# Preface

In our previous e-book, "Mastering RAG," our goal was clear: building enterprise-grade RAG systems, productionizing them, monitoring their performance, and improving them. At the core of it, we understood how RAG systems enhance an LLM's ability to work with specific knowledge by providing relevant context.

In this e-book, we're taking a step further and asking, "How do we use LLMs to accomplish end-to-end tasks?" This singular question opens up a door: AI agents. A RAG system helps an LLM provide accurate answers based on given context. An AI agent takes that answer and actually does something with it — makes decisions, executes tasks, or coordinates multiple steps to achieve a goal.

A RAG-enhanced LLM could help answer questions about policy details by pulling relevant information. But an AI agent could actually process the claim end-to-end by analyzing the documentation, checking policy compliance, calculating payments, and even coordinating with other systems or agents when needed.

The ideas behind agents has existed for years. It can be a software program or another computational entity that can accept input from its environment and take actions based on rules. With AI agents, you're getting what has never been there before: the ability to understand the context without predefined rules, the capacity to tune decisions based on context, and learning from every interaction. What you're getting is not just a bot working with a fixed set of rules but a system capable of making advanced decisions in real-time.

Companies have quickly adapted, adopted, and integrated AI agents into their workflows. Capgemini's research found that "10% of organizations already use AI agents, more than half plan to use them in 2025 and 82% plan to integrate them within the next three years."

This e-book aims to be your go-to guide for all things AI agents. If you're a leader looking to guide your company to build successful agentic applications, this e-book can serve as a great guide to get you started. We also explore approaches to measuring how well your AI agents perform, as well as common pitfalls you may encounter when designing, measuring, and improving them.

**The book is divided into five chapters:**

**Chapter 1** introduces AI agents, their optimal applications, and scenarios where they might be excessive. It covers various agent types and includes three real-world use cases to illustrate their potential.

**Chapter 2** details three frameworks—LangGraph, Autogen, and CrewAI—with evaluation criteria to help choose the best fit. It ends with case studies of companies using these frameworks for specific AI tasks.

**Chapter 3** explores the evaluation of an AI agent through a step-by-step example of a finance research agent.

**Chapter 4** explores how to measure agent performance across systems, task completion, quality control, and tool interaction, supported by five detailed use cases.

**Chapter 5** addresses why many AI agents fail and offers practical solutions for successful AI deployment.

We hope this book will be a great stepping stone in your journey to build trustworthy agentic systems.

**- Pratik Bhavsar**

# Contents

## Chapter 3:
## How to Evaluate Agents

44/61



## Chapter 4:
## Metrics for Evaluating AI Agents

62/79

## Chapter 5:
## Why Most AI Agents Fail & How to Fix Them

```
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
from langgraph.prebuilt import create_react_agent

system_prompt = "You are a helpful finance expert named Fred in year 2024. First of all you
create a plan to get answer to the research query. Then you use tools to get answers to the
questions. Finally you use the answers to each question in the plan to give your final
verdict."

llm = ChatOpenAI(model="gpt-4o-mini")
tools = [TavilySearchResults(max_results=3)]
agent_executor = create_react_agent(llm, tools, state_modifier=system_prompt)
```

# 01

# CHAPTER

# WHAT ARE AI AGENTS?

# What are AI agents?

Let's start by understanding what AI agents are and which tasks you should use them for to maximize their potential.

AI agents are software applications that use large language models (LLMs) to autonomously perform specific tasks, ranging from answering research questions to handling backend services. They're incredibly useful for tasks that demand complex decision-making, autonomy, and adaptability. You might find them especially helpful in dynamic environments where the workflow involves multiple steps or interactions that could benefit from automation.

Salesforce estimates that salespersons spend 71% of their time on non-selling tasks (like administrative tasks and manually entering data). Imagine the time that could have gone into directly engaging with customers, developing deeper relationships, and ultimately closing more sales. This is true across multiple domains and applications: finance, health care, tech, marketing, sales, and more.

Let's use an example to understand this better. Imagine you run an online retail business and receive hundreds of customer inquiries every day about order statuses, product details, and shipping information. Instead of answering each and every query yourself, you can integrate an AI agent into your solution to handle these queries.

Here's how it would typically work:

1. **Customer Interaction**
A customer messages your service asking, "When will my order ship?"

2. **Data Retrieval**
The AI agent accesses the order management system to find the specific order details.

3. **Response Generation**
Based on the data retrieved, the agent automatically provides an updates to the customer, such as sending "Your order will ship tomorrow and you'll receive a tracking link via email once it's on its way."

The return to having an AI agent is multifold here:

- Super quick response time that keeps your customers happy
- Frees up your human staff to handle more complex queries and issues
- Improves your overall productivity and efficiency

**Fig 1.1** is an example of how agents are leveraged for code generation.

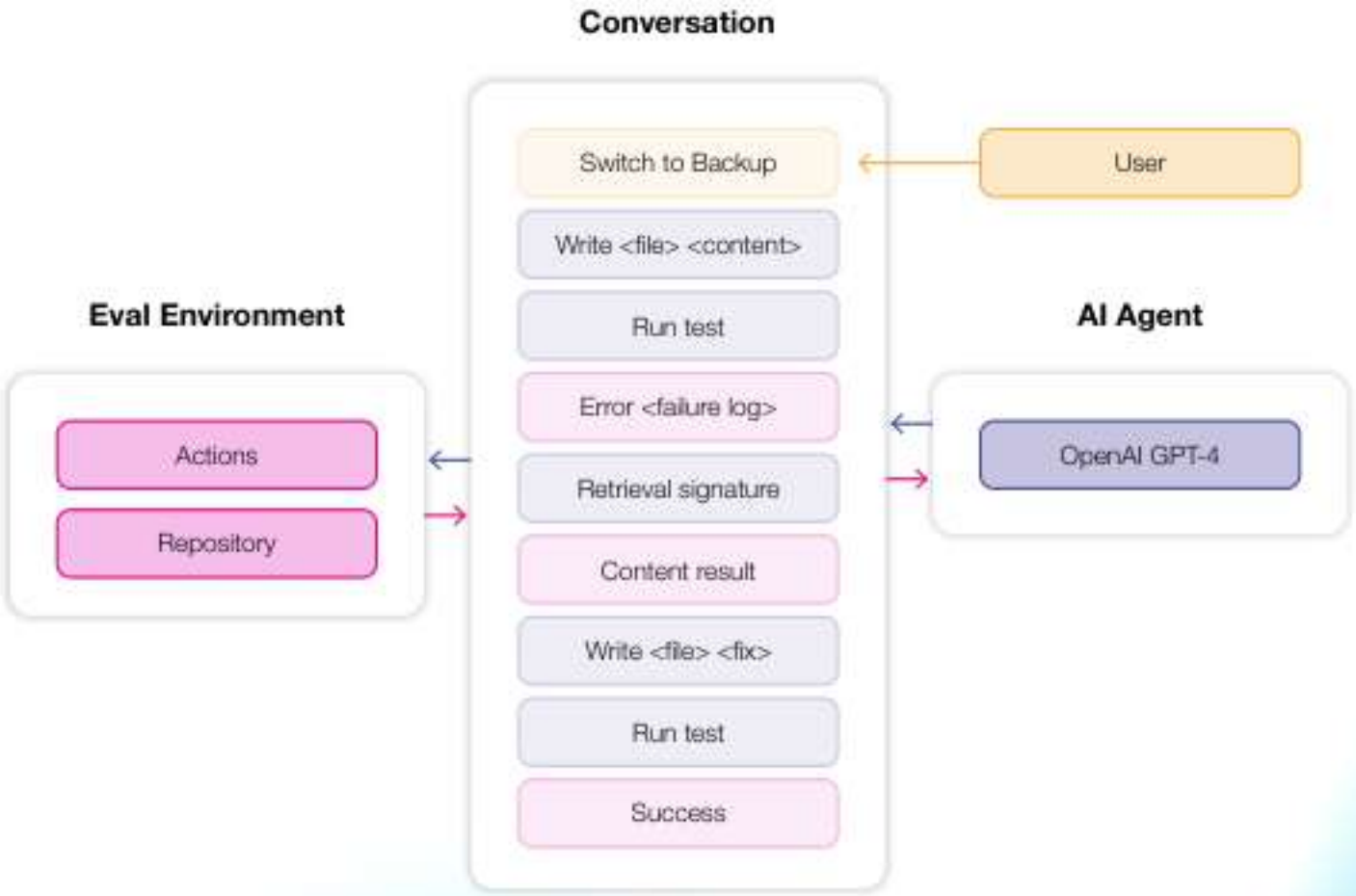


**Fig 1.1:** Automated AI-Driven Development using AI agents

# Types of AI Agents

Now that we're familiar with what AI agents are, let's look at different types of AI agents along with their characteristics, examples, and when you can use them. See **Table 1.1** below to get a quick idea of the types of AI agents and where and when you can use them.

| Name of the agent | Key Characteristics | Examples | Best For |
|---|---|---|---|
| Fixed Automation: The Digital Assembly Line | No intelligence, predictable behavior, limited scope | RPA, email autoresponders, basic scripts | Repetitive tasks, structured data, no need for adaptability |
| LLM-Enhanced: Smarter, but Not Einstein | Context-aware, rule-constrained, stateless | Email filters, content moderation, support ticket routing | Flexible tasks, high-volume/low-stakes, cost-sensitive scenarios |
| ReAct: Reasoning Meets Action | Multi-step workflows, dynamic planning, basic problem-solving | Travel planners, AI dungeon masters, project planning tools | Strategic planning, multi-stage queries, dynamic adjustments |
| ReAct + RAG: Grounded Intelligence | External knowledge access, low hallucinations, real-time data | Legal research tools, medical assistants, technical support | High-stakes decisions, domain-specific tasks, real-time knowledge needs |
| Tool-Enhanced: The Multi-Taskers | Multi-tool integration, dynamic execution, high automation | Code generation tools, data analysis bots | Complex workflows requiring multiple tools and APIs |
| Self-Reflecting: The Philosophers | Meta-cognition, explainability, self-improvement | Self-evaluating systems, QA agents | Tasks requiring accountability and improvement |
| Memory-Enhanced: The Personalized Powerhouses | Long-term memory, personalization, adaptive learning | Project management AI, personalized assistants | Individualized experiences, long-term interactions |
| Environment Controllers: The World Shapers | Active environment control, autonomous operation, feedback-driven | AutoGPT, adaptive robotics, smart cities | System control, IoT integration, autonomous operations |
| Self-Learning: The Evolutionaries | Autonomous learning, adaptive/scalable, evolutionary behavior | Neural networks, swarm AI, financial prediction models | Cutting-edge research, autonomous learning systems |

**Table 1.1:** Types of agents and their characteristics

# Fixed Automation – The Digital Assembly Line

This level of AI agents represents the simplest and most rigid form of automation. These agents don't adapt or think—they just execute pre-programmed instructions. They are like assembly-line workers in a digital factory: efficient but inflexible. Great for repetitive tasks, but throw them a curveball, and they'll freeze faster than Internet Explorer. (See **Table 1.2** below)

| Feature | Description |
|---|---|
| Intelligence | No learning, adaptation, or memory. |
| Behavior | Predictable and consistent, follows pre-defined rules. |
| Scope | Limited to repetitive, well-defined tasks. Struggles with unexpected scenarios. |
| Best Use Cases | Routine tasks, structured data, situations with minimal need for adaptability. |
| Examples | RPA for invoice processing, email autoresponders, basic scripting tools (Bash, PowerShell). |

**Table 1.2:** Characteristics of a fixed automation agent

The fixed automation workflow (See **Fig 1.2**) follows a simple, linear path. It begins when a specific input (like a file or data) triggers the system, which consults its predefined rulebook to determine what to do. Based on these rules, it executes the required action and finally sends out the result or output. Think of it as a digital assembly line where each step must be completed in exact order, without deviation.

