# Agents and Agentic AI in the Era of Large Language Models

Anjan Goswami

April 16, 2025

## Introduction

The emergence of Large Language Models (LLMs) has brought forth a new paradigm in AI: **agentic AI**. Unlike traditional model deployments that operate in a single-shot or static context, agentic AI refers to systems that operate autonomously, perform multi-step reasoning, interact with tools and APIs, and self-reflect or iterate to improve performance. These AI agents are not just responders—they are actors, planners, and collaborators, capable of tackling complex, long-running tasks.

## 1 Defining Agentic AI with LLMs

In the context of LLMs, an agent is a system that can:

- **Understand and decompose** user goals into manageable sub-tasks.
- **Dynamically interact** with its environment using tools (e.g., APIs, databases, web browsers, code interpreters).
- **Maintain memory or state** across interactions to inform future actions.
- **Reason** about actions, outcomes, and potential errors over time.
- **Plan** sequences of actions to achieve a goal.
- **Learn or adapt** behavior based on feedback or self-evaluation.

Agentic behavior requires more than just language generation—it demands autonomy, persistence, contextual awareness, sophisticated planning, and decision-making capabilities.

## 2 LLM Capabilities that Enable Agents

Several core capabilities inherent or engineered into modern LLMs make them suitable foundations for building agents:

- **Subtasking and Decomposition:** LLMs can break down complex instructions into smaller, logical steps, often using reasoning techniques like Chain-of-Thought (CoT). This supports task planning and the creation of recursive or multi-step workflows.
- **Tool Use (Function Calling):** Through fine-tuning or dedicated APIs (like OpenAI's function calling, Anthropic's tool use, or Google's Gemini function calling), LLMs can select and invoke external tools. This allows agents to interact with the real world, access external knowledge, perform calculations, or execute code.
- **Stateful Reasoning and Memory:** Agents need to remember past interactions and current context to operate effectively over extended periods. Frameworks and platforms provide mechanisms for short-term memory (within a session) and long-term memory (persisted in external storage like vector databases or graph databases).
- **Reflection and Self-Improvement:** Some advanced agent designs allow models to evaluate their own outputs, reasoning steps, or tool usage outcomes, and then revise their plans or actions. This capability is crucial for improving performance autonomously.

## 3 Engineering Capabilities into Agents

Creating robust AI agents involves several engineering strategies:

- **Prompt Engineering:** Designing detailed system messages and instructions to guide the LLM's persona, goal decomposition logic, planning process, and tool selection criteria.
- **Function Calling / Tool Invocation:** Utilizing structured APIs provided by LLM vendors (e.g., OpenAI, Anthropic, Google) that allow the model to request the execution of predefined functions or tools based on the conversation context.
- **Orchestration Logic:** Wrapping LLM calls within programmatic structures (often using Python frameworks like LangChain, AutoGen, or CrewAI, or managed services like OpenAI Assistants API) to handle the agent loop (plan -> act -> observe -> repeat), manage memory, implement error handling, and coordinate multiple agents.
- **Memory Management:** Integrating mechanisms for storing and retrieving relevant information. This can range from simple conversation buffers to sophisticated Retrieval-Augmented Generation (RAG) systems using vector databases or knowledge graphs.
- **Fine-Tuning or RAG:** Enhancing the agent's performance in specific domains by providing domain-specific knowledge through Retrieval-Augmented Generation or lightweight model fine-tuning to improve decision-making and tool use accuracy.

## 4 What an Agent Platform Should Provide (And How It's Implemented)

A robust platform for building and deploying agents should ideally support:

