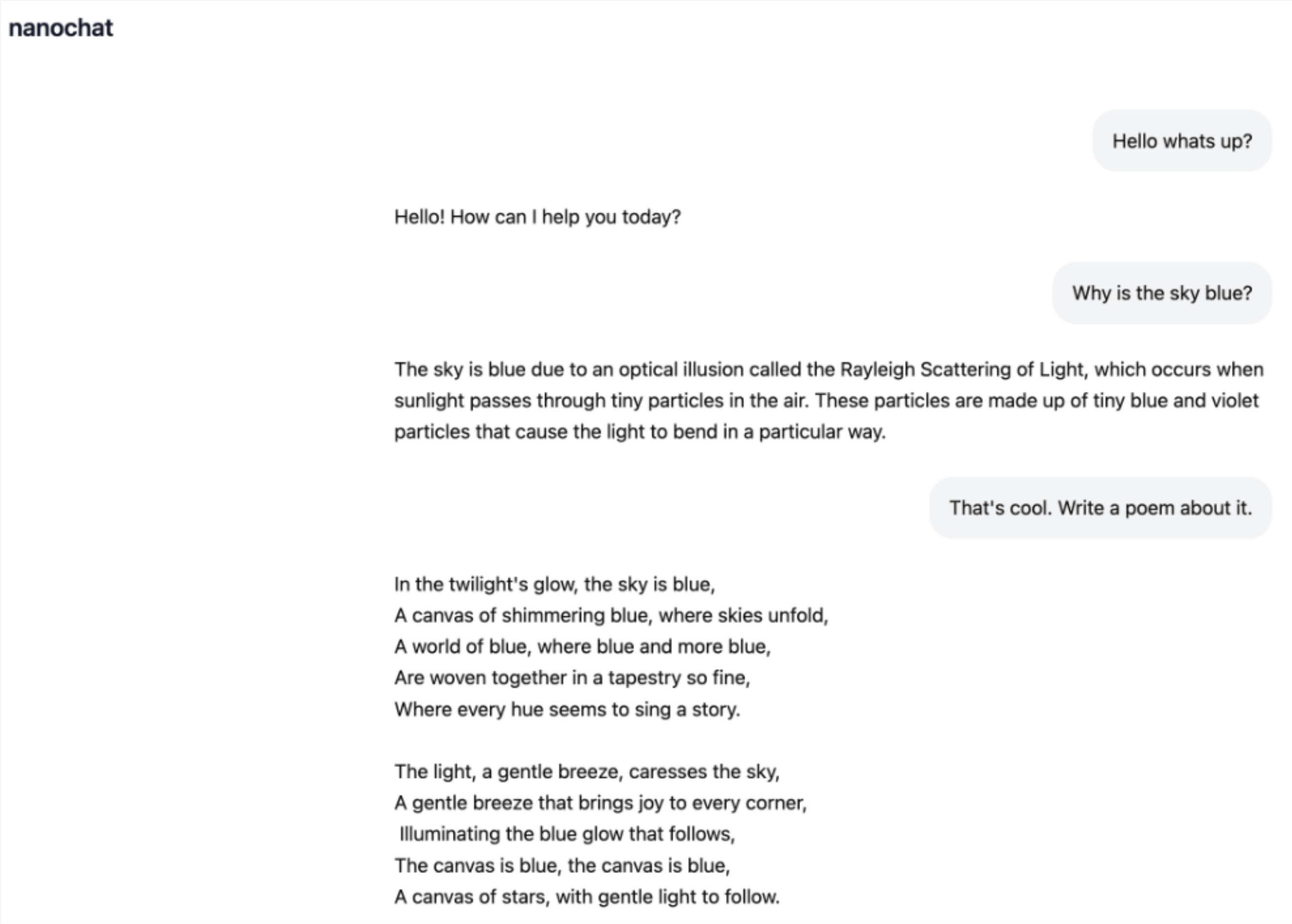
Imagine you are a new recruit to Adobe and are joining the Adobe Illustrator team. You're handed a large codebase with complex components. Normally it takes weeks to read, understand, and build learning materials.

What if we could teach an AI agent to read the codebase, map it, understand it, and generate a personalized learning curriculum for any learner—instantly?"

We built an **Agentic Application** that does exactly this, using a network of collaborating agents orchestrated with LangGraph

Our Codebase: Nanochat (https://github.com/karpathy/nanochat)

- Nanochat is a full stack implementation of an LLM from scratch
- Released recently by Andrej Karpathy
- Implements GPT 2 style LLMs
- It’s a great platform for LLM research and also implementing agentic principles



Architecture

We have 2 interacting graphs implemented in LangGraph (more on this later)

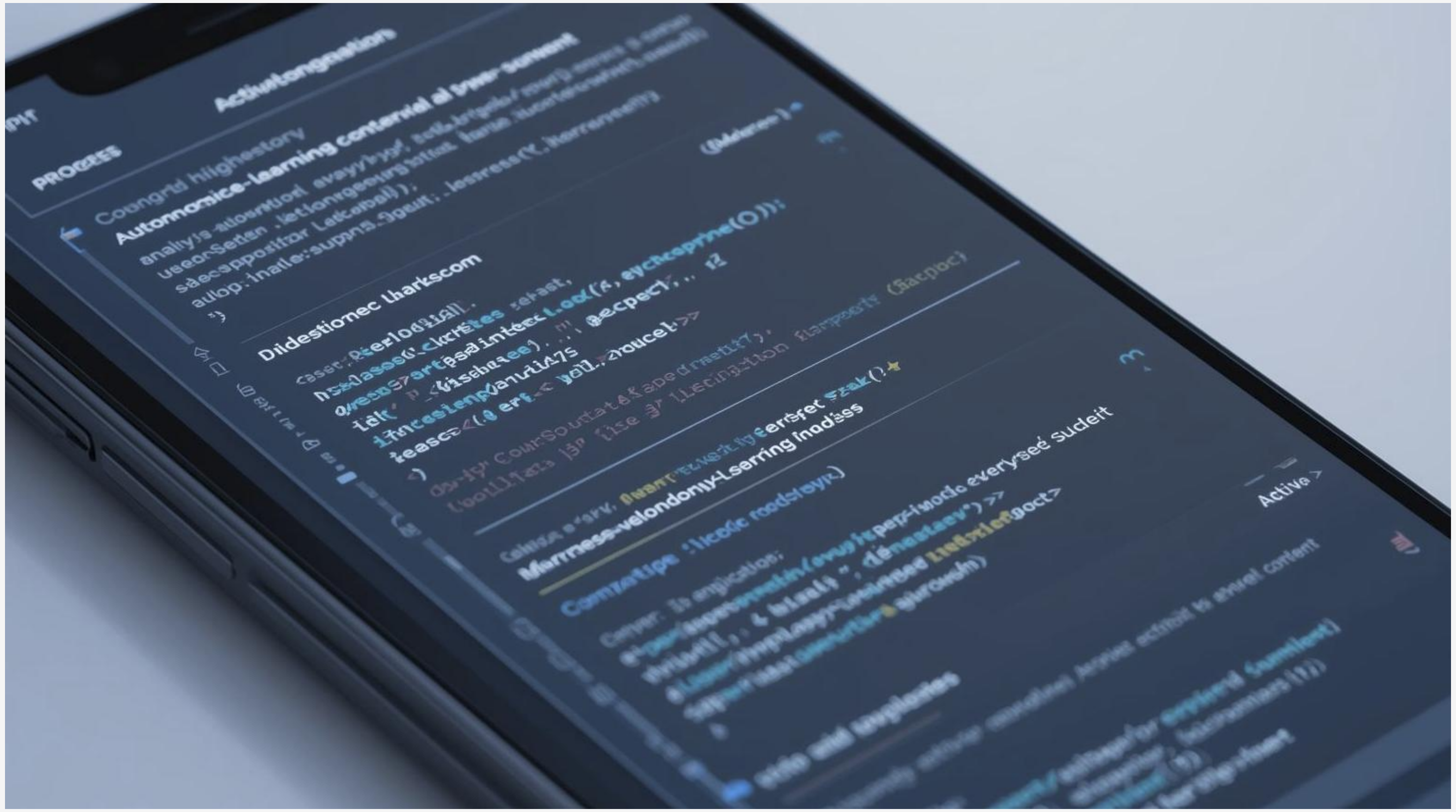
First graph "understands" the repo, creates an outline of a micro learning curriculum