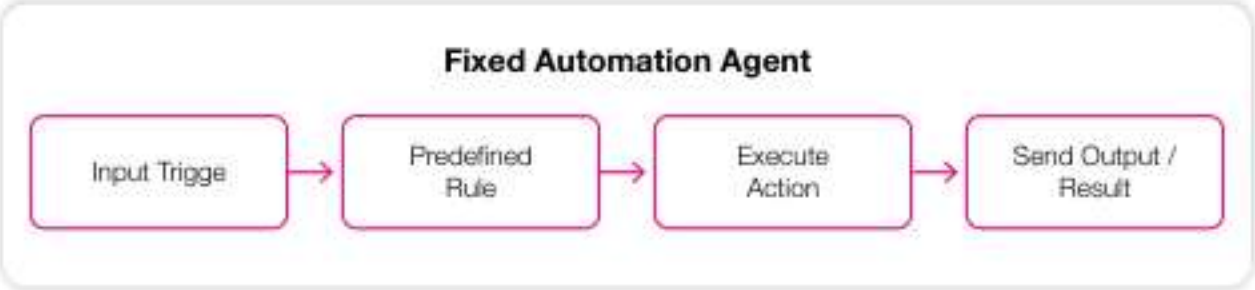


**Fig 1.2:** Workflow of a fixed automation agent

# LLM-Enhanced – Smarter, but Not Exactly Einstein

These agents leverage LLMs to provide contextual understanding and handle ambiguous tasks while operating within strict boundaries. LLM-Enhanced Agents balance intelligence and simplicity, making them highly efficient for low-complexity, high-volume tasks. Take a look at their features below in **Table 1.3**.

| Feature | Description |
|---|---|
| Intelligence | Context-aware; leverages LLMs to process ambiguous inputs with contextual reasoning. |
| Behavior | Rule-constrained; decisions are validated against predefined rules or thresholds. |
| Scope | Stateless; no long-term memory; each task is processed independently. |
| Best Use Cases | Tasks requiring flexibility with ambiguous inputs, high-volume/low-stakes scenarios, and cost-sensitive situations where "close enough" is sufficient. |
| Examples | Email filters, AI-enhanced content moderation, customer support classification. |

**Table 1.3:** Characteristics of an LLM-enhanced agent

The workflow below **(Fig 1.3**) shows how these smarter agents process information: starting with the input, the agent uses LLM capabilities to analyze and understand the input context. This analysis then passes through rule-based constraints that keep the agent within defined boundaries, producing an appropriate output. It's like having a smart assistant who understands context but still follows company policy before making decisions.

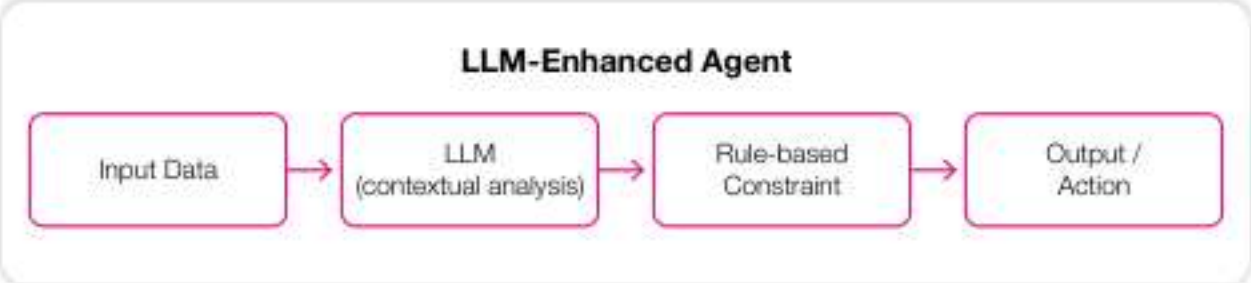


**Fig 1.3:** Workflow of a LLM-enhanced agent

# ReAct – Reasoning Meets Action

ReAct agents combine Reasoning and Action to perform tasks that involve strategic thinking and multi-step decision-making. They break complex tasks into manageable steps, reasoning through problems dynamically and acting based on their analysis. These agents are like your type-A friend who plans their weekend down to the minute. **Table 1.4** lists their characteristics.

| Feature | Description |
|---|---|
| Intelligence | Reasoning and action; mimics human problem-solving by thinking through a problem and executing the next step. |
| Behavior | Handles multi-step workflows, breaking them down into smaller, actionable parts. Dynamically adjusts strategy based on new data. |
| Scope | Assists with basic open-ended problem-solving, even without a direct solution path. |
| Best Use Cases | Strategic planning, multi-stage queries, tasks requiring dynamic adjustments, and re-strategizing. |
| Examples | Language agents solving multi-step queries, AI Dungeon Masters, project planning tools. |

**Table 1.4:** Characteristics of a fixed ReAct agent

The ReAct workflow starts with an Input Query and then enters a dynamic cycle between the Reasoning and Action Phase, as you'll see in **Fig 1.4**. Unlike simpler agents, it can loop between thinking and acting repeatedly until the desired outcome is achieved before producing the final Output/Action. Think of it as a problem solver that keeps adjusting its approach - analyzing, trying something, checking if it worked, and trying again if needed.

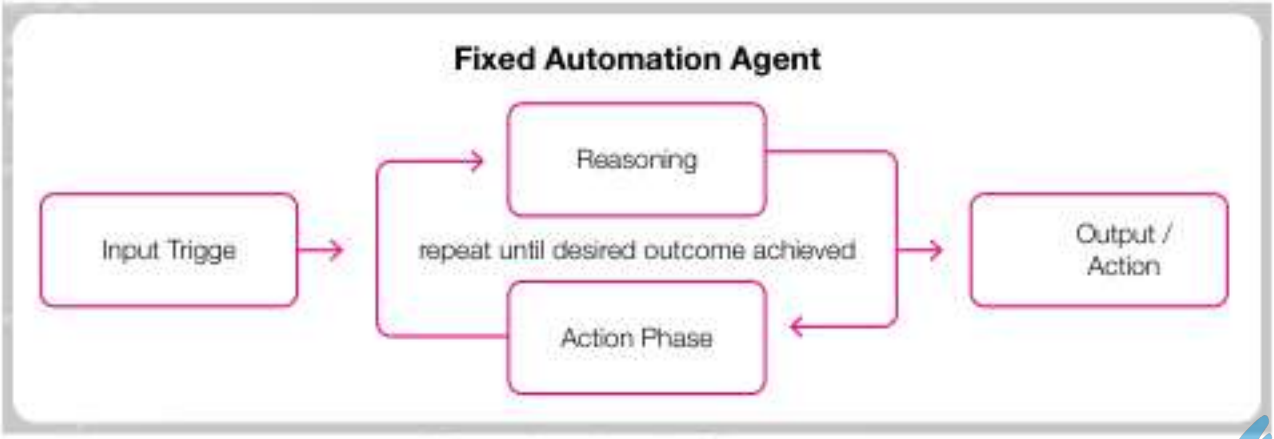


**Fig 1.4:** Workflow of a ReAct agent

# ReAct + RAG – Grounded Intelligence

Now, moving on to agents who are much more intelligent, we come to ReAct + RAG agents that combine reasoning, action, and real-time access to external knowledge sources. This integration allows them to make informed decisions grounded in accurate, domain-specific data, making them ideal for high-stakes or precision-critical tasks (especially when you add evaluations). These agents are your ultimate trivia masters with Google on speed dial. See **Table 1.5** to learn how this agent works.

| Feature | Description |
|---|---|
| Intelligence | Employs a RAG workflow, combining LLMs with external knowledge sources (databases, APIs, documentation) for enhanced context and accuracy. |
| Behavior | Uses ReAct-style reasoning to break down tasks, dynamically retrieving information as needed. Grounded in real-time or domain-specific knowledge. |
| Scope | Designed for scenarios requiring high accuracy and relevance, minimizing hallucinations. |
| Best Use Cases | High-stakes decision-making, domain-specific applications, tasks with dynamic knowledge needs (e.g., real-time updates). |
| Examples | Legal research tools, medical assistants referencing clinical studies, technical troubleshooting agents. |

**Table 1.5: Characteristics of a ReAct + RAG agent**

Starting with an Input Query, this advanced workflow combines ReAct's reasoning-action loop with an additional Knowledge Retrieval step. The agent cycles between Reasoning, Action Phase, and Knowledge Retrieval (See **Fig 1.5**) — consulting external sources as needed — until it reaches the desired outcome and produces an Output/Action. It's like having a problem solver who not only thinks and acts but also fact-checks against reliable sources along the way.

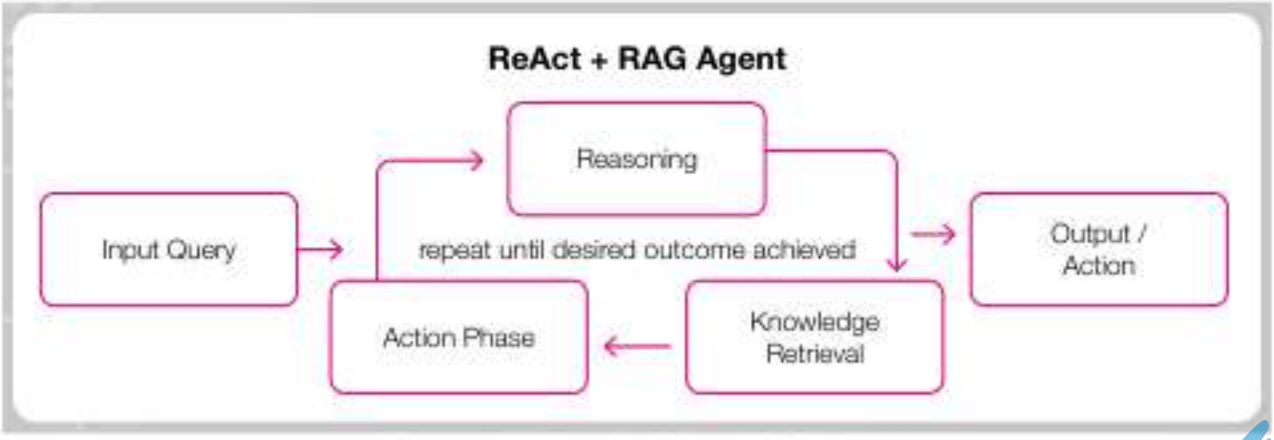


**Fig 1.5:** Workflow of a ReAct + RAG agent

# Tool-Enhanced – The Multi-Taskers

Tool-enhanced agents are versatile problem solvers that integrate multiple tools, leveraging APIs, databases, and software to handle complex, multi-domain workflows. They combine reasoning, retrieval, and execution for seamless, dynamic task completion. Think of them as tech-savvy Swiss Army knives capable of combining reasoning, retrieval, and execution seamlessly! (See **Table 1.6**)

| Feature | Description |
|---|---|
| Intelligence | Leverages APIs, databases, and software tools to perform tasks, acting as a multi-tool integrator. |
| Behavior | Handles multi-step workflows, dynamically switching between tools based on task requirements. |
| Scope | Automates repetitive or multi-stage processes by integrating and utilizing diverse tools. |
| Best Use Cases | Jobs requiring diverse tools and APIs in tandem for complex or multi-stage automation. |
| Examples | Code generation tools (GitHub CoPilot, Sourcegraph's Cody, Warp Terminal), data analysis bots combining multiple APIs. |

**Table 1.6:** Characteristics of tool-enhanced agents

Starting with an Input Query, the agent combines reasoning with a specialized tool loop. After the initial reasoning phase, it selects the appropriate tool for the task (Tool Selection) and then executes it (Tool Execution). This cycle repeats until the desired outcome is achieved, leading to the final Output/Action. (See **Fig 1.6**)