1. **Agent Orchestration:** Composing and coordinating single or multiple agents, potentially with specialized roles (e.g., planner, executor, verifier).
   - *Implementation:* State machines, workflow engines (like Temporal), graph-based libraries (like LangGraph), or message-passing systems (like AutoGen).
2. **Memory and Context Tracking:** Dynamically storing and retrieving information relevant to the agent's task and history.
   - *Implementation:* Rolling buffers for short-term memory (STM). For long-term memory (LTM), techniques include vector databases (e.g., Pinecone, Chroma, Milvus) for semantic similarity search (often used in RAG), relational databases, key-value stores, or graph databases (like Neo4j, FalkorDB) to capture complex relationships between pieces of information (Episodic and Semantic Memory).
3. **Tool Integration Layer:** A standardized way to register external tools, describe their functionality (often via schemas) so the LLM can understand them, and securely invoke them.
   - *Implementation:* API wrappers, schema definitions (like OpenAPI/Swagger or JSON Schema), secure credential management.
4. **State Management and Error Handling:** Ensuring agents can track their progress, gracefully recover from errors (e.g., tool failures, invalid LLM responses), and potentially checkpoint their state.
   - *Implementation:* Database persistence for agent state, retry mechanisms with backoff, fallback tools or strategies, logging, and monitoring.
5. **Observability and Debugging:** Providing tools to inspect logs, trace agent execution paths (decision-making, tool calls), monitor performance, and debug issues.
   - *Implementation:* Integration with logging frameworks (e.g., OpenTelemetry), specialized tracing tools (like LangSmith), dashboards, and debug modes.

## 5 Building a Scalable Agent Platform

To move agent systems from prototypes to production-ready applications, consider:

- **Modular Design:** Structure agents as composable units (potentially microservices) with clear interfaces, facilitating independent development, testing, and updates.
- **Asynchronous Processing:** Utilize task queues (like Celery, RabbitMQ) and asynchronous execution models to handle multiple agent tasks concurrently and manage long-running processes efficiently.
- **Session Management:** Reliably track and persist agent session states, allowing recovery across restarts or failures.
- **Fault Tolerance and Recovery:** Implement patterns from distributed systems like retries, idempotency, fallbacks, and potentially consensus mechanisms (e.g., multiple agents voting on an action) for critical tasks.
- **Monitoring and Verification Loops:** Employ automated checks, validation agents, or human-in-the-loop steps to verify agent outputs before they impact users or critical systems.

This approach draws heavily from distributed computing and microservices architecture principles, treating agents as workers within a larger, orchestrated system.

## 6 Building an Agent: A Practical Example (Simple Research Agent)

Let's outline the conceptual steps to build a basic agent that researches a topic online using a framework like LangChain or the OpenAI Assistants API:

1. **Define the Goal:** The user asks the agent to "Research the latest developments in quantum computing."
2. **Choose LLM & Framework:** Select a capable LLM (e.g., GPT-4o, Claude 3.7 Sonnet, Gemini 1.5 Pro) and an agent framework/API (e.g., LangChain, OpenAI Assistants API).
3. **Define the Agent's Role & Instructions (Prompting):** Instruct the agent: "You are a research assistant. Your goal is to find the latest (within the last 6 months) significant developments in quantum computing. Break down the task, use search tools to find information, synthesize the findings, and present a concise summary."
4. **Define Tools:** Provide the agent access to a web search tool (e.g., via an API like SerpAPI or Tavily). Describe the tool so the LLM knows how to use it (e.g., `web_search(query: str) -> str`).
5. **Set Up Memory (Optional but Recommended):** Configure short-term memory to track the conversation flow and intermediate findings. For more complex tasks, long-term memory could store previous research sessions or user preferences.
6. **Run the Agent Loop:**
   - **Plan:** The LLM receives the goal and plans the first step: "Search for 'latest developments quantum computing 2025'."
   - **Act (Tool Use):** The agent framework identifies the need for the `web_search` tool and executes it with the query.
   - **Observe:** The agent receives the search results (e.g., snippets from recent articles).
   - **Reason & Plan Next Step:** The LLM processes the results, extracts key information, and plans the next step. This might involve refining the search ("quantum computing hardware breakthroughs 2025"), searching for specific papers, or deciding it has enough information to synthesize.

- **Repeat:** The loop continues (search, read, synthesize) until the agent determines the goal is met.
- **Final Output:** The agent generates a summary based on its findings and memory.

Using OpenAI's Assistants API abstracts some of this: you define the assistant's instructions, enable tools (like `file_search` or custom functions), upload files if needed, and then interact via threads, letting the API manage the state and tool calls behind the scenes.