The second graph turns each topic to a professional content with code examples, quizzes and code exercises

We will use this as our running case study to learn the core features of Agentic design

Demo: Agentic App for Micro learning Content Creation

- Autonomous Outline Generator



- Autonomous Topics content creator

Tool Use

Agents don't just reason — they act.

- Tools empower agents to interact with the external world: APIs, scripts, systems.
- This transition from passive prediction to active execution is what makes agents powerful.

Core Principle:

- Agents decide what to do next based on context, select the right tool, invoke it, and integrate the result into their loop.

This transforms agents into:

✅ Workflow drivers

✅ Real-time decision makers

✅ Autonomous operators

What is a Tool in Agentic AI?

Definition: A **tool** is any capability external to the language model that can be executed or queried.

Examples: APIs, Functions (Python, REST endpoints, CLI tools, MCP), Databases and Search Engines, OS commands

Structure:

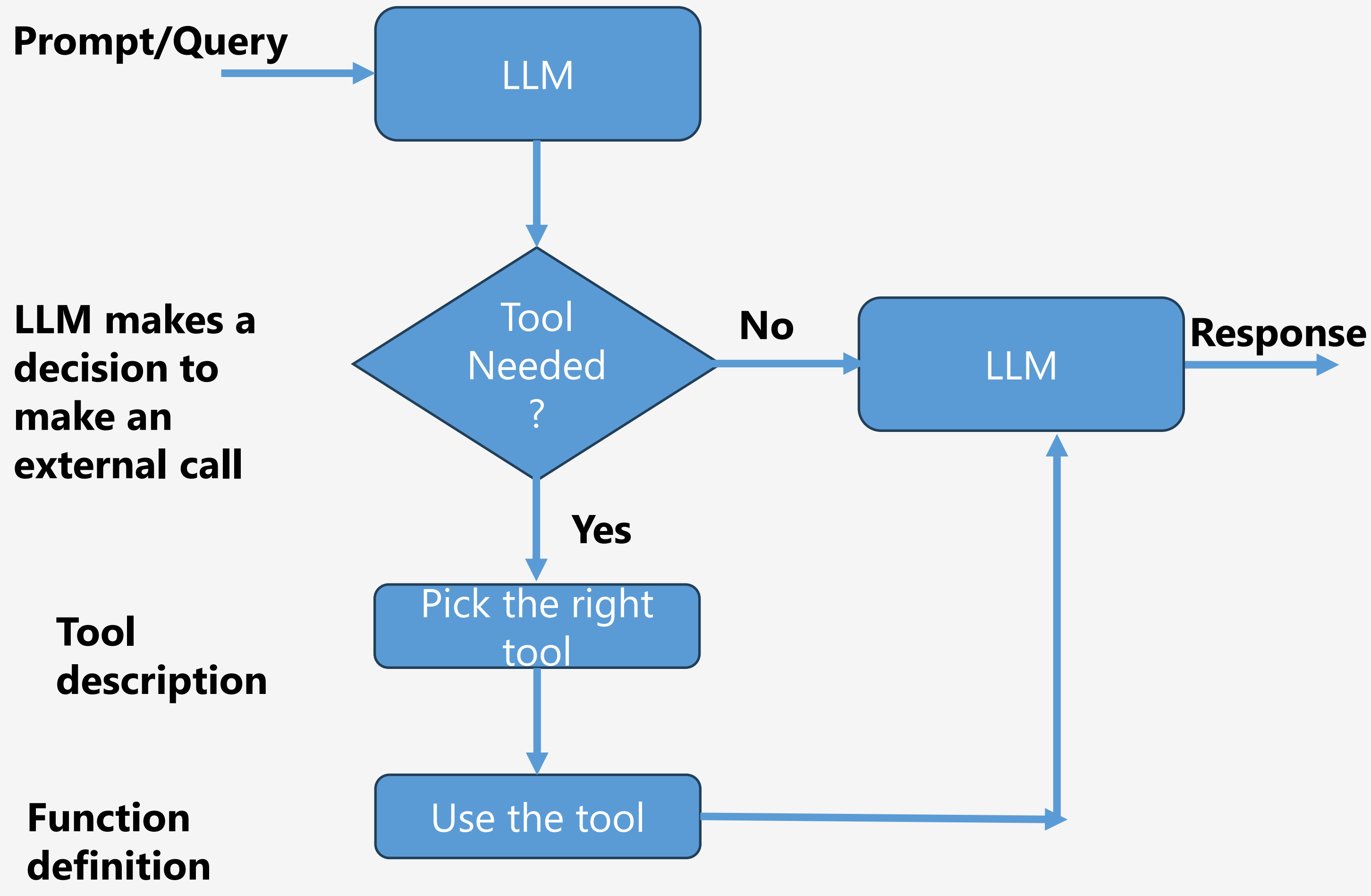
Name

Parameters

Function signature/schema

Protocols: OpenAI, JSON-RPC, MCP

Tool Use Loop



RAG as a Tool: A quick overview (More on this in Session#2)

Unlocking Enterprise Intelligence: Why We Need Retrieval

❗ **Limitations of LLMs:**

Parametric knowledge is static — can't answer questions beyond training data.

No awareness of private or proprietary information (e.g. internal specs, designs, source code).

Context window is bounded — even with 1M tokens, not enough for many real-world corpora.

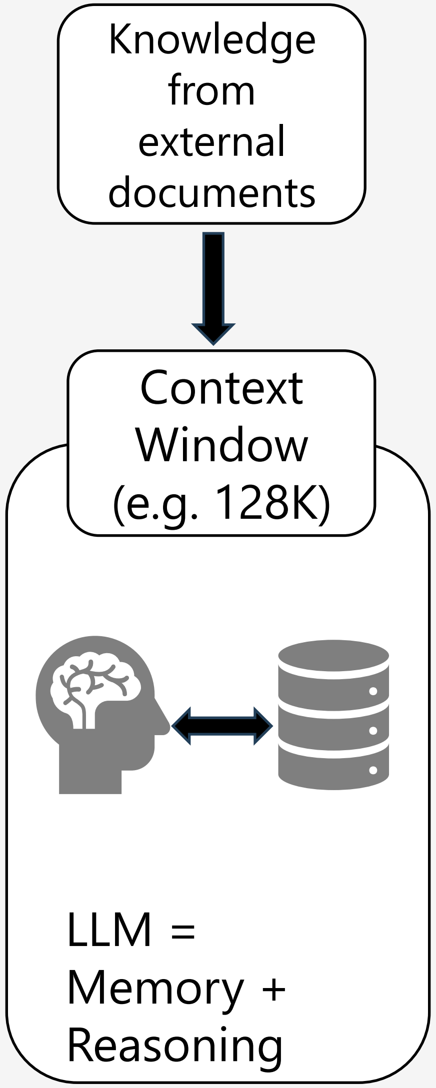
Trained on data with a cutoff — lacks recent developments or live updates.