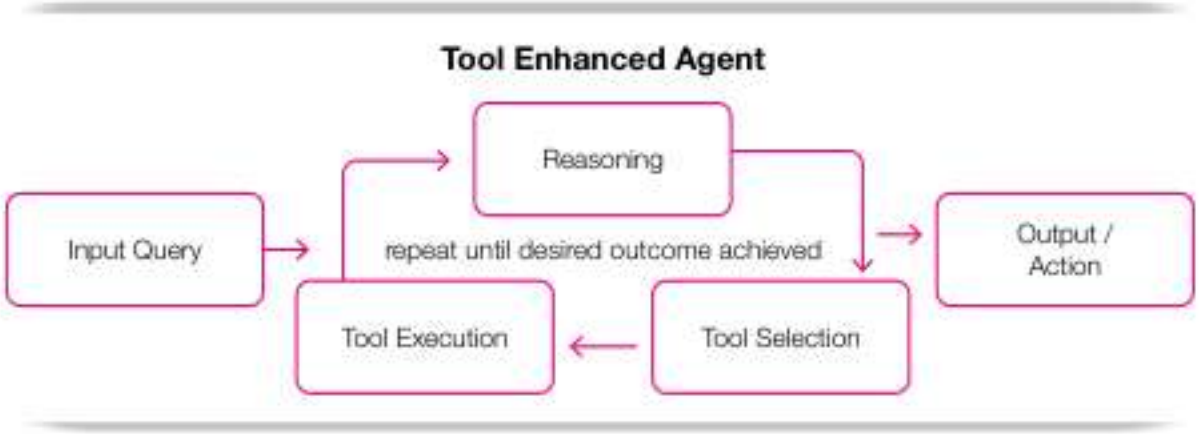


**Fig 1.6:** Workflow of tool-enhanced agents

# Self-Reflecting – The Philosophers

These agents think about their thinking. Self-reflecting agents introduce meta-cognition—they analyze their reasoning, assess their decisions, and learn from mistakes. This enables them to solve tasks, explain their reasoning, and improve over time, ensuring greater reliability and accountability. (See **Table 1.7**)

| Feature | Description |
|---|---|
| Intelligence | Exhibits meta-cognition, evaluating its own thought processes and decision outcomes. |
| Behavior | Provides explanations for actions, offering transparency into its reasoning. Learns from mistakes and improves performance over time. |
| Scope | Suited for tasks requiring accountability and continuous improvement. |
| Best Use Cases | Quality assurance, sensitive decision-making where explainability and self-improvement are crucial. |
| Examples | AI that explains its reasoning, self-evaluating learning systems, quality assurance (QA) agents. |

**Table 1.7:** Characteristics of self-reflecting agents

Starting with an Input Query, the agent goes through a cycle of Reasoning and Execution, but with a crucial additional step: Reflection. After each execution, it reflects on its performance and feeds those insights back into its reasoning process. This continuous loop of thinking, doing, and learning continues until the desired outcome is achieved, producing the final Output/Action. This is evident in **Fig 1.7**.

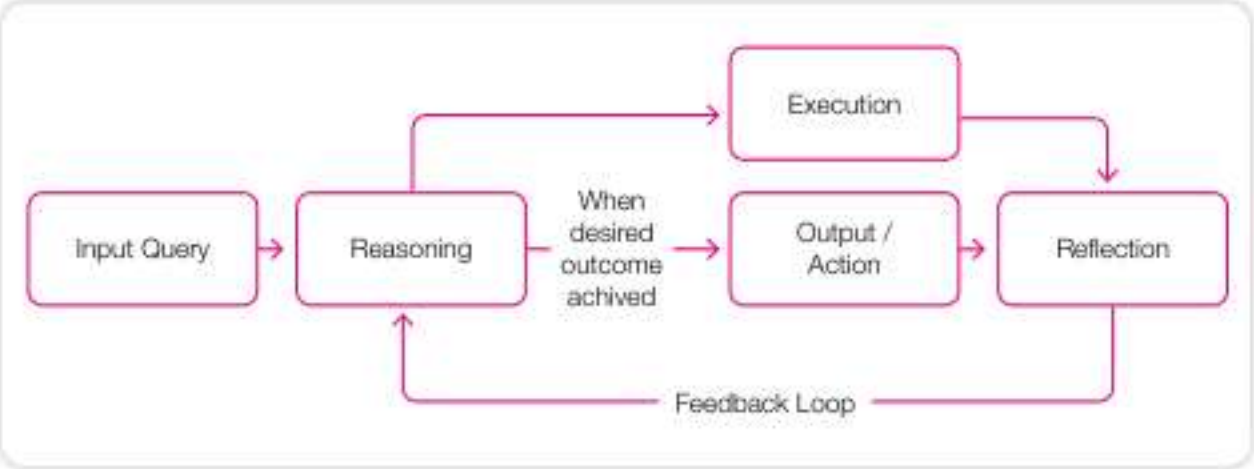


**Fig 1.7:** Workflow of self-reflecting agents

# Memory-Enhanced – The Personalized Powerhouses

Give an agent a little memory, and you have the ultimate personal assistant. Memory-enhanced agents bring personalization to the forefront by maintaining historical context and remembering user preferences, previous interactions, and task history. They act as adaptive personal assistants, providing tailored experiences and continuous, context-aware support. These agents remember your preferences, track your history, and theoretically — would never (ever) forget your coffee order! (See Table 1.8)

| Feature | Description |
|---|---|
| Intelligence | Possesses long-term memory, storing and recalling past interactions, preferences, and task progress. |
| Behavior | Provides context-aware personalization, adapting decisions and actions based on user-specific data and history. Learns and improves over time. |
| Scope | Excels at tasks requiring individualized experiences, tailored recommendations, and maintaining consistency across multiple interactions. |
| Best Use Cases | Personalized assistance, long-term interactions, tasks spanning multiple sessions. |
| Examples | Project management AI with task history, customer service bots tracking interactions, personalized shopping assistants. |

**Table 1.8:** Characteristics of memory-enhanced agents

Look at **Fig 1.8**: Starting with an Input Query, this agent first recalls relevant past experiences and preferences (Memory Recall), then uses this context for Reasoning about the current task. After deciding on a course of action, it executes it (Action/Execution), updates its memory with new information (Memory Update), and produces the Output.

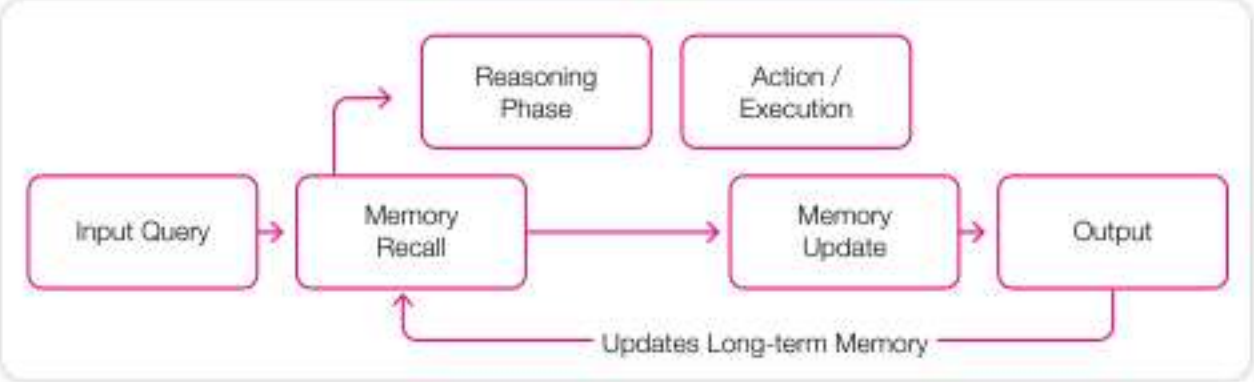


**Fig 1.8:** Workflow of memory-enhanced agents

# Environment Controllers – The World Shapers

Environment-controlling agents extend beyond decision-making and interaction—they actively manipulate and control environments in real time. These agents are equipped to perform tasks that influence the digital landscape or the physical world, making them ideal for applications in automation, robotics, and adaptive systems. Think smart thermostats, but on steroids! (See **Table 1.9**)

| Feature | Description |
|---|---|
| Intelligence | Autonomous learning; refines models and processes based on feedback, data, or environmental changes without manual updates. |
| Behavior | Adaptive and scalable, adjusting to changing conditions and new tasks. Exhibits evolutionary behavior, improving performance over time. |
| Scope | Suited for cutting-edge research and autonomous learning systems, offering high potential but requiring careful monitoring. |
| Best Use Cases | Situations where autonomous learning and adaptation are crucial, such as complex research, simulation, or dynamic environments. |
| Examples | Neural networks with evolutionary capabilities, swarm AI systems, autonomous robotics, financial prediction models. |

**Table 1.9:** Characteristics of environment-controlling agents

Observe the workflow in **Fig 1.9** carefully. Starting with an Input Query, the agent first observes its surroundings (Perception Phase), reasons about the current state and required changes (Reasoning Phase), takes action to modify the environment (Action Phase), and then receives feedback about the changes (Feedback Phase). This cycle repeats until the desired goal is met, producing both an Output and changed system state.

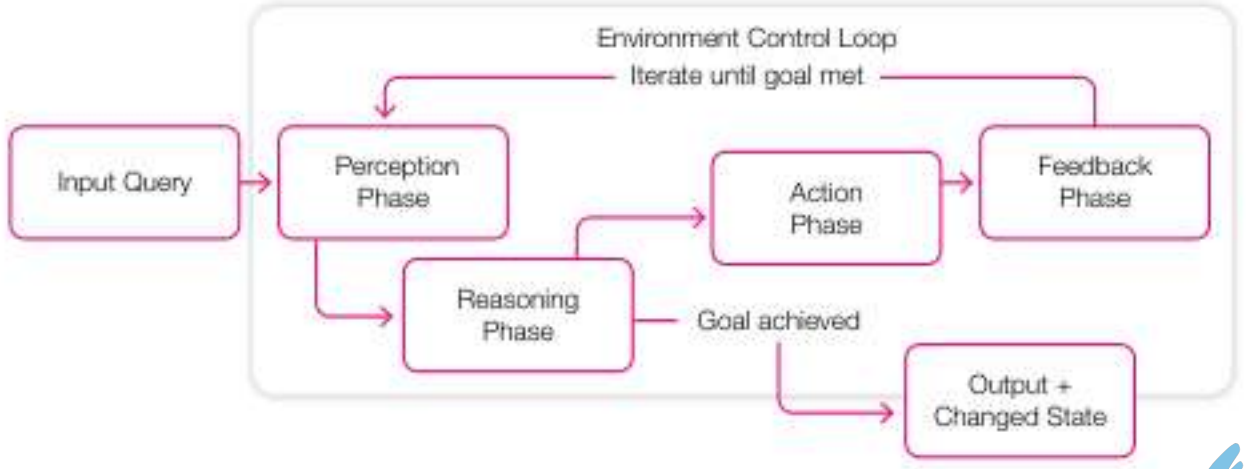


**Fig 1.9:** Workflow of an environment-controlled agent

# Self-Learning – The Evolutionaries

The holy grail of AI agents: those that can improve themselves over time. They learn, adapt, and evolve without needing constant babysitting. These agents improve themselves over time, learning from interactions, adapting to new environments, and evolving without constant human intervention. They combine elements of reasoning, memory, environment control, and self-reflection with autonomous learning capabilities to adapt and optimize their behavior.

Are they the future of AI? Potentially. Are they also terrifying? Without evaluations, observation, regulation, and oversight, very much so.

| Feature | Description |
|---|---|
| Intelligence | Autonomous learning; refines models and processes based on feedback, data, or environmental changes without manual updates. |
| Behavior | Adaptive and scalable, adjusting to changing conditions and new tasks. Exhibits evolutionary behavior, improving performance over time. |
| Scope | Suited for cutting-edge research and autonomous learning systems, offering high potential but requiring careful monitoring. |
| Best Use Cases | Situations where autonomous learning and adaptation are crucial, such as complex research, simulation, or dynamic environments. |
| Examples | Neural networks with evolutionary capabilities, swarm AI systems, autonomous robotics, financial prediction models. |

**Table 1.10:** Self-learning agents' characteristics

From the workflow in **Fig 1.10**, you'll realize how a self-learning agent are akin to an AI researcher that gets smarter with every experiment, constantly refining its methods and knowledge.

Starting with an Input Query, the agent enters a continuous cycle beginning with the Learning Phase where it processes available data, moves to Reasoning to analyze it, then takes Actions based on its analysis. The Feedback Phase evaluates results, leading to an Evolution Phase where the agent adapts and improves its models. This cycle repeats continuously, producing not just an Output but an evolved version of both the solution and the agent itself.