## 7 Task Complexity and Workflow Design

The complexity of the task significantly influences agent design:

- **Simple Tasks:** Tasks requiring only a few steps or a single tool might be handled by a single agent with basic prompting and tool use (e.g., summarizing a provided text, simple Q&A with RAG).
- **Complex Tasks:** Tasks involving multiple stages, dependencies, diverse tools, long-running execution, or conditional logic often benefit from more sophisticated architectures:
  - **Multi-Agent Systems:** Breaking down the task and assigning specialized roles to different agents (e.g., a Planner Agent, a Research Agent, a Writer Agent, a Validator Agent). Frameworks like AutoGen and CrewAI are specifically designed for this.
  - **Graph-Based Workflows:** Representing the task flow as a graph (nodes and edges) rather than a simple linear sequence. This allows for branching, looping, parallel execution, and more complex state management. LangChain's **LangGraph** extension is built for this, enabling cyclical and stateful agent interactions.

## 8 Communication and Task Flow: Chains vs. Graphs

How agents (or steps within an agent) interact determines the workflow structure:

- **Linear Chains:** The simplest structure where the output of one step (or agent) directly feeds into the next. This is common in basic LangChain applications (e.g., `SequentialChain`). Suitable for well-defined, sequential processes.
  - *Communication:* Direct passing of output from one step/agent to the next.
- **Graphs (DAGs or Cyclic):** More complex tasks often require non-linear flows. A graph structure allows:
  - **Dependencies:** Task B starts only after Task A finishes.
  - **Parallelism:** Tasks C and D can run simultaneously.
  - **Conditional Logic:** Run Task E if Task B succeeds, otherwise run Task F.
  - **Cycles/Loops:** Revisit a previous step based on an evaluation (e.g., refine a plan based on execution results). Frameworks like LangGraph, AutoGen, and CrewAI enable building these graph-like structures.
  - *Communication:* Agents might broadcast messages, write to shared state/memory, or have explicit handoffs managed by the orchestration layer (e.g., AutoGen's conversation patterns, CrewAI's task delegation, LangGraph's state transitions).

## 9 LLMs Suitable for Building Agents

Choosing the right LLM is critical. Factors include reasoning ability, instruction following, tool use proficiency, context window size, and cost.

- **Closed-Source Options:**
  - **GPT-4 / GPT-4o (OpenAI):** Strong reasoning, reliable function calling, vision capabilities, large context windows. Widely used in agent frameworks.

  - **Claude 3 family (Anthropic) (especially 3.7 Sonnet & 3 Opus):** Excellent for complex reasoning, long context, strong safety alignment, and increasingly powerful tool use. Anthropic emphasizes collaborative potential.
  - **Gemini 1.5 / 2.0 / 2.5 Pro (Google):** Very large context windows, strong multimodal capabilities, native function calling, integrates with Google Cloud (Vertex AI).

- **Open-Source / Open-Weight Models:**
  - **Llama 3 (Meta):** High-quality base models, perform well with fine-tuning for agentic tasks like planning and tool use.
  - **Mixtral / Mistral Large (Mistral AI):** Strong performance, often optimized for instruction following and conversational abilities.
  - **Specialized Models:** Models fine-tuned specifically for agentic capabilities, planning, or tool use are emerging (though specific names change rapidly).

## 10 Popular Agentic AI Frameworks and Platforms

Several frameworks and platforms simplify agent development:

- **LangChain:** A versatile Python/JS library providing modules for chains, agents, tools, memory, and RAG. Highly flexible but can become complex ("LangChain spaghetti") for intricate agents without careful design. LangGraph is its extension for graph-based stateful workflows.
- **AutoGen (Microsoft Research):** Focuses on multi-agent conversations and collaboration. Agents fulfill roles and interact via messages. Powerful for complex task orchestration and research but can have a steeper learning curve.
- **CrewAI:** An open-source framework emphasizing collaborative multi-agent systems with defined roles, goals, and tasks. Aims for intuitive design of agent "crews." Built on LangChain.
- **OpenAI Assistants API:** A managed service providing persistent threads, built-in retrieval, code interpreter, and function calling. Simplifies state and tool management but offers less control than frameworks and involves platform lock-in. Includes specific API costs beyond model usage.
- **Google Vertex AI Agents:** Part of Google Cloud, allows building agents using Gemini models. Integrates deeply with GCP services. Can involve more direct coding compared to some frameworks, focusing on enterprise applications. Supports both single and multi-agent patterns.
- **Anthropic's Tool Use:** While not a full "platform" in the same vein as OpenAI Assistants, Anthropic provides robust tool-use capabilities within its Claude model APIs (Messages API), enabling developers to build agentic workflows. They also offer specific solutions like Claude Code (agentic coding assistant in preview).
- **Other Frameworks:** LlamaIndex (data-centric agents/RAG), Semantic Kernel (Microsoft, enterprise focus), DSPy (programmatic optimization of prompts/models).