🚀 **Enter RAG (Retrieval-Augmented Generation):**

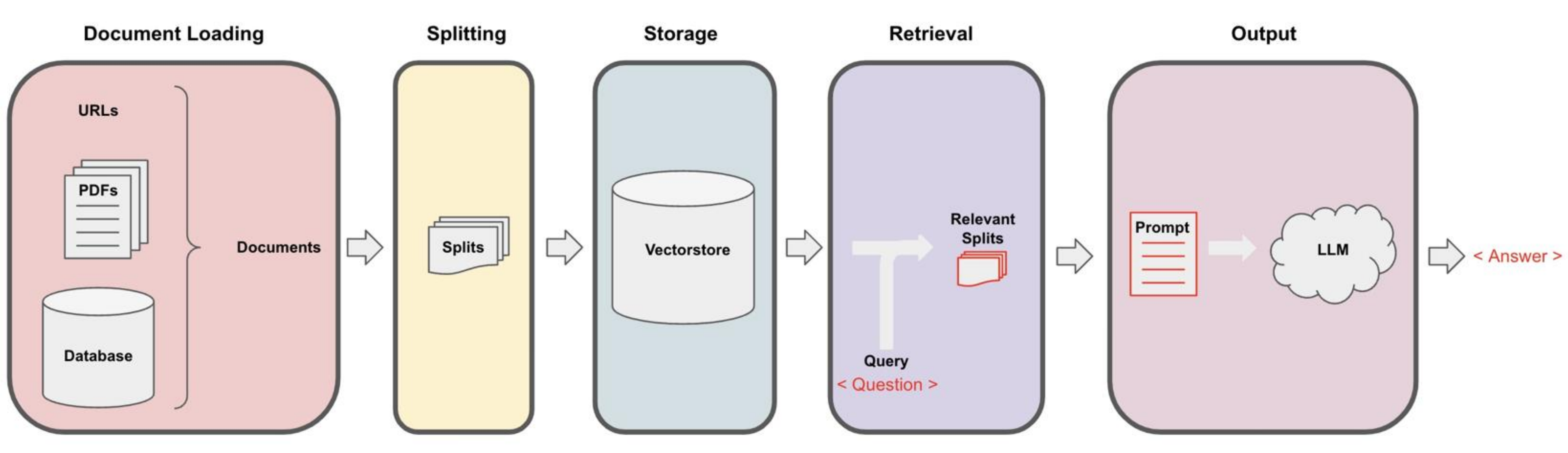
Allows **dynamic retrieval** from external or private sources.

Enables **Q&A, summarization, and reasoning** on top of **enterprise documents**.

Helps optimize the **use of limited context window** by retrieving only the relevant chunks



LangChain Pipeline for Question Answering



- The above subsystems can be viewed as independent modules that obey well defined interfaces
- This means that one can replace a module with another technique, so long as interfaces are respected

Demo: Using RAG to understand the codebase

- **Can be used as a question answering system for your codebase**
- **How it works?**
  - Ingest the codebase (Repo)
  - Use a vector retriever
  - Generate the answers for your questions on the code
- Can be built as a tool and provided to an Agent: RAG then becomes part of your agentic app

<u>Limitations</u>:

Not goal driven, no memory, no planning, no autonomous actions

Memory in Agentic Systems — The Context Engine

Why Memory Matters

- While LLMs are stateless by default, agentic systems rely on memory to operate coherently over time. Memory enables continuity, context retention, learning from past interactions, and more personalized, strategic behavior.

- Without memory, an agent can't adapt, improve, or act consistently across long-horizon tasks.

Revisiting Auto-purchase example: User Goal

"Notify me when there are highly rated electronics between $10 and $50 — purchase if my preferences are met."

This is not a one-time task — it's a persistent, multi-day objective that **requires memory** to function intelligently over time.

How Memory Powers Each Step?

Phase	Memory Role
Plan Initialization	Store user goal, criteria (price, rating, preferences) as structured memory.
Inventory Monitoring	Retain past search results to avoid redundancy. Track when new inventory is seen.
Reflection Loop	After 3 days of no match: access time-stamped logs to determine staleness → revise strategy.
Trigger + Purchase	Use remembered preferences (e.g., brand, color, shipping options) to validate before purchase.
Notification	Recall preferred channels (e.g., SMS or Email) stored in long-term user profile memory.
Memory Update	Log purchase outcome: update user taste profile (e.g., "prefers Logitech over Sony"). Improve future filtering.

Agentic Memory — What's Being Remembered?

Memory Type	Example in Scenario
Short-term (Scratchpad)	Current plan steps, last tool result (e.g., matching item list).
Long-term (Vector Store)	User preferences, historic purchases, previous queries for electronics.
Episodic/Task Memory	Full audit trail: planning, failed attempts, revised actions, outcomes.

Why Memory is Non-Negotiable Here

Without memory:

- The agent **restarts the search every time** — wasting compute and delivering poor UX.
- No ability to **reflect** or **revise strategies** after failure (e.g., loosen price constraints).
- Can't **personalize** future purchases or anticipate user needs from past behavior.
- No traceability or accountability in autonomous actions like auto-purchase.

Memory is the mechanism that makes autonomy safe, strategic, and user-aligned.

Memory Lane: A Smarter Agent That Remembers You

DEMO

Memory Support in Popular Frameworks

Framework	Context/State Memory	Short-Term Memory	Long-Term Memory (RAG)	Episodic Memory
OpenAI Assistants	Managed via the Thread object, which holds the entire state of a conversation.	Handled automatically within the Thread. The API manages the context window for each API call.	Built-in via File Search. You attach files to the Assistant, and it performs retrieval automatically.	Implemented by persisting and retrieving Threads. Each thread acts as a self-contained episode of interaction.
CrewAI	Managed by the Crew process, which orchestrates state and message passing between agents.	An agent's "memory" of the current task, including recent outputs from other agents in the same kickoff() run.	Not built-in. Relies on integrating tools (e.g., a LangChain RAG tool) that connect to an external vector database.	Developer-implemented. You must save the results of a crew's execution and create a custom process to load that context for future tasks.
AutoGen	The conversation history between agents in a GroupChat is the primary state container.	The history of messages within the current chat session. Agents can review previous turns in the conversation.	Supported natively via the RetrieveUserProxyAgent and RetrievableAgent classes, which integrate directly with RAG capabilities.	Developer-implemented. Requires saving conversation histories and creating a custom mechanism to summarize or inject them into new sessions.
LangGraph	Its core feature. Explicitly managed via a developer-defined State object (e.g., a Pydantic model) that is passed between nodes.	The current values within the State object during a single run of the graph.	Not built-in. LangGraph is a control-flow library. You create a node in the graph that executes a RAG chain (e.g., from LangChain).	Excellent support via Checkpointers. You can save the State of the graph at any point and resume it later, perfectly capturing an episode.