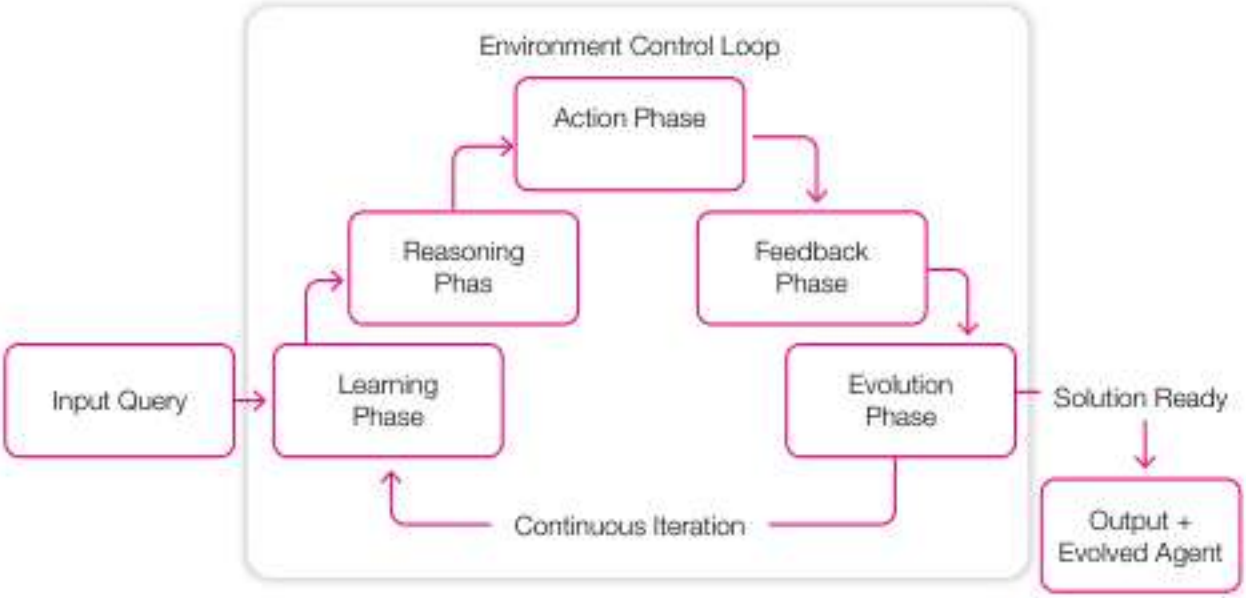


**Fig 1.10:** Workflow of a self-learning agent

What's fascinating is that each type has its own sweet spot—there's no "one-size-fits-all" solution. The key is matching the right agent type to your specific needs, whether you need the reliable consistency of fixed automation for routine tasks or the adaptive intelligence of self-learning agents for cutting-edge research.

# When to Use Agents?

We've looked at the agent types and where each one excels. That said, you still need to be able to gauge where you'll need an AI agent. Agents are highly beneficial when tasks require complex decision-making, autonomy, and adaptability. They excel in environments where the workflow is dynamic and involves multiple steps or interactions that can benefit from automation. You'll see how workflows in different domains can benefit from the use of AI agents in **Table 1.11** below:

| Domain | Task | Benefits of Using AI Agents |
|---|---|---|
| Customer Support | Handling queries, providing real-time assistance, issue escalation | Agents enhance the efficiency and customer experience by offering timely and accurate responses, allowing human staff to focus on more complex issues. |
| Research and Data Analysis | Gathering, processing, and analyzing data | They autonomously provide deep insights from large datasets, helping you understand patterns without manual effort. |
| Financial Trading | Real-time data processing | Agents excel in making quick decisions based on rapidly-changing market conditions. |
| Education | Personalized learning experiences | These agents adapt to each student's learning pace, offering tailored feedback and supporting unique learning journeys effectively. |
| Software Development | Code generation, debugging, and testing | Agents streamline the development process by handling repetitive tasks like coding and testing, improving code quality, and reducing development time. They also learn and improve over time, which continually enhances their assistance. |

**Table 1.11:** Domains and applications that can benefit from the use of AI agents

# When Not to Use Agents?

Agents offer many advantages, but there are certain scenarios in which deploying them might not be the best option.

If the tasks you're dealing with are straightforward, occur infrequently, or require only minimal automation, the complexity and cost of implementing AI agents might not make sense for you. Simple tasks that existing software solutions can handle efficiently do not necessarily benefit from the added intricacy of agent-based systems. In such cases, sticking with traditional methods can be more efficient and cost-effective.

Also, if your tasks require deep domain-specific knowledge or expertise—like conducting complex legal analyses, making intricate medical diagnoses, or handling high-stakes decision-making in unpredictable environments—these are typically better left to experienced professionals. When you rely solely on agents for these critical tasks, it can lead to suboptimal or even harmful outcomes.

That said, fields like psychotherapy, counseling, or creative writing thrive on the nuances of human emotion and the creative process—areas where agents largely fall short. In these domains, the human touch is irreplaceable and essential for achieving meaningful outcomes.

Implementing agents also requires a significant investment from you in terms of time, resources, and expertise. If you're running a small business or managing a project with a tight budget, the costs of developing and maintaining these agents may not justify the benefits. In highly regulated industries, your use of agents might be restricted due to compliance and security concerns as well, and ensuring agents adhere to stringent regulatory requirements can be very challenging and resource-intensive.

# 10 Questions to Ask Before You Consider an AI Agent

Before you consider using AI agents, you'll need to ask yourself a set of questions to help you evaluate if it's actually worth the time, capital, and resources you'll be putting into it:

**01** **What is the complexity of the task?**

Is the task simple and repetitive, or does it involve complex decision-making that could benefit from automation?

**02** **How often does the task occur?**

Is this a frequent task where automation could save significant time and resources, or is it a rare event that might not justify the investment?

**03** **What is the expected volume of data or queries?**

Will the agent be handle large volumes of data or queries where speed and efficiency are crucial?

**04** **Does the task require adaptability?**

Are the conditions under which the task is performed constantly changing, requiring adaptive responses that an AI can manage?

**05** **Can the task benefit from learning and evolving over time?**

Is there a benefit to having a system that learns from its interactions and improves its responses or strategies over time?

**06** **What level of accuracy is required?**

Is it critical that the task is performed with high accuracy, such as in medical or financial settings, where AI might need to meet high standards?

**07** **Is human expertise or emotional intelligence essential?**

Does the task require deep domain knowledge, human intuition, or emotional empathy that AI currently cannot provide?

**08** **What are the privacy and security implications?**

Does the task involve sensitive information that must be handled with strict privacy and security measures?

**09** **What are the regulatory and compliance requirements?**

Are there specific industry regulations or compliance issues that need to be addressed when using AI?

**10** **What is the cost-benefit analysis?**

Does the return on investment in terms of time saved, efficiency gained, and overall performance outweigh the costs of implementing and maintaining an AI system?

Take time to evaluate these questions; this will help you better determine if an AI agent fits your needs and how it could be effectively implemented to enhance your operations or services.

# 3 Interesting Real-World Use Cases of AI Agents

Now that we've learned what agents are and when to and when not to use them, it's time to go through some interesting real-world use cases of AI agents.

## 1. Wiley and Agentforce

**Company:**
Wiley

**AI Agent:**
Agentforce by Salesforce

**Use Case:**
Customer service automation

**Problem:**
Wiley faced challenges handling spikes in service calls during peak times, particularly at the start of new semesters when thousands of students use Wiley's educational resources.

**Need:**
The company needed an efficient customer service system to manage the increased volume and maintain positive customer experiences.

**Solution:**
Wiley invested in Salesforce's Agentforce, an AI agent designed to enhance customer service operations. This integration has significantly improved case resolution rates and faster resolution of customer queries, especially during peak times, such as the start of new semesters when demand spikes.

**ROI:**
A 40%+ increase in case resolution compared to their previous chatbot, a 213% ROI, and $230K in savings

## 2. Oracle Health and Clinical AI agent

**Company:**
Oracle Health

**AI Agent:**
Clinical AI Agen

**Use Case:**
Enhancing patient-provider interactions

**Problem:**
Healthcare providers faced documentation and time management challenges during patient visits, leading to burnout and reduced patient engagement.

**Need:**
There was a need for a solution that could streamline clinical workflows and improve documentation accuracy while allowing providers more time to interact with patients.

**Solution:**
Oracle Health developed its Clinical AI Agent, which automates documentation processes and enhances patient-provider interactions through a multimodal voice user interface. This allows providers to access patient information quickly and generate accurate notes efficiently.

**ROI:**
AtlantiCare, using the Clinical AI Agent, reported a 41% reduction in total documentation time, saving approximately 66 minutes per day, which translates to improved productivity and enhanced patient satisfaction.

# 3. Magid and Galileo

**Company:**
Magid

**AI Agent:**
RAG-based system powered with real-time observability capabilities

**Use Case:**
Empowering newsrooms with generative AI technology

**Problem:**
Magid, a leader in consumer intelligence for media brands, needed to ensure consistent, high-quality content in a fast-paced news environment. The complexity of diverse topics made it challenging to uphold accuracy, and errors could potentially lead to significant repercussions.

**Need:**
A robust observability system was essential for monitoring AI-driven workflows and ensuring the quality of outputs across various clients. This scalability was crucial for managing the daily production of numerous stories.

**Solution:**
Magid integrated Galileo's real-time observability capabilities into their product ecosystem. This integration provided production monitoring, relevant metrics for tracking tone and accuracy, and customization options tailored to Magid's needs.

**ROI:**
With Galileo, Magid achieved 100% visibility over inputs and outputs, enabling customized offerings as they scale. This visibility helps identify trends and develop client-specific metrics, enhancing the accuracy of news delivery.

We'll look at many more use cases across multiple domains throughout the rest of this e-book. We'll examine how agents have drivengreater

productivity, quicker resolutions, and helped things get done faster.

In the next chapter, we're going to learn features of three prominent frameworks for building AI agents. Lots of exciting stuff ahead!

# 02
# CHAPTER

# FRAMEWORKS FOR BUILDING AGENTS

# CHAPTER 2 FRAMEWORKS FOR BUILDING AGENTS

The first chapter examined what AI agents are and when to use them. Before we move on to the frameworks you can use to build these agents, let's do a quick recap.

AI agents are particularly useful for dynamic, complex environments like customer support or data-heavy sectors such as finance, where they automate and speed up processes. They're also great for personalizing education and streamlining software development.

However, they are not ideal for straightforward tasks that traditional software efficiently handles or for fields requiring deep expertise, empathy, or high-stakes decision making, where human judgment is crucial. The cost and regulatory compliance may also make them less viable for small projects or heavily regulated industries.

That said, the framework you choose to build these agents can significantly affect their efficiency and effectiveness. In this chapter, we'll evaluate three prominent frameworks for building AI agents — LangGraph, Autogen, and CrewAI — to help you make an informed choice.

# LangGraph vs. Autogen vs. CrewAI

Below are three frameworks you can consider when building AI agents:

## LangGraph

LangGraph is an open-source framework designed by Langchain to build stateful, multi-actor applications using LLMs. Inspired by the long history of representing data processing pipelines as directed acyclic graphs (DAGs), LangGraph treats workflows as graphs where each node represents a specific task or function.

This graph-based approach allows for fine-grained control over the flow and state of applications, making it particularly suitable for complex workflows that require advanced memory features, error recovery, and human-in-the-loop interactions. LangGraph integrates seamlessly with LangChain, providing access to various tools and models and supporting various multi-agent interaction patterns.

## Autogen

Autogen is a versatile framework developed by Microsoft for building conversational agents. It treats workflows as conversations between agents, making it intuitive for users who prefer interactive ChatGPT-like interfaces.

Autogen supports various tools, including code executors and function callers, allowing agents to perform complex tasks autonomously. The highly customizable framework allows you to extend agents with additional components and define custom workflows. Autogen is designed to be modular and easy to maintain, making it suitable for both simple and complex multi-agent scenarios.

## CrewAI

CrewAI is a framework designed to facilitate the collaboration of role-based AI agents. Each agent in is assigned specific roles and goals, allowing them to operate as a cohesive unit. This framework is ideal for building sophisticated multi-agent systems such as multi-agent research teams. CrewAI supports flexible task management, autonomous inter-agent delegation, and customizable tools.