## 11 Platform Comparison: Features, Trade-offs, and Cost

| **Feature** | **LangChain/ LangGraph** | **AutoGen (Microsoft)** | **CrewAI** | **OpenAI Assistants API** | **Google Vertex AI Agents** | **Anthropic Tool Use** |
|---|---|---|---|---|---|---|
| **Type** | Open-Source Framework | Open-Source Framework | Open-Source Framework | Managed API Service | Cloud Platform Service (GCP) | Model API Capability |
| **Core Concept** | Chains, Graphs, Modules | Multi-Agent Conversation | Role-Based Agent Crews | Assistants, Threads, Tools | Agents, Tools, GCP Integration | Tool Use via Model API |
| **Ease of Start** | Moderate (LangChain), High (LangGraph) | Moderate-High | Moderate | High | Moderate-High (requires GCP setup) | Moderate (requires user orchestration) |
| **Flexibility** | Very High | High | High | Moderate | High (within GCP ecosystem) | High (within model limits) |
| **Control** | Very High | High | High | Moderate | High | High |
| **Multi-Agent** | Yes (esp. LangGraph) | Yes (Core Focus) | Yes (Core Focus) | Possible via API calls, less native | Yes | Possible via user orchestration |
| **State/Memory** | User Implements (many options) | Built-in conversation memory | User Implements (via LangChain) | Managed by API | User Implements (GCP services) | User Implements |
| **Ecosystem** | Very Large | Growing (Microsoft backed) | Growing (Built on LangChain) | OpenAI Ecosystem | Google Cloud Ecosystem | Anthropic Ecosystem |
| **Cost Factors** | LLM API calls, Infrastructure (DBs) | LLM API calls (potentially many), Infra | LLM API calls, Infra | API Fees + LLM calls + Storage | GCP Service Costs + LLM calls | LLM API calls + Infra |
| **Trade-offs** | Complexity at scale, flexibility | Setup complexity, research focus | Newer, relies on LangChain core | Less control, vendor lock-in | GCP lock-in, potentially code-heavy | Requires building orchestration |

### Cost-Benefit Considerations

- **LLM Costs:** Different models have vastly different costs per token (input/output). High-capability models like GPT-4o, Claude 3 Opus, or Gemini Advanced are more expensive than smaller/faster models like Claude 3.7 Sonnet, GPT-4.1, Gemini Flash, or Mistral models. Agentic workflows, especially multi-agent ones, can involve many LLM calls, amplifying costs.
- **Platform Fees:** Managed services like OpenAI Assistants API have additional costs for state management (per thread) and potentially tool usage. Google Vertex AI incurs standard GCP service costs.
- **Infrastructure Costs:** Running vector databases, task queues, or hosting services adds to the operational cost.
- **Development Costs:** The complexity of the framework impacts development time and effort. Easier-to-use platforms might reduce initial build time but offer less long-term flexibility.
- **Value Proposition:** The cost must be weighed against the value the agent provides. Simple automation might warrant cheaper models and simpler frameworks, while complex, high-value tasks might justify more expensive models, sophisticated multi-agent platforms, and higher operational costs.

## 12 Conclusion

Agentic AI represents a significant leap forward, transforming LLMs from passive responders into autonomous, goal-oriented systems. By leveraging capabilities like planning, tool use, memory, and self-reflection, LLM-based agents can execute complex workflows across diverse domains, from automating business processes and customer support to aiding in scientific discovery and software development.

Building robust, scalable, and reliable agents requires combining LLM strengths with principles from software engineering, distributed systems, and even cognitive science. The choice of LLM, framework, or platform depends heavily on the task's complexity, required control, scalability needs, and cost considerations. With powerful models from OpenAI, Anthropic, Google, and the open-source community, coupled with increasingly sophisticated frameworks like LangChain, AutoGen, and CrewAI, and managed services like the OpenAI Assistants API, the development of capable AI agents is accelerating, paving the way for a future where AI doesn't just process information—it acts upon it.