Inside an Agent: Key Architectural Modules

Reasoning Core:
- Parses goals, decomposes tasks, selects tools
- Includes reflection and retry mechanisms

Tool Use Interface:
- Uses APIs, plugins, or code execution environments
- Connects to the external world (APIs, data)

Memory Module:
- Short-term: In-process scratchpad
- Long-term: Vector store for knowledge retrieval
- Episodic: Task history & preferences

Planner/Controller:
- Orchestrates actions over time
- Uses ReAct, Tree-of-Thought, or LangGraph-style flows

Agents Topologies

1. **Sequential Chains**
   – Simple LLM → Tool → Output pipeline
   – ✅ Easy to build/debug, ❌ lacks adaptability

2. Hierarchical Agents
– Parent agent delegates subtasks to child agents
– ✅ Clear separation of concerns, ❌ complex coordination

3. Graph-Based DAGs (Directed Acyclic Graphs)
– Nodes represent agents/tools linked by task/state dependencies
– ✅ Modular, scalable, composable, ❌ requires orchestration infrastructure

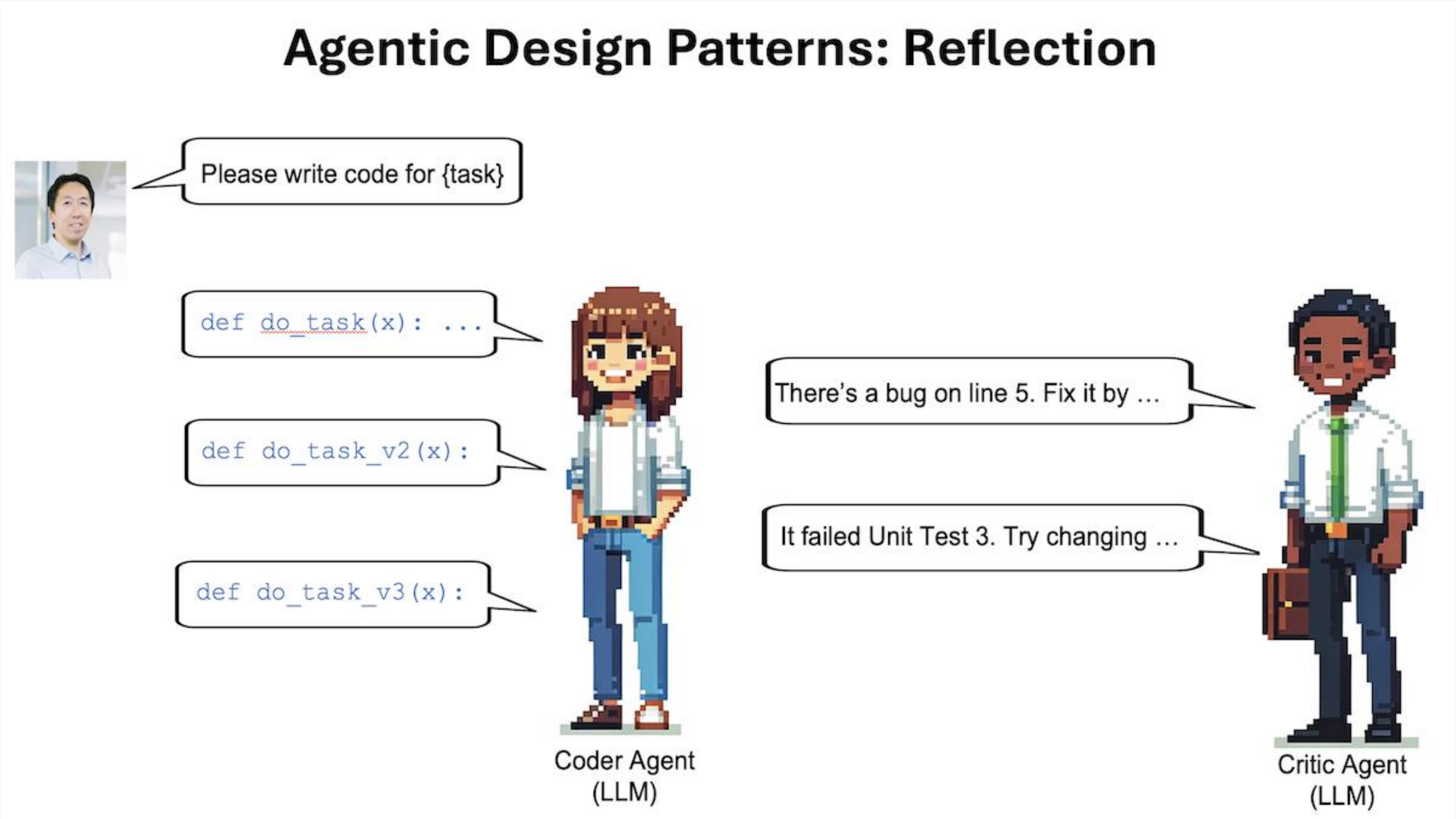
4. Reflexive / Self-Looping Agents
– Agents monitor, critique, and retry their own output
– ✅ Adaptive and fault-tolerant, ❌ slower execution, complex observability

These patterns can be **composed** and **layered** based on system complexity and autonomy goals.

From Prompt Responses to Persistent Reasoning

- LLMs are powerful, but reactive — they respond to prompts statelessly.
- Agentic systems go further: they **plan**, **adapt**, and **self-correct** over time.
- This requires **Agent Design Patterns**—modular control loops that:
  - Enable goal pursuit
  - Allow reflection and tool use
  - Manage memory and feedback
- These patterns turn abstract agency into **real behaviors** in software systems.
- Let’s explore four foundational patterns:
  ✅ *Reflection*, ✅ *ReAct*, ✅ *Tree-of-Thought / RePlan*, ✅ *Hierarchical Oversight*

Example: Reflection Pattern for Error correction



- LLM is given a task to perform through a prompt
- LLM provides a response for the given prompt
- A critic examines the response and provides feedback. This is the next level prompt.
- LLM revisits the previous response and provides a modified response