## Practical Considerations

For practical consideration, let's compare LangGraph, Autogen, and CrewAI across several key aspects.

## How easy are they to use?

Ease of use determines how quickly and efficiently you can start using a framework. It also affects the learning curve and the time required to build and deploy agents.

Consider LangGraph. This framework visualizes workflows as graphs using directed acyclic graphs (DAGs). You'll find this approach intuitive if you're familiar with data processing pipelines. It makes it easier for you to visualize and manage complex interactions. You might need a deeper understanding of graph theories, which could initially steepen your learning curve.

Then there's Autogen, which models workflows as conversations between agents. If you prefer interactive, chat-based environments, this framework will likely feel more natural to you. Autogen simplifies the management of agent interactions, allowing you to focus more on defining tasks and less on the underlying complexities. This can be a great help when you're just starting out.

CrewAI, on the other hand, focuses on role-based agent design, where each agent has specific roles and goals. This framework is designed to enable AI agents to operate as a cohesive unit, which can be beneficial for building complex, multi-agent systems. It provides a structured approach to defining and managing agents. It's very straightforward to get started with CrewAI.

**Winner:** Autogen and CrewAI have an edge due to their conversational approach and simplicity.

## What tools and functionalities do they support?

Tool coverage is an essential aspect you'll want to consider when evaluating a framework. It refers to the range of tools and functionalities that a framework supports, enhancing the capabilities of your agents.

For instance, LangGraph offers robust integration with LangChain, which opens up a wide array of tools and models for your use. It supports functionalities like tool calling, memory, and human-in-the-loop interactions. This comprehensive integration allows you to tap into a broad ecosystem, significantly extending your agents' functionality. If your project requires a rich toolkit for complex tasks, LangGraph's capabilities might be particularly valuable.

Moving on to Autogen, this framework stands out with its support for various tools, including code executors and function callers. Its modular design is a key feature, simplifying the process of adding and integrating new tools as your project evolves. If flexibility and scalability are high on your list, Autogen's approach lets you adapt and expand your toolset as needed without much hassle.

Lastly, CrewAI is built on top of LangChain, which means it inherits access to all of LangChain's tools. It allows you to define and integrate custom tools tailored to your specific needs. This capability is ideal if you're looking to craft a highly customized environment for your agents.

**Winner:** LangGraph and Crew have an edge due to their seamless integration with LangChain, which offers a comprehensive range of tools. All the frameworks allow the addition of custom tools.

## How well do they maintain context?

Memory support is crucial for agents to maintain context across interactions, enabling them to provide more coherent and relevant responses. There are different types of memory that agents can use:

| Memory Type | Description |
|---|---|
| Short-Term Memory | Keeps track of recent interactions and outcomes. |
| Long-Term Memory | Stores insights and learnings from past interactions. |
| Entity Memory | Focuses on capturing details about specific entities. |
| Contextual Memory | Integrates short-term, long-term, and entity memories. |

**Table 2.1:** Memory types that agents can use

LangGraph supports built-in short-term, long-term, and entity memory, enabling agents to maintain context across interactions. It includes advanced features like error recovery and the ability to revisit previous states, which are helpful for complex problem-solving.

Autogen employs a conversation-driven approach to support memory, enabling agents to remember previous interactions and stay contextually aware. This setup ensures that agents maintain a coherent context throughout their interactions, which is essential for tasks that depend on continuity.

CrewAI features a comprehensive memory system that includes short-term, long-term, and entity memory. This system allows agents to accumulate experiences and enhance their decision-making capabilities over time, ensuring they can recall important details across multiple interactions.

**Winner:** Both LangGraph and CrewAI have an edge due to their comprehensive memory system, which includes short-term, long-term, and entity memory.

## Are They Well-Organized and Easy to Interpret?

Structured output is vital for ensuring that the responses generated by agents are well-organized and easily interpretable. Structured output can include JSON, XML, or other formats that facilitate further processing and analysis.

LangGraph allows nodes to return structured output, which can be used to route to the next step or update the state. This makes managing complex workflows easier and ensures the output is well-organized. An ideal use case is a customer service system that routes queries through different departments based on content analysis, urgency, and previous interaction history.

Autogen supports structured output through its function-calling capabilities. Agents can generate structured responses based on the tools and functions they use. This ensures that the output is well-defined and can be easily processed by other components. A coding assistant system where multiple specialized agents (code writer, reviewer, tester) need to work together dynamically is a good use case to think of.

CrewAI supports structured output by allowing agents to parse outputs as Pydantic models or JSON. This ensures that the output is well-organized and easily interpretable. You can define the structure of the output to meet their specific requirements. For example, consider a data processing pipeline in which multiple agents need to transform and validate data according to specific schemas.

**Winner:** LangGraph and CrewAI have an edge due to their ability to define structured output.

## What's the Quality of Documentation?

Documentation quality affects how easily developers can understand and use the framework. Good documentation can reduce the learning curve and improve the overall developer experience.

LangGraph provides comprehensive documentation, including detailed guides and examples. The documentation is well-structured, making it easy to find the information you need. It covers various aspects of the framework, from basic concepts to advanced features.

Autogen has documentation with numerous examples and tutorials. The documentation covers various aspects of the framework, making it accessible to beginners and advanced users alike. It includes detailed explanations of key concepts and features.

CrewAI provides detailed documentation, including how-to guides and examples. The documentation is designed to help you get started quickly and understand the framework's core concepts. It includes practical examples and step-by-step instructions.

**Winner:** All frameworks have excellent documentation, but it's easy to find more examples of LangGraph and CrewAI.

## Do They Provide Multi-Agent Support?

Multi-agent support is crucial when you're dealing with complex applications that involve various interaction patterns among multiple agents. This includes:

- Hierarchical
- Sequential
- Dynamic interactions

When agents are grouped by tools and responsibilities, they tend to perform better because focusing on a specific task typically yields better results than when an agent

must choose from many tools. Giving each prompt its own set of instructions and few-shot examples can further boost performance. Imagine each agent powered by its own finely-tuned large language model—this provides a practical framework for development, allowing you to evaluate and improve each agent individually without affecting the broader application.

LangGraph supports various multi-agent patterns, including hierarchical and dynamic group chats. It lets you easily define complex interaction patterns between agents. Its graph-based approach aids in visualizing and managing these interactions effectively. In LangGraph, you explicitly define different agents and their transition probabilities as nodes in a graph. This method gives you extensive control over constructing complex workflows, which is essential for managing transition probabilities between nodes.

Autogen emerged as one of the first multi-agent frameworks, framing workflows more as "conversations" between agents. This conversational model adds flexibility, allowing you to define how agents interact in various patterns, including sequential and nested chats. Autogen's design simplifies the management of these complex multi-agent interactions, enabling effective collaboration among agents.

CrewAI supports role-based interactions and autonomous delegation among agents. It facilitates processes like sequential and hierarchical task execution, which are critical for efficiently managing multi-agent interactions. This setup ensures that agents can work together seamlessly to achieve common goals. CrewAI provides a higher-level approach than LangGraph, focusing on creating cohesive multi-agent "teams."

**Winner:** LangGraph has an edge due to its graph-based approach, which makes it easier to visualize and manage complex interactions.

## What About Caching?

Caching is critical for enhancing agent performance by reducing latency and resource consumption. It does this by storing and reusing previously computed results, which can significantly speed up operations.

LangGraph supports caching through its built-in persistence layer. This allows you to save and resume graph execution at any point. The caching mechanism ensures that previously computed results can be reused, improving performance as well.

AutoGen supports caching API requests so they can be reused when the same request is issued.

All tools in CrewAI support caching, which enables agents to reuse previously obtained results efficiently. This reduces the load on external resources and speeds up the execution time. The cache_function attribute of the tool allows you to define finer control over the caching mechanism.

**Winner:** All frameworks support caching, but LangGraph and CrewAI might have an edge.

## Looking at the Replay Functionality

Replay functionality allows you to revisit and analyze previous interactions, which is useful for debugging and improving agent performance. This helps you understand the decision-making process and identify areas for improvement.

LangGraph enhances your debugging and experimentation capabilities with its time travel feature. This allows you to rewind and explore different scenarios easily. It provides a detailed history of interactions, enabling thorough analysis and understanding of each step in your process.

While Autogen does not offer an explicit replay feature, it does allow you to manually update the state to control the agent's trajectory. This workaround provides some level of replay functionality, but it requires more hands-on intervention from you.

CrewAI provides the ability to replay from a task specified from the latest crew kickoff. Currently, only the latest kickoff is supported, and it will only allow you to replay from the most recent crew run.

**Winner:** LangGraph and CrewAI make it easy to replay with inbuilt capabilities.

## What About Code Execution?

Code execution capabilities enable agents to perform complex tasks by writing and executing code. This is particularly useful for tasks that require dynamic calculations or interactions with external systems.

LangGraph integrates with LangChain to support code execution within its workflows. You can define nodes specifically for executing code, which becomes part of the

overall workflow. This integration means you can seamlessly incorporate complex code executions into your projects.

Autogen supports code execution through its built-in code executors. Agents can write and execute code to perform tasks autonomously. The framework provides a safe environment for code execution, ensuring that agents can perform tasks securely.

CrewAI supports code execution through customizable tools. You can define tools that execute code and integrate them into the agent's workflow. This provides flexibility in defining the capabilities of agents and allows for dynamic task execution.

**Winner:** Autogen might have a slight edge due to its built-in code executors, but the other two are also capable.

## Human in the Loop Support?

Human-in-the-loop interactions allow agents to receive human guidance and feedback, improving their performance and reliability. This is particularly important for tasks that require human judgment or intervention.

LangGraph supports human-in-the-loop interactions through its interruption features. You can pause the graph execution to provide feedback or make adjustments.

Autogen supports human-in-the-loop interactions through its three modes: NEVER, TERMINATE, and ALWAYS.

CrewAI supports human-in-the-loop interactions by allowing agents to request human input during task execution by setting the ***human_input*** flag in the task definition. When enabled, the agent prompts the user for input before delivering its final answer.

**Winner:** All frameworks support humans in the loop in different ways.

## How Well Do They Accommodate Customization?

Customization options determine how easily you can tailor the framework to your specific needs and requirements. This includes the ability to define custom workflows, tools, and interactions.

LangGraph provides fine-grained control over the flow and state of the application. You can customize the behavior of nodes and edges to suit specific needs. The framework's graph-based approach also makes it easy to define complex workflows.

Autogen is customizable, allowing users to extend agents with additional components and define custom workflows. The framework is designed to be modular and easy to maintain.

CrewAI offers extensive customization options, including role-based agent design and customizable tools.

**Winner:** All the frameworks provide customization, but the mileage might vary.

## How Good Are They At Scaling?

Scalability is a must to ensure that the framework can grow alongside your requirements. The framework should sustain its performance and reliability as you incorporate more agents, tools, and interactions. We have no winners here. All three frameworks offer the flexibility to scale the system by adding agents, tools, and customizations according to your needs.

**Winner:** It remains unclear which framework scales more effectively as more elements are added. We recommend experimenting with them to get a better idea.

## Let's Compare Them All

Well, that's a lot of information to process at once! Refer to the table below (Table 2.2) for a quick overview of what we discussed in this chapter.