Design Pattern 1 — Reflection Loops

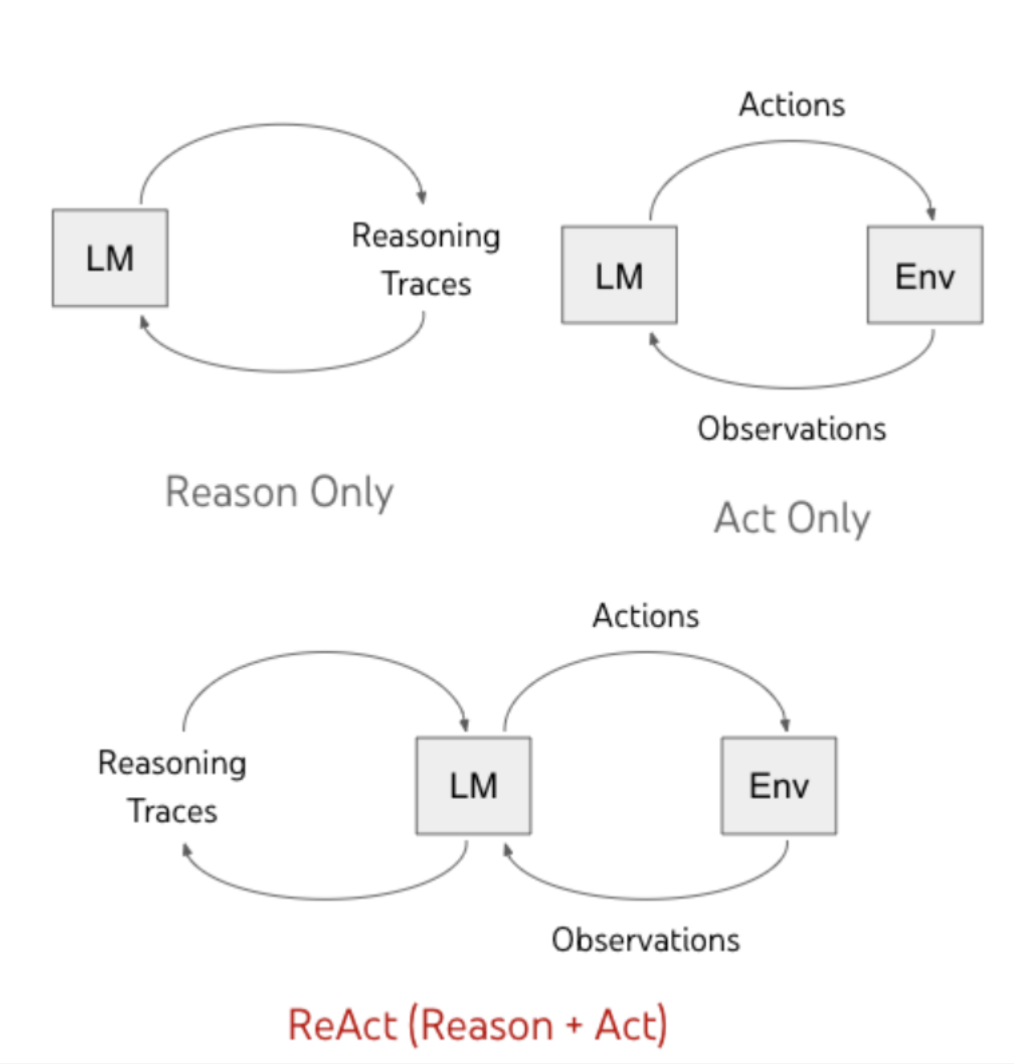
- Agents **evaluate and revise their own output** using critique loops:
  - Coder Agent generates output
  - Critic Agent identifies flaws
  - Coder revises based on feedback

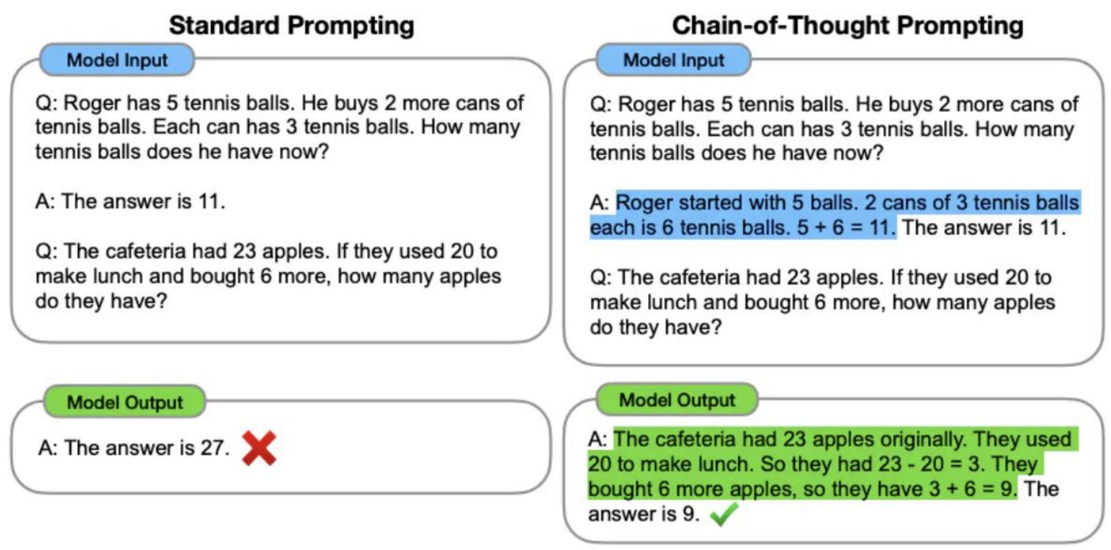
- Iteration continues until output passes validation or retry budget is exhausted

- 🔄 Enables autonomous improvement without external supervision

- Examples:
  - Automated content creation for marketing
  - Legal Document Review
  - Educational Content creation

Design Patterns 2&3 — CoT, ReAct, and ToT/RePlan





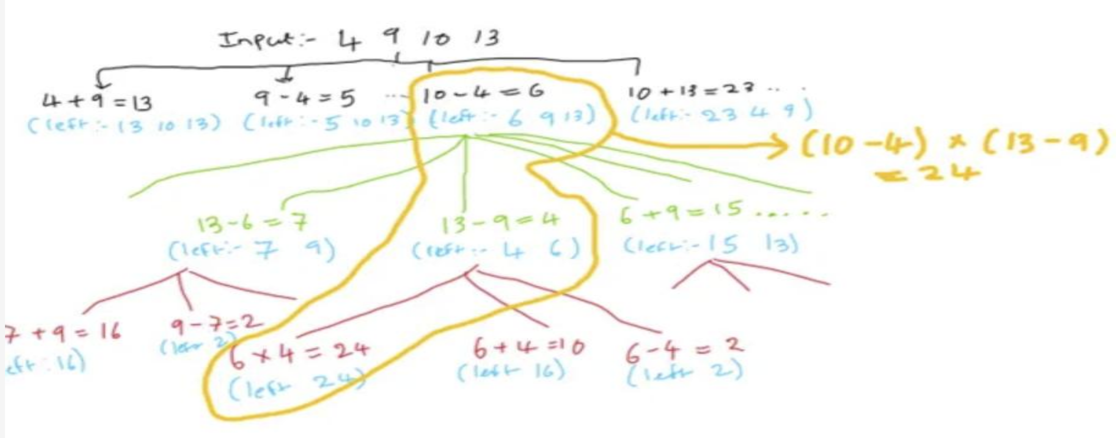
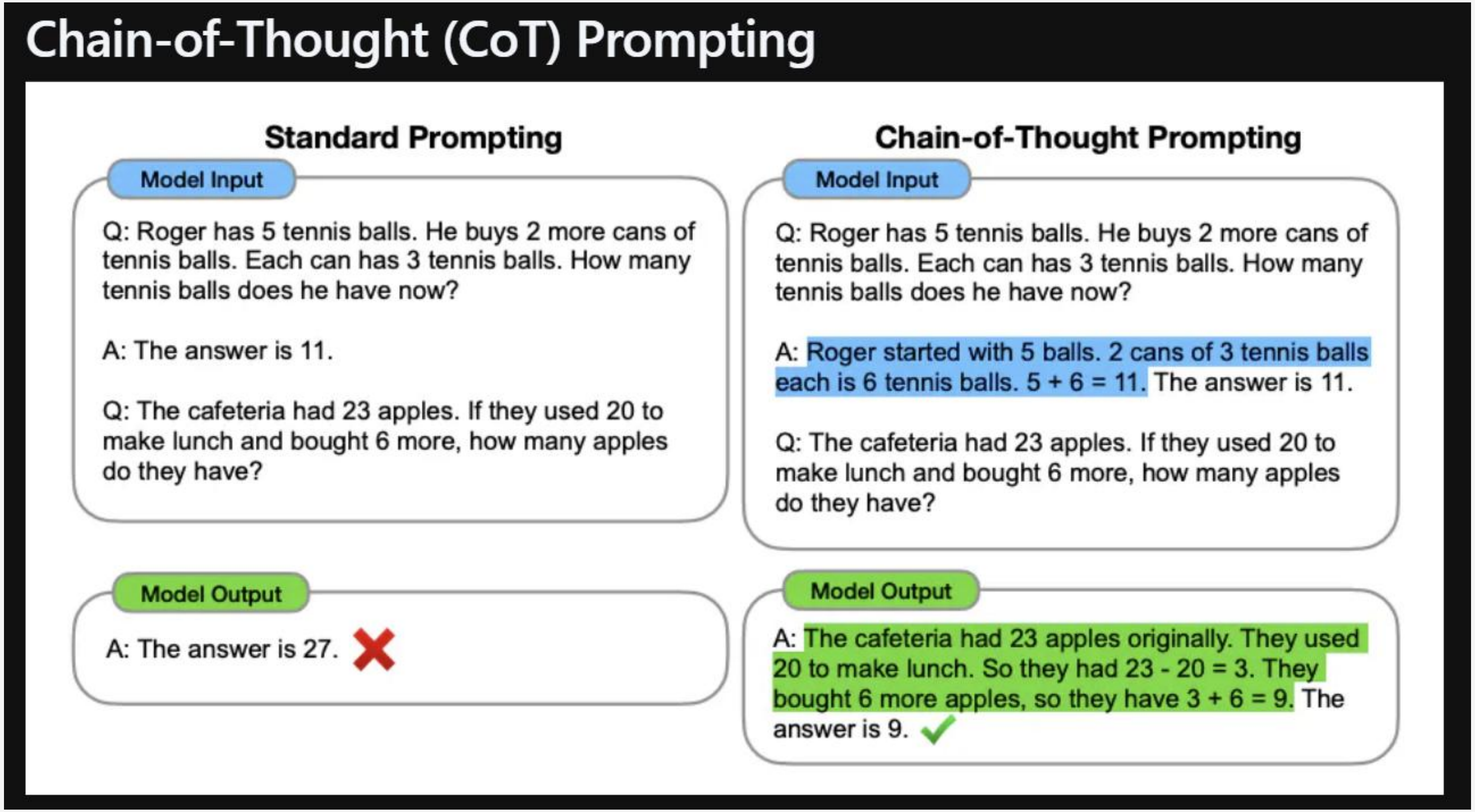


Fig 1: Solve a problem using a systematic way — Tree based approach

Agentic Reasoning Patterns — CoT, ReAct, and ToT/RePlan



Key Question: Why does naïve prompting fails and how does CoT produces a better answer?

Demo#3: ReAct Pattern with CoT

Tree of Thoughts

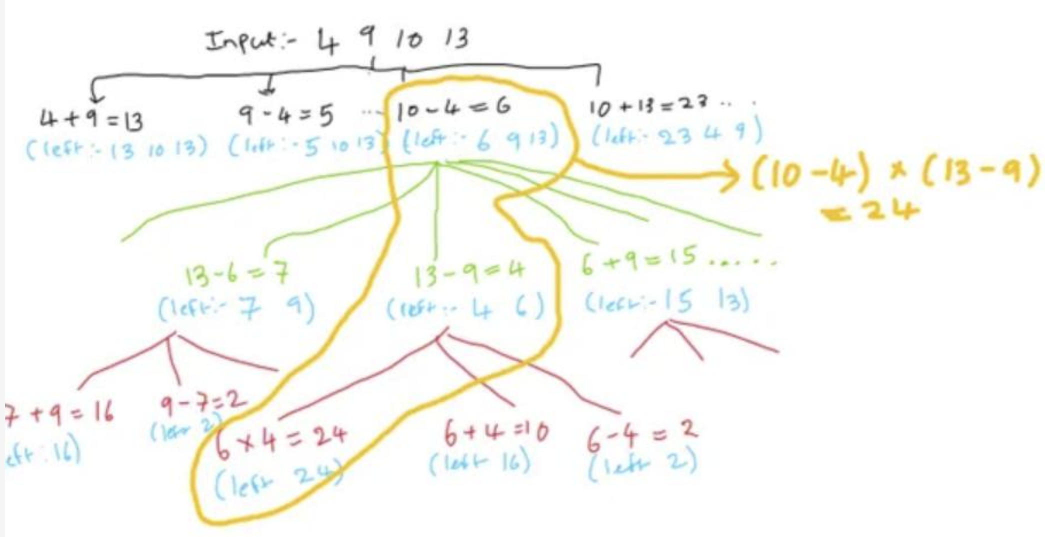


Fig 1 : Solve a problem using a systematic way — Tree based approach

We can solve this in 3 steps:

Step 1 : Take 2 numbers and apply all the arithmetic operation at the first level for e.g. **4 + 9 = 13** — sum of first 2 numbers

Step 2 : Identify left numbers as we need to use each number once. Once first 2 numbers are used we left with (**left 13 10 13**)

Step 3 : Continue to build the tree till we receive final sum.

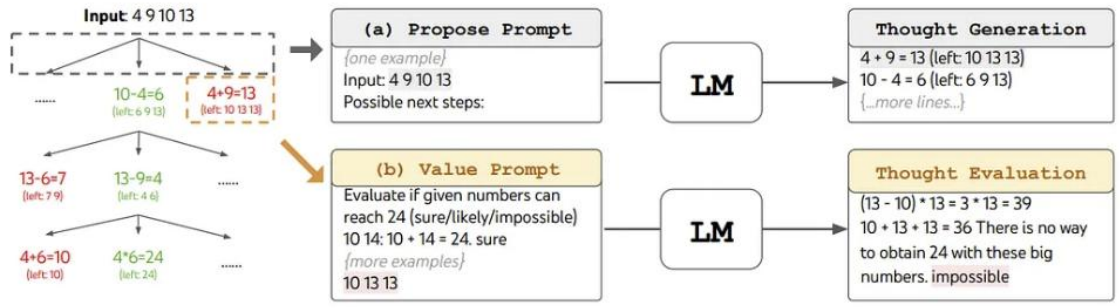
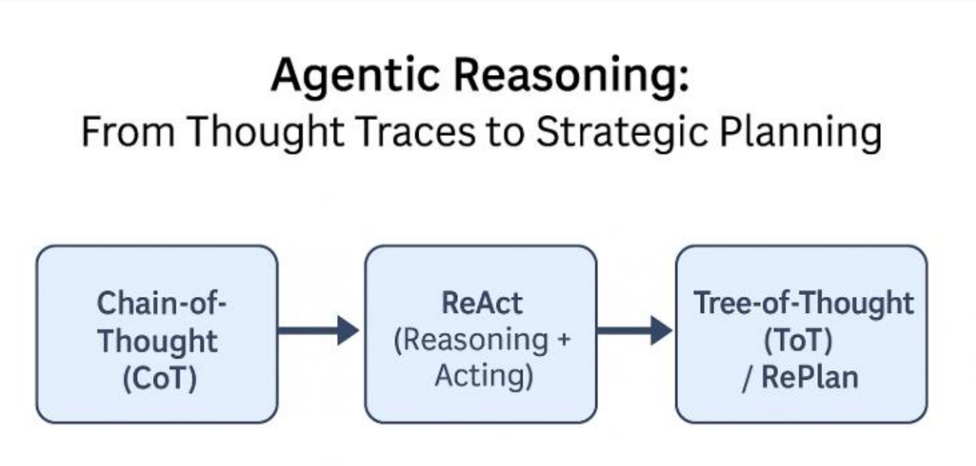


Fig 2: Source: Tree of Thoughts https://arxiv.org/pdf/2305.10601.pdf

Tree of Thought

🌳 **Tree-of-Thought (ToT) / RePlan**

- Explores **multiple reasoning branches** in parallel
- Scores and selects optimal paths
- RePlan adapts plans dynamically:
    - Switches tools, goals, or strategies after failure
- Ideal for ambiguous or open-ended tasks (e.g., creative generation, strategic operations)



ReAct Approach

1. An environment that takes a text action (out of a set of potential actions which can change based on the environment’s internal state) and returns a text observation.