**To sum it up:**

**LangGraph** excels in scenarios where workflows can be represented as graphs

**Autogen** is ideal for conversational workflows

**Crew**AI is designed for role-based multi-agent interactions

| Criteria | LangGraph | Autogen | CrewAI | Final Verdict |
|---|---|---|---|---|
| Ease of Usage | ✗ | ✓ | ✓ | Autogen and CrewAI are more intuitive due to their conversational approach and simplicity. |
| Multi-Agent Support | ✓ | ✓ | ✓ | CrewAI excels with its structured role-based design and efficient interaction management among multiple agents. |
| Tool Coverage | ✓ | ✓ | ✓ | LangGraph and CrewAI have a slight edge due to their extensive integration with LangChain. |
| Memory Support | ✓ | ✓ | ✓ | LangGraph and CrewAI are advanced in memory support features, ensuring contextual awareness and learning over time. |
| Structured Output | ✓ | ✓ | ✓ | LangGraph and CrewAI have strong support for structured outputs that are versatile and integrable. |
| Documentation | ✓ | ✓ | ✓ | LangGraph and CrewAI offer extensive and well-structured documentation, making it easier to get started and find examples. |
| Multi-Agent Pattern Support | ✓ | ✓ | ✓ | LangGraph stands out due to its graph-based approach, which makes it easier to visualize and manage complex interactions. |
| Caching | ✓ | ✓ | ✓ | LangGraph and CrewAI lead with comprehensive caching mechanisms that enhance performance. |
| Replay | ✓ | ✗ | ✓ | LangGraph and CrewAI have inbuilt replay functionalities, making them suitable for thorough debugging. |
| Code Execution | ✓ | ✓ | ✓ | Autogen takes the lead slightly with its innate code executors, but others are also capable. |
| Human in the Loop | ✓ | ✓ | ✓ | All frameworks provide effective human interaction support and are equally strong in this criterion. |
| Customization | ✓ | ✓ | ✓ | All the frameworks offer high levels of customization, serving various requirements effectively. |
| Scalability | ✓ | ✓ | ✓ | All frameworks are capable of scaling effectively, recommend experimenting with each to understand the best fit. |
| Open source LLMs | ✓ | ✓ | ✓ | All frameworks support open-source LLMs. |

**Table 2.2:** Overview of comparisons between LangGraph, Autogen, and CrewAI on core features, technical capabilities, and development experience

# Popular Use Cases Centered Around These Frameworks

All the comparisons aside, here are some interesting use cases and collaborations centered around LangGraph, Autogen, and CrewAI.

## LangGraph

Chaos Labs has developed the Edge AI Oracle using LangChain and LangGraph for enhanced decision-making in prediction markets. This system utilizes a multi-agent council to ensure accurate, objective, and transparent resolutions. Each agent, ranging from data gatherers to bias analysts and summarizers, plays a role in processing queries through a decentralized network. This setup effectively reduces single-model biases and allows for consensus-driven, reliable outputs in high-stakes environments.

## Autogen

Built on top of Autogen, OptiGuide employs LLMs to simplify and enhance supply chain operations. It integrates these models to analyze and optimize scenarios efficiently, such as assessing the impact of different supplier choices. The system ensures data privacy and doesn't transmit proprietary information. Applied within Microsoft's cloud infrastructure for server placement, OptiGuide improves operational efficiency and stakeholder communication and reduces the need for extensive manual oversight.

## CrewAI

Waynabox has transformed travel planning by partnering with CrewAI, offering personalized, hassle-free travel experiences. This collaboration utilizes CrewAI's multi-agent system to automatically generate tailored itineraries based on real-time data and individual preferences. The integration of AI agents—handling activities, preferences, and itinerary customization—allows travelers to enjoy unique adventures without the stress of planning. This has helped simplify itinerary planning and enhanced Waynabox's service to create a more exciting and seamless travel experience.

In this chapter, we reviewed three frameworks, LangGraph, Autogen, and CrewAI, and how they compare in different aspects, such as ease of use, multi-agent support, and others (See Table 2.2). We also looked at examples of companies that have used these frameworks in different scenarios and domains to ultimately focus on three factors: reduction of manual "redundant" work, seamless operations, and productivity improvement.

However, it is also imperative to consider the accuracy and reliability of AI agents. This takes us to the next chapter, where we'll examine the importance of careful monitoring and feedback to ensure they provide reliable, well-sourced information, necessitating evaluation.

```python
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
from langgraph.prebuilt import create_react_agent

system_prompt = "You are a helpful finance expert named Fred in year 2024. First of all you create a plan to get answer to the research query. Then you use tools to get answers to the questions. Finally you use the answers to each question in the plan to give your final verdict."

llm = ChatOpenAI(model="gpt-4o-mini")
tools = [TavilySearchResults(max_results=3)]
agent_executor = create_react_agent(llm, tools, state_modifier=system_prompt)
```

# 03 CHAPTER

# HOW TO EVALUATE AGENTS

# HOW TO EVALUATE AGENTS

In the previous chapter, we examined three frameworks, LangGraph, Autogen, and CrewAI, and some interesting use cases related to them.

The next important step in our journey is to understand how we can ensure the accuracy and reliability of AI agents. Why is this important in the first place?

Evaluating AI agents is like checking the work of a new employee. You have to make sure they're doing their job correctly and reliably. Without regular checks and constructive feedback, it's tough to trust that the information the agents provide is accurate and helpful.

The best way to understand this is through an example. So, in this chapter, we're going to build a financial research agent, and we'll cover how, much like humans, agents can be taught to solve problems by first understanding the issue, making a plan, taking action, and lastly, evaluating the result.

Let's jump in!

## Requirements

You can install these dependencies in a Python 3.11 environment.

*pip install --quiet -U langgraph==0.2.56 langchain-community==0.3.9*

*langchain-openai==0.2.11 tavily-python==0.5.0 promptquality==0.69.1*

To do so, sign up on Tavily and OpenAI to generate an API key. Save the keys in a .env file, as shown below.

*OPENAI_API_KEY=KKK*

*TAVILY_API_KEY=KKK*

## Defining the Problem

This chapter aims to build a financial research agent that "thinks" through and acts on problems within a financial dataset. We can create a workflow that receives a question, breaks it down into granular questions, searches the web using Tavily, and analyzes the results.

To analyze the results, we use the ReAct agent, which works with the Tavily API to think through and act on problems.

## Define the ReAct Agent

Within your IDE of choice, you can create a new Jupyter Notebook agent.ipynb.

We can import a prebuilt ReAct agent along with a web search tool called Tavily. While we use the same agent for all steps in this example, you could use different agents for different tasks. The best part? You can customize it further in later examples.

Look at **Fig 3.1** to understand this better. This code sets up an AI-driven chat agent named Fred, designed to function as a finance expert in 2024. Fred will use specific tools and a planning framework to research and answer questions.

```
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
from langgraph.prebuilt import create_react_agent

system_prompt = "You are a helpful finance expert named Fred in year 2024. First of all you
create a plan to get answer to the research query. Then you use tools to get answers to the
questions. Finally you use the answers to each question in the plan to give your final
verdict."

llm = ChatOpenAI(model="gpt-4o-mini")
tools = [TavilySearchResults(max_results=3)]
agent_executor = create_react_agent(llm, tools, state_modifier=system_prompt)
```

**Fig. 3.1:** Setting up the agent

## State Management

Now, let's talk about how our agent keeps track of everything it needs to do. Think of it like a smart to-do list system with three main parts.

First, we need a way to track what the agent plans to do. We'll use a simple list of steps written as text strings. This is like having a checklist of tasks the agent needs to complete.

Second, we want to remember what it has already done and what happened with each task. For this, we'll use a list of pairs (or tuples in programming terms). Each pair contains both the action taken and what resulted from that action.

Lastly, we need to store two more important pieces of information: the original question that was asked (the input) and the final answer once the agent finishes its work (the response).

This setup gives our agent everything it needs to function effectively.

In Fig 3.2, the *PlanExecute* class, a dictionary type, manages an execution process, including input, plan steps, previous steps, and a response. The *Plan* class, using Pydantic, defines a structured plan with steps that should be followed in a sorted order.

```
import operator
from pydantic import BaseModel, Field
from typing import Annotated, List, Tuple
from typing_extensions import TypedDict


class PlanExecute(TypedDict):
    input: str
    plan: List[str]
    past_steps: Annotated[List[Tuple], operator.add]
    response: str


class Plan(BaseModel):
    """Plan to follow in future"""

    steps: List[str] = Field(
        description="different steps to follow, should be in sorted order"
    )
```

**Fig. 3.2:** Defining structures for managing and executing a sequential plan of actions

The planning step is where our agent will begin to tackle a research question. We'll use a special feature called function calling to create this plan. Let's break down how it works.

First, we create a template for how our agent should think. We tell it that it's a finance research agent working in October 2024, and its job is to break down big questions into smaller, manageable steps.

This template, called *planner_prompt* (See **Fig 3.3**), gives our agent clear instructions: create a simple, step-by-step plan where each step leads logically to the next. Ensure that no steps are missing or unnecessary. The final step should give us our answer.

The code sets this up by using *ChatPromptTemplate*, which has two main parts:

- A system message that explains the agent's role and how it should plan
- A placeholder for the messages we'll send it

```
from langchain_core.prompts import ChatPromptTemplate

planner_prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are a finance research agent working in Oct 2024. For the given
objective, come up with a simple step by step plan. \
This plan should involve individual tasks, that if executed correctly will yield the
correct answer. Do not add any superfluous steps. The result of the final step should be
the final answer. Make sure that each step has all the information needed - do not skip
steps. At the end use the info collected to give the final answer to the main question
containing the facts.""",
        ),
        ("placeholder", "{messages}"),
    ]
)
```

**Fig. 3.3:** Guiding the agent to create a step-by-step plan that should lead to the correct answer for a given objective

We then connect this template to **ChatOpenAI** using gpt-4o-mini with *temperature* set to 0 for consistent results. We take gpt-4o-mini being low on cost. The "structured output" part means the plan will come out in a specific format we can easily work with.

When we test it with a real question like "Should we invest in Tesla given the current situation of EVs?" the agent will create a detailed plan for researching this investment decision. Each step will help gather the information needed to make an informed recommendation about Tesla stock based on the current electric vehicle market conditions. (See **Fig 3.4**)

Think of it like creating a research roadmap. We're giving our agent the tools and guidelines it needs to break down complex questions into manageable research tasks.

**Fig. 3.4:** Testing the agent with a question

Think of re-planning as the agent's ability to adjust its strategy based on what it has already learned. This is similar to how we might revise our research approach after discovering new information. Let's break down how this works.

First, we create two types of possible actions the agent can take:

- **Response**: When the agent has enough information to answer the user's question
- **Plan**: When the agent needs to do more research to get a complete answer

The re-planning prompt is like giving our agent a structured way to think about what to do next. It looks at three things:

- The original question (objective)
- The initial plan it made
- What steps have already been completed and what was learned

Using this information, the agent can decide to either:

- Create new steps to gather more needed information
- Give a final answer if it has enough information

The clever part is that the agent won't repeat steps it's already done. It focuses only on what still needs to be investigated. This makes the research process more efficient and prevents redundant work. It's like having a research assistant who can intelligently adjust their approach based on what they've already discovered.

This process helps our agent stay focused and efficient, only pursuing new information when needed and knowing when it's time to provide a final answer to the user.

We connect this re-planning ability to gpt-4o with the temperature set to 0. By setting the *temperature* to 0 (See **Fig 3.5**), we force the model to generate the same response for the same input. This helps us in making experiments reproducible.

```
from typing import Union


class Response(BaseModel):
    """Response to user."""

    response: str


class Act(BaseModel):
    """Action to perform."""

    action: Union[Response, Plan] = Field(
        description="Action to perform. If you want to respond to user, use Response. "
        "If you need to further use tools to get the answer, use Plan."
    )


replanner_prompt = ChatPromptTemplate.from_template(
    """For the given objective, come up with a simple step by step plan. \
This plan should involve individual tasks, that if executed correctly will yield the
correct answer. Do not add any superfluous steps. \
The result of the final step should be the final answer. Make sure that each step has all
the information needed - do not skip steps.

Your objective was this:
{input}

Your original plan was this:
{plan}

You have currently done the follow steps:
{past_steps}

Update your plan accordingly. If no more steps are needed and you can return to the user,
then respond with that. Otherwise, fill out the plan. Only add steps to the plan that still
NEED to be done. Do not return previously done steps as part of the plan."""
)


replanner = replanner_prompt | ChatOpenAI(
    model="gpt-4o", temperature=0
).with_structured_output(Act)
```

**Fig. 3.5:** Replanner_prompt to review and update a given plan based on past actions

## Create the Graph

Think of this graph as a roadmap that shows how our agent moves from one task to another. We have three main functions that work together:

```
from typing import Literal
from langgraph.graph import END


async def execute_step(state: PlanExecute):
    plan = state["plan"]
    plan_str = "\n".join(f"{i+1}. {step}" for i, step in enumerate(plan))
    task = plan[0]
    task_formatted = f"""For the following plan:
{plan_str}\n\nYou are tasked with executing step {1}, {task}."""
    agent_response = await agent_executor.ainvoke(
        {"messages": [("user", task_formatted)]}
    )
    return {
        "past_steps": [(task, agent_response["messages"][-1].content)],
    }


async def plan_step(state: PlanExecute):
    plan = await planner.ainvoke({"messages": [("user", state["input"])]})
    return {"plan": plan.steps}


async def replan_step(state: PlanExecute):
    output = await replanner.ainvoke(state)
    if isinstance(output.action, Response):
        return {"response": output.action.response}
    else:
        return {"plan": output.action.steps}


def should_end(state: PlanExecute):
    if "response" in state and state["response"]:
        return END
    else:
        return "agent"
```

**Fig. 3.6:** Managing and executing using state-based logic

The ***execute_step*** function handles individual tasks. It takes the first item from our plan, formats it properly, and has the agent work on it. It's like giving a specific assignment to a research assistant and getting back their findings. The agent keeps track of what it did and what it learned.

The ***plan_step*** function is where everything begins. When given a question, it creates the initial research plan. This is like creating a first draft of how to tackle the problem.

The ***replan_step*** function is where the agent decides what to do next. After completing a task, it looks at what it has learned and either:

- Creates new steps if more research is needed
- Provides a final answer if it has enough information

Finally, we have the ***should_end*** function, which works like a checkpoint. It checks whether we have a final answer ready. If we do, it ends the process. If not, it tells the agent to continue working. You can see all these functions in the code snippet below, in ***Fig 3.6.*** We use StateGraph to create a map that guides our agent through its research journey via different actions it can take. Here's how it flows:

First, we create the basic structure of the workflow with its three main stops:

- A planning station ("planner")
- A research station ("agent")
- A reviewing station ("replan")

Then, we connect these stations in a logical order:

1. Everything starts at the planning station
2. From planning, the agent moves to doing research
3. After research, it goes to reviewing what it learned

At the reviewing station, the agent makes an important decision:

- Either continue with more research if needed
- Or finish up if it has a complete answer

This creates a smooth cycle in which the agent can continue researching until it has everything it needs to answer the original question. It's like having an intelligent research assistant who knows when to dig deeper and when they've found enough information.

Finally, we compile this workflow into something we can easily use, just like any other tool in our system. This makes our research agent ready to tackle real questions and provide thorough, well-researched answers. See **Fig 3.7**.

```python
from langgraph.graph import StateGraph, START

workflow = StateGraph(PlanExecute)

# Add the plan node
workflow.add_node("planner", plan_step)

# Add the execution step
workflow.add_node("agent", execute_step)

# Add a replan node
workflow.add_node("replan", replan_step)

workflow.add_edge(START, "planner")

# From plan we go to agent
workflow.add_edge("planner", "agent")

# From agent, we replan
workflow.add_edge("agent", "replan")

workflow.add_conditional_edges(
    "replan",
    # Next, we pass in the function that will determine which node is called next.
    should_end,
    ["agent", END],
)

# Finally, we compile it!
# This compiles it into a LangChain Runnable,
# meaning you can use it as you would any other runnable
app = workflow.compile()
```

**Fig. 3.7:** Creating the structure of the workflow

We can visualize the agent workflow with a mermaid diagram, as shown in Fig 3.8. See the output in Fig 3.9.

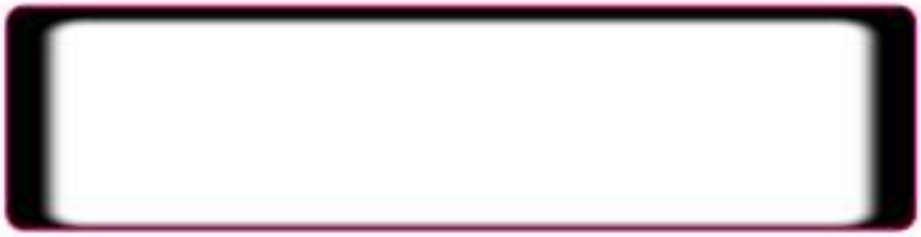

**Fig. 3.8:** Visualizing the workflow using Mermaid Chart

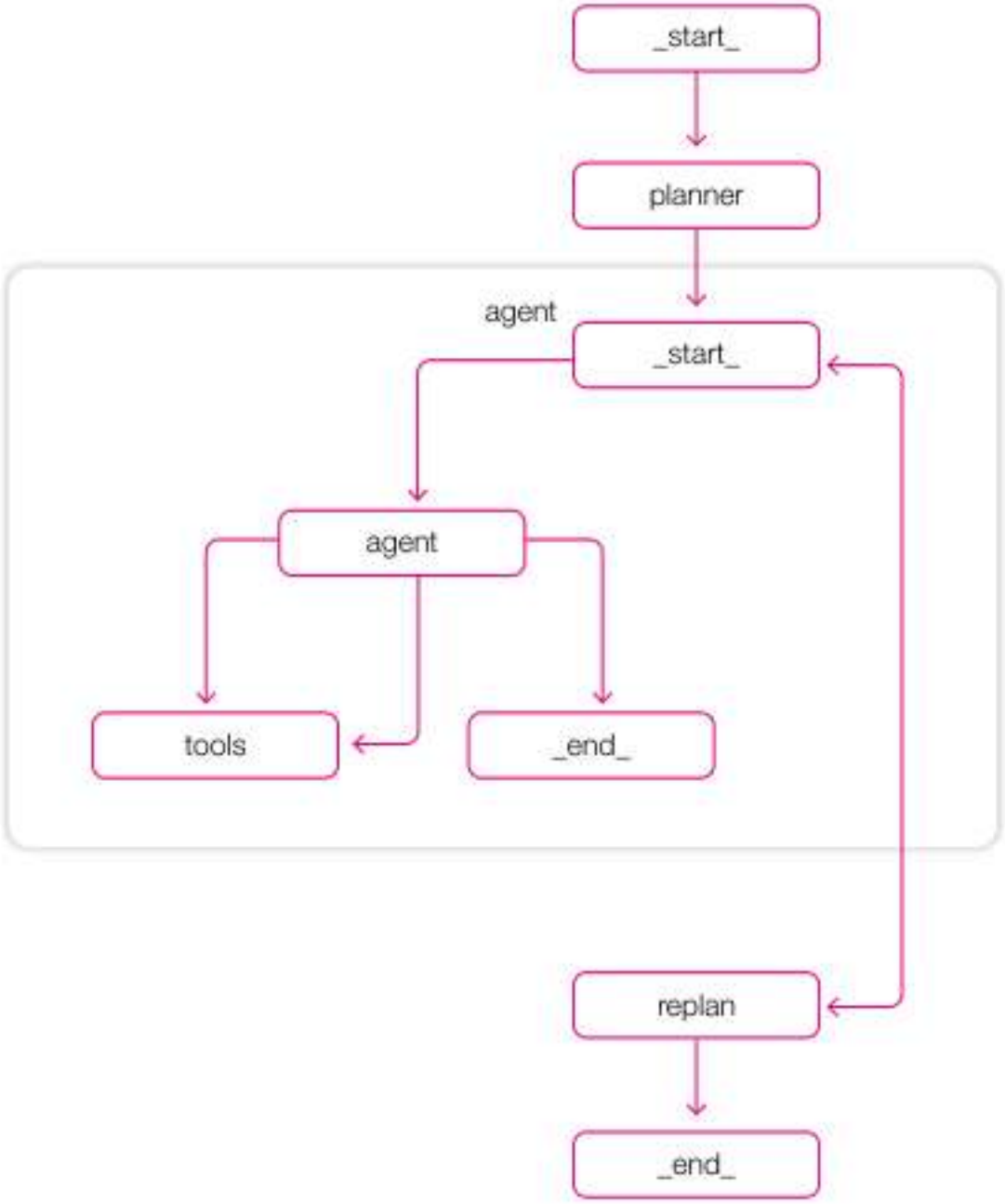


**Fig. 3.9:** Mermaid Chart workflow output

## Create the LLM Judge

Next, we create a LLM judge to evaluate our agent's performance. This ensures our agents' responses adhere to the given context and maintain relevance and accuracy.

The inbuilt scorers make it very easy to set up one for us. We use gpt-4o as our LLM for the context adherence metric, with three evaluations per response for better to ensure great evaluation accuracy. This scorer specifically looks at how well the agent sticks to the context and provides relevant information.

Note that we're using GPT-4o to evaluate a smaller AI model, which is like having an expert oversee a novice's work. GPT-4o, with its advanced capabilities and deep understanding of language nuances, can be a reliable benchmark for judging the smaller model's (in our case, the 4o-mini) responses. See **Fig 3.10.**

```
gpt4o_scorer = pq.CustomizedChainPollScorer(
                        scorer_name=pq.CustomizedScorerName.context_adherence_plus,
                        model_alias=pq.Models.gpt_4o,
                        num_judges=3)
```

**Fig. 3.10:** Implementing the LLM as Judge functionality

We then set up a Galileo evaluation callback that will track and record our agent's performance. It's like having a quality control system that monitors our research process. Next, we set some basic config for our agent:

- It can't go through more than 30 cycles (recursion_limit).
- It must use our evaluation system (callbacks).

## Use Galileo Callbacks

You'll observe in Fig 3.11 that we're using the Galileo callback, GalileoPromptCallback, which is used to log the execution of chains in applications like Langchain.

With just two lines of code, we can get all the information needed to visualize and debug the traces.

```
evaluate_handler = pq.GalileoPromptCallback(project_name='finance-research-agent',
run_name=f'test', scorers=[gpt4o_scorer])

config = {"recursion_limit": 30, "callbacks": [evaluate_handler]}
```

**Fig. 3.11:** Galileo Callback

We then run our agent with a specific test question. The system will process this question through the research workflow we built earlier.

The code is set up to show us what's happening at each step (that's what the async for loop does). It will print out each action and result as they happen, letting us watch the research process in real-time.

Finally, we close our evaluation session with evaluate_handler.finish(). This saves all the performance data we collected during the run to the Galileo Evaluate console so we can see the chain visualization and the agent metrics. See Fig 3.12 and Fig 3.13.

```
inputs = {"input": "Should we invest in Tesla given the current situation of EV?"}

async for event in app.astream(inputs, config=config):
    for k, v in event.items():
        if k != "__end__":
            print(v)

evaluate_handler.finish()
```

**Fig. 3.12:** Closing the evaluation session


**Fig. 3.13:** Chain visualization

You can run several experiments to evaluate the research agent's performance. For instance, you can use the project dashboard to see how different test runs performed based on key metrics (see **Figure 3.14**).

The standout performer was test-3, which earned the top rank with impressive results.

**Performance of test-3:**

- Context Adherence Score: 0.844 (High relevance to the research questions)
- Speed: Completed tasks in 84,039 milliseconds (Fastest among all tests)
- Responses Processed: 3 during the run
- Cost: $0.0025 per run (Low cost)

**Overall Test Performances:**

- Response Time Range: From 134,000 to 228,000 milliseconds
- Context Adherence Score Range: From 0.501 to 0.855
- Number of Responses: Ranged from 1 to 7 per test
- Cost Efficiency: Remained consistent across all runs, between $0.002 and $0.004 per run

These results give valuable insights into our agent's capabilities and help identify the most effective configuration for future research tasks.

**Fig. 3.14:** Galileo's dashboard that shows multiple runs

Now, you can go inside each test run to see agent executions (See **Fig 3.15**). The dashboard reveals seven different research queries that our agent processed. Each query focused on analyzing different companies' financial metrics. Here's what you'll observe:

- The agent shows varying performance across different samples
- There's an alert noting that six samples had latency greater than 10 seconds, which suggests room for optimization.
- Average Latency for run: 210,623 ms
- Average Cost for run: $0.004 per execution

This detailed view helps you understand where the agent performs well and where it might need improvements in terms of speed and accuracy.