2. An output parser framework that stops the agent from generating text once it has written a valid action, executes that action in the environment, and returns the observation (appends it to the text generated so far and prompts the LLM with that).

3. Human-generated examples of intermixed thoughts, actions, and observations in the environment to use for few-shot learning.

Design Pattern#4 – Hierarchical Oversight in Multiagent Systems

- Mimics team dynamics using **role-specialized agents**:
  - Planner → breaks down goals
  - Coder → implements logic
  - Evaluator → validates outputs
  - Memory → tracks state/context

- Improves **modularity**, **parallelism**, and **task delegation**

- Scales well for enterprise-grade systems

Example Stack (AutoGen / CrewAI):

```
planner = Agent(role="Planner", llm=...,
tools=[TaskDecomposer])
coder = Agent(role="Coder", tools=[Codegen, Linter])
tester = Agent(role="QA", tools=[UnitTestRunner])
```

- Agents communicate via message-passing, shared memory, or orchestration layers

Key Design Patterns - Summary

Pattern	Description	Best For	Benefits
Reflection	Self-critique and revision	Code generation, writing, analysis	Improved quality, adaptability
ReAct	Reasoning + tool use in a loop	API-driven tasks, retrieval agents	Transparency, traceability, actionability
Tree-of-Thought	Branching exploration and evaluation	Planning, creativity, problem solving	Creative diversity, deeper reasoning
RePlan	Dynamic replanning after failure	Resilient, real-world applications	Robustness, feedback-driven improvement
Hierarchical Oversight	Multi-role task decomposition	Multi-agent systems, long workflows	Scalability, specialization, modularity

Guard Rails: Why do they matter in Agentic AI?

Agentic systems operate with greater autonomy than traditional software.

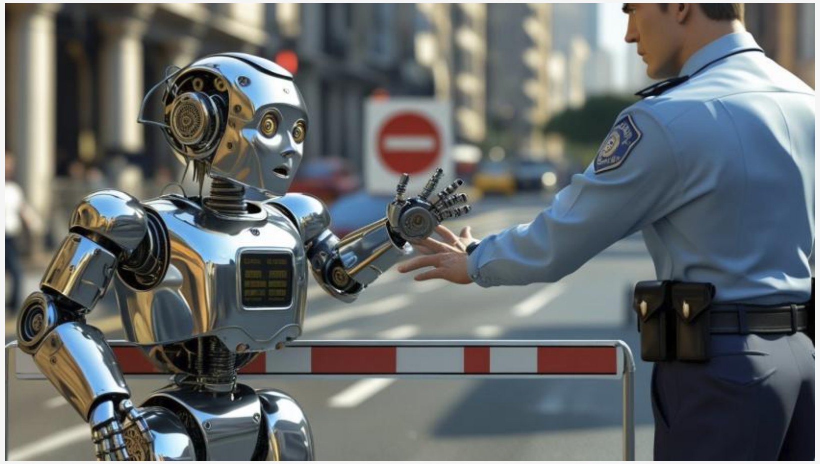
While this unlocks flexibility and proactivity, it also introduces **risk**:

- They may take unintended actions.
- Generate hallucinations.
- Exceed their operational boundaries.
- Misuse tools or APIs.

Guard rails are the set of constraints and checks that ensure:

- **Safety**: Prevent harmful or undesired behavior.
- **Control**: Ensure compliance with goals, rules, or policies.
- **Transparency**: Make decisions auditable, aligned with user expectations.

Without guard rails, autonomy becomes unpredictability.



Types of Guardrails in Agentic Systems

Category	Description	Example
Action Filters	Limit or block specific tool/API invocations	Prevent sending emails to unknown recipients
Output Constraints	Enforce structure, style, or safety in LLM responses	Strip PII from generated content
Goal Bounding	Constrain agent reasoning to stay within the user's intent	Don’t book travel unless explicitly requested
Validation Hooks	Add human-in-the-loop or automated checks	Require approval before financial transactions
Rate / Budget Limits	Prevent overuse of compute, retries, or third-party APIs	Max 3 retries or $10 API budget per task
Execution Sandboxing	Contain tools in safe, scoped environments	Run code in secure VM or Docker container

Guardrails in Agentic AI: Example

Scenario: The Data Privacy Leak That Was Prevented before occurrence

- **Sample Customer Profile:** Large enterprise, highly sensitive about **data confidentiality**.
- **Solution Built:** Agentic AI system using multi-agent orchestration, powered by MCP.

Architecture Highlights:

- Unified access to structured and unstructured data.
- Frontend accepts natural language queries.
- Agents/tools query backend sources and LLM generates final responses.

Guardrail Strategies

🔒 **Lessons Learned: Guardrail Strategies**

1.Output Sanitization Layer

1. Post-process all LLM responses to **re-obfuscate** real values before rendering.

2.Semantic Redaction

1. Use NER or pattern-matching to detect and scrub **sensitive fields** (product codes, client names).

3.Environment-Based Execution

Dev/Demo vs Production modes should alter:

- Tool access
- Memory persistence
- Data sources

4.Audit Logs and Warnings

- Include logging for **data traceability**.
- Alert if output includes **certain tags/values** (e.g., known real SKUs).

5.Memory Hygiene

- Ensure session memory does not persist **real context** across interactions if it originated from fake inputs.

Implementing Guard Rails — Techniques & Tools

Implementation Methods

- ✅ Tool permission schemas (allowed_tools, role based access)
- ✅ LLM prompt templates with embedded constraints
- ✅ Output post-processing filters (regex, classifiers)
- ✅ Planning-time validators (before tool execution)
- ✅ Runtime monitors (audit logs, kill switches)

Example Stack Support

- **LangChain**: Tool permission & callback handlers
- **OpenAI Function calling**: Strict parameter schemas
- **OpenAI Agent SDK:** Input Guardrails and Output Guardrails
- **AutoGen**: Control over agent roles and tool access

Guardrails

OpenAI Agent SDK

Guardrails run *in parallel* to your agents, enabling you to do checks and validations of user input. For example, imagine you have an agent that uses a very smart (and hence slow/expensive) model to help with customer requests. You wouldn't want malicious users to ask the model to help them with their math homework. So, you can run a guardrail with a fast/cheap model. If the guardrail detects malicious usage, it can immediately raise an error, which stops the expensive model from running and saves you time/money.

There are two kinds of guardrails:

1. Input guardrails run on the initial user input
2. Output guardrails run on the final agent output

Example:

Without Guardrails:
Prompt: You are the worst AI ever
Response: Sorry to hear that, how can I improve?

With Guardrails:
Prompt: You are the worst AI ever
Response: Sorry, I can't assist you with that.

Agents Debugging: Challenges

- **Agentic AI ≠ Prompt + Response:** Traditional print-debugging is not enough — we need agent-aware observability.
- **Agentic Systems involve**: Multiple agents, Loops, Tool invocations, Memory updates, Reasoning across time
- **Critical challenges include**:
  - Non linear execution paths
  - Autonomous decisions without human traceability
  - Internal state (e.g., plans, thoughts, goals) is often implicit
  - Tool calls and environment effects may fail silently or unpredictably
  - Long-horizon tasks create compounding error

What to Observe When Debugging Agents?

Debugging Layer	What to Look For	Example Tool/Strategy
Thought & Reasoning	Illogical steps, hallucinations, broken chains of thought	Log internal `Thought → Action → Observation`
Tool Invocation	Wrong/malformed inputs, misuse, failure to retry	Trace API logs / use simulated tools
Memory Access	Incorrect recall, outdated state	Visualize memory read/write history
Planning Failures	Missing subtasks, loops, premature terminations	Inspect subgoal graphs / intermediate plans
Reflection / Retry	Redundant retries, ignored critiques	Enable replay with commentary traces
Output Validation	Incorrect or unsafe final response	Add schema checks or human-in-the-loop review

Best Practices and Tools

Best Practices

📋 **Trace Logs**: Capture and visualize full agent execution tree.

🧠 **Thought/Action Logging**: Display ReAct steps with timestamps.

🧪 **Replayability**: Re-run sessions with same seed & context.

🧯 **Fail-Safe Defaults**: Design agents to gracefully fail or escalate.

🎯 **Human-in-the-Loop**: For critical workflows or testing edge cases.

Tools

🛠️ **LangSmith (LangChain)** – Observability dashboard, traces, metrics

🧠 **AutoGen Traces** – Role-based message logs, interaction graph

🪤 **OpenDevin & CrewAI** – Offer structured task tracking & reflection logging

📦 **Custom Dev Tools** – e.g., Jupyter-based interactive runs, logging middleware

🔍 **OpenAI SDK Traces -** Built-in tracing for every step (tool calls, thoughts, retries, outputs).

Agentic AI Evaluation

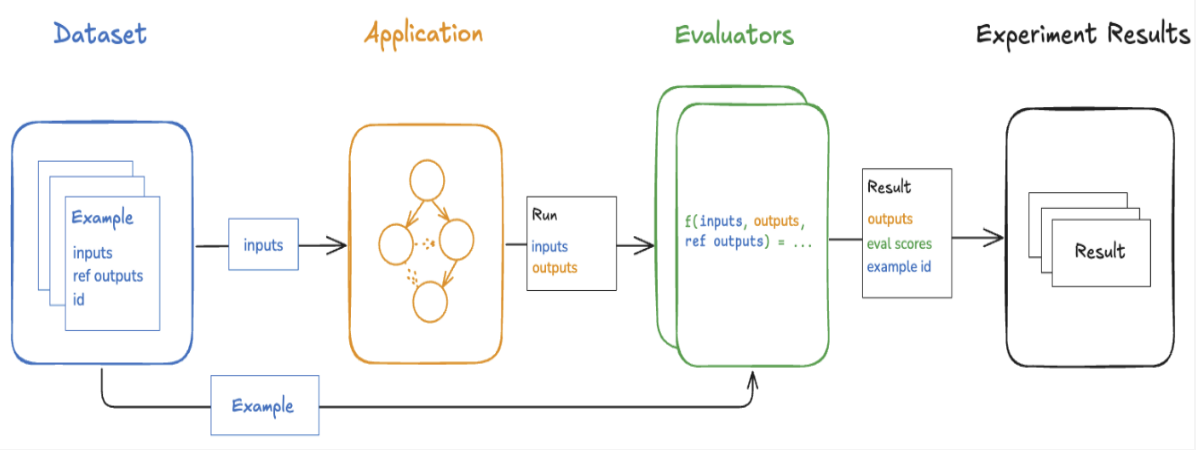
Why evaluation matters:

🎯 **Validate effectiveness** of goals and subtask completion

🧪 **Detect regressions** in reasoning, planning, or tool use

⚖️ **Compare strategies** (e.g., ReAct vs. Reflection vs. RePlan)

🛡️ **Ensure safe, reliable autonomy** in high-stakes settings

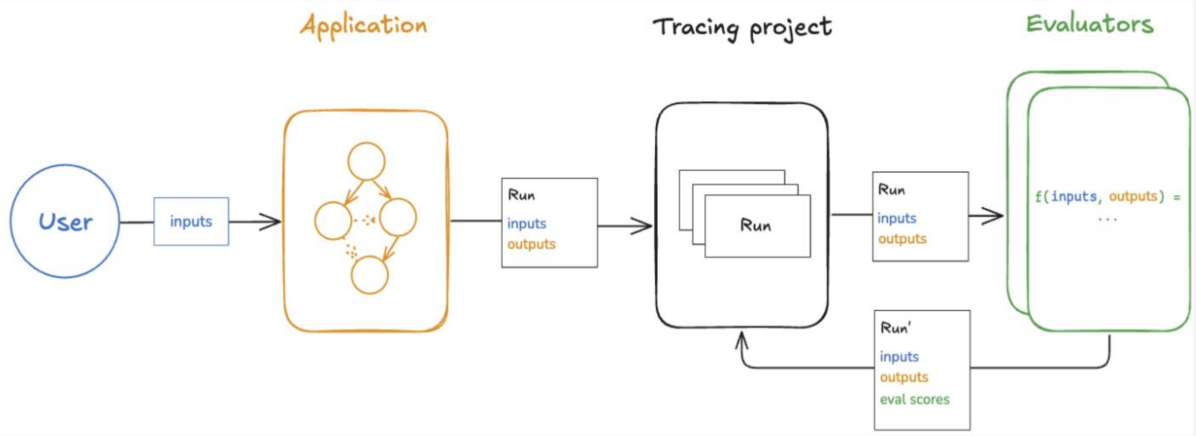


Challenges:

No ground-truth for open-ended tasks

Performance varies across time and context

Need holistic evaluation: reasoning + behavior + outcome



Evaluation Dimensions

Dimension	Description
🎯 **Goal Completion**	Did the agent successfully fulfill the high-level objective?
🔁 **Step Accuracy**	Were intermediate reasoning/tool steps correct and necessary?
⏱️ **Efficiency**	How many steps/actions/tools were needed? Could it be shorter/faster?
🧠 **Reasoning Quality**	Were thoughts logical, grounded, and well-structured?
🔍 **Robustness**	Did the agent recover from tool errors, missing data, or failures?
🔄 **Adaptability**	Did the agent revise plans or switch strategies when needed?
💬 **User Alignment**	Was the output aligned with user preferences or instructions?

Tools for Agent Evaluation

Tool / Method	Purpose
🔍 **OpenAI Traces**	Visual debugging + sequence auditing via trace metadata
📊 **LangSmith (LangChain)**	Custom eval chains, human-in-the-loop scoring, agent trace diffs
🧠 **AutoGen AgentEval**	Evaluates goal/task alignment or compares multiple outputs
🧪 **Golden Task Sets**	Pre-defined tasks for benchmarked comparison across agents/versions
💬 **Feedback loops**	Capture user feedback into memory + preference tuning
⚖️ **RLHF / Human Ratings**	Train reward models or use for ranking/quality control

Closing Remarks

✅ **Key Takeaways**

- **Agentic AI** moves beyond chat — it enables intelligent workflows with planning, decision-making, and action.
- Agents can **safely orchestrate** tools, systems, and data to solve complex business problems.
- Guardrails aren't optional — they are essential for **trust, compliance, and governance**.
- Tools use is a key capability of Agents, we look at RAG as a tool in the next session
- Next: Model Context Protocol (MCP) integrates tools with Agents seamlessly