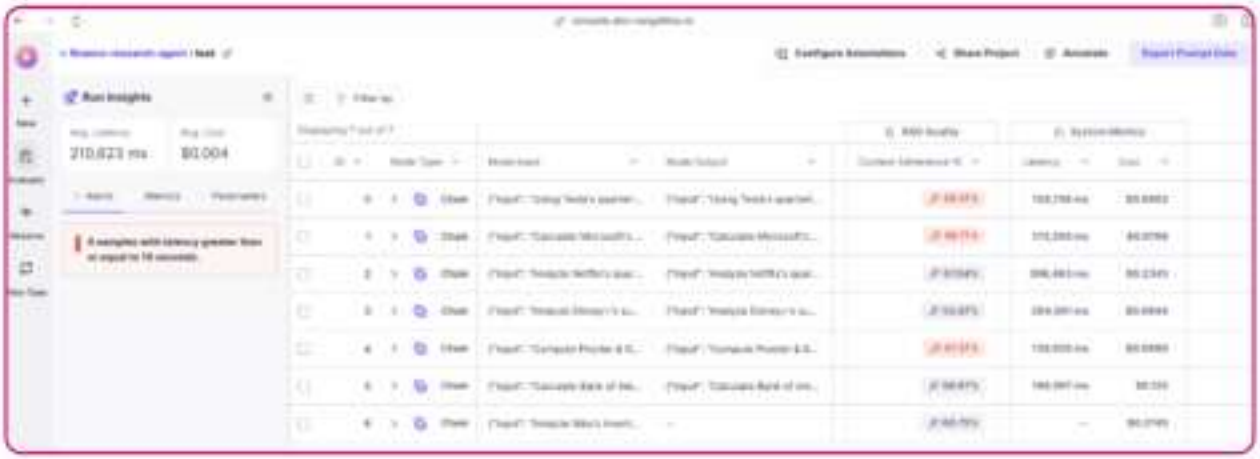

**Fig. 3.15:** Detailed view for each test run

Looking at the trace view (**Fig 3.16**), you can see a detailed breakdown of an execution chain where the context adherence was notably low at 33.33%. The system explanation helps us understand why:

*"The response has a 33.33% chance of being consistent with the context. Based on the analysis, while some of the figures like those for later 2022 and 2023 are supported by document references (such as Q3 2023 and Q4 2023), many earlier quarters' figures lack direct evidence from the documents or explicit mentions, leading to incomplete support for claims."*

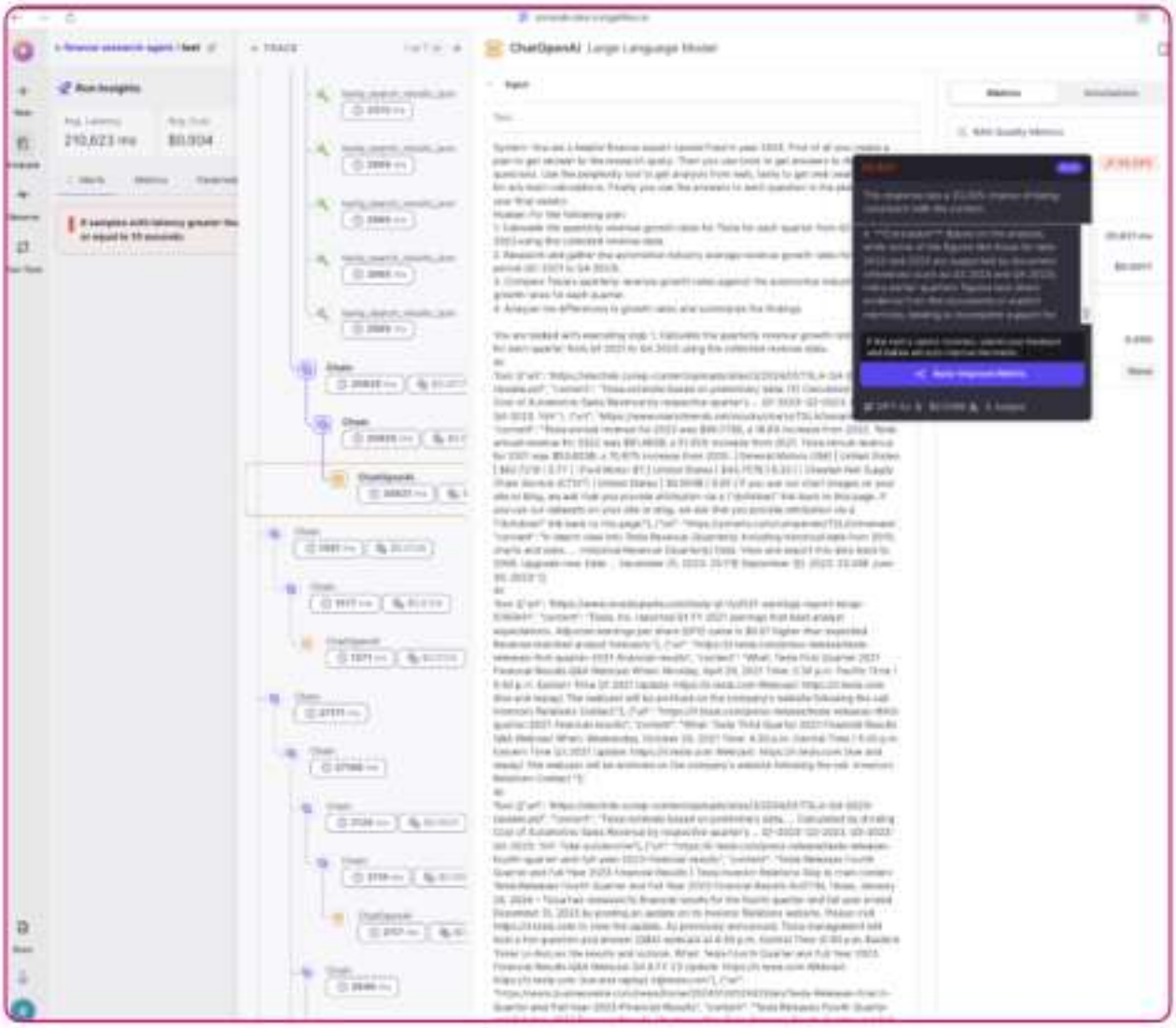

**Fig. 3.16:** Detailed breakdown of an execution chain to help with evaluation

This reveals two key issues in our agent's performance:

| | |
|---|---|
| The agent is citing recent data (2022-2023) correctly with proper sources. | However, it's making claims about earlier data without proper documentation or references. |

To improve this, there are two possible main paths:

1. Improve the retrieval system:
   - Make sure we're gathering sufficient historical data.
   - Expand the search scope to include earlier quarterly reports.
   - Better source verification for historical data.
2. Enhance the prompts:
   - Add explicit instructions to cite sources for all numerical claims.
   - Include requirements to clearly distinguish between verified and unverified data.
   - Add checks for data completeness before making comparisons.

Let's take a quick look at what we learned in the chapter. We saw how our agent implemented the ReAct (Reasoning and Acting) framework to:

- Break down complex questions into smaller steps
- Plan and execute research tasks systematically
- Re-evaluate and adjust its approach based on findings

We also explored the evaluation process using:

- An LLM judge (GPT-4) to assess response quality
- Metrics like context adherence, speed, and cost efficiency
- Galileo's evaluation dashboard for performance tracking

That said, testing the finance research agent in this chapter teaches you something very important and valuable: an AI is only as good as our ability to check its work. By looking closely at how the agent performed, you could see exactly what it did well (like finding recent data quickly) and what it struggled with (like backing up older numbers with proper sources). The evaluation step helped spot these issues easily, showing us where to improve the agent.

The next chapter is going to get even more interesting (plus, you have five solid use cases to look at!) as we explore different metrics to evaluate the AI agents across four dimensions: System Metrics, Task Completion, Quality Control, and Tool interaction.

# 04
# CHAPTER

# METRICS FOR EVALUATING AI AGENTS

# Metrics for Evaluating AI Agents

Before we explore metrics for evaluating AI, let's recall our key insights into agent evaluation. Using LLM-based judges (like GPT-4o) and robust metrics (such as context adherence), we effectively measured an agent's performance across various dimensions, including accuracy, speed, and cost efficiency. We then set up Galileo's evaluation callback to track and record the agent's performance.

This next chapter will explore various metrics for evaluating AI agents using five solid case studies.

Let's consider a document processing agent. While it might initially demonstrate strong performance metrics, we may have to probe into several questions:

- Is it maintaining optimal processing speeds and resource usage?
- How consistently does it complete assigned tasks without human intervention?
- Does it reliably adhere to specified formatting and accuracy requirements?
- Is it selecting and applying the most appropriate tools for each task?

Through a series of hypothetical case studies, we'll explore how organizations may transform their AI agents into reliable digital colleagues using key metrics. These examples will demonstrate practical approaches to:

- Improving task completion rates and reducing human oversight
- Enhancing output quality and consistency
- Maximizing effective tool utilization and selection

You should remember that the goal isn't perfection but establishing reliable, measurable, and continuously improving AI agents that deliver consistent value across all four key performance dimensions. See **Fig 4.1**


**Fig 4.1:** Four key performance dimensions to evaluate AI agents

# Case Study 1: Advancing the Claims Processing Agent

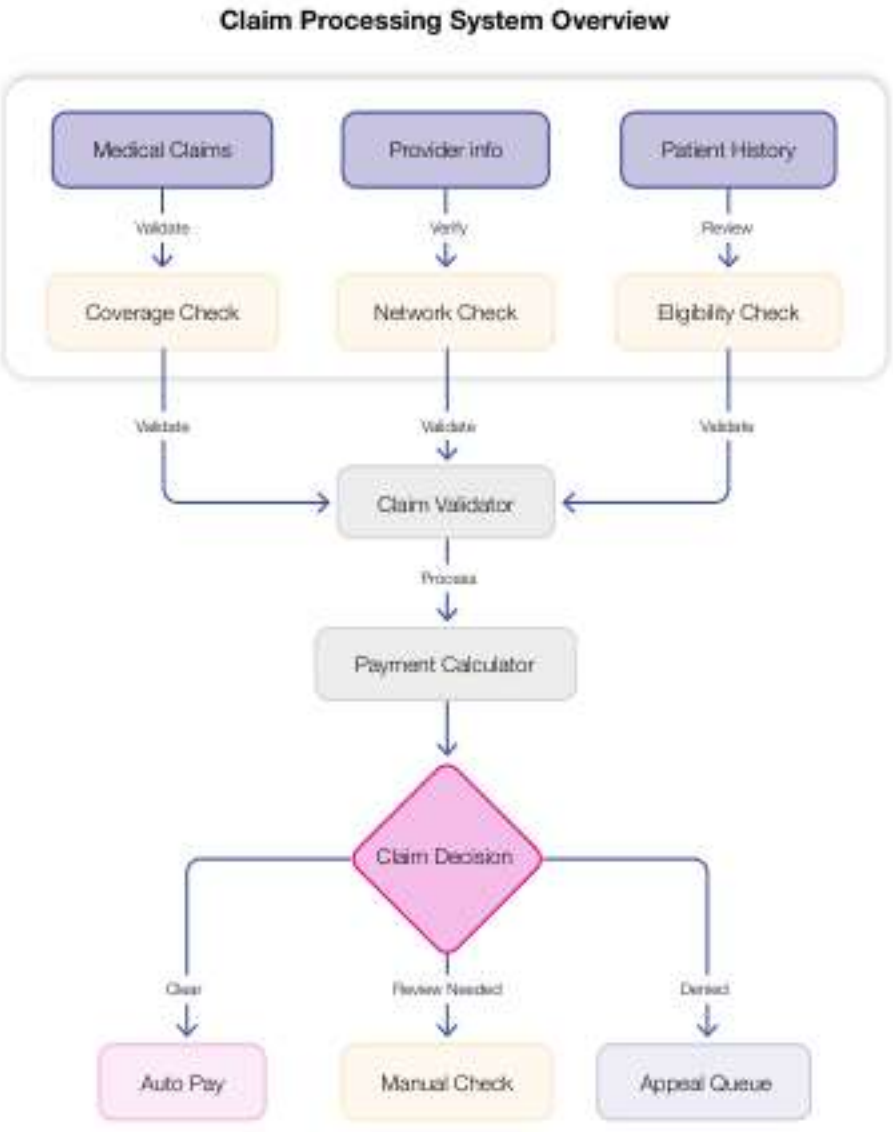


**Fig 4.2:** An overview of the Claims Processing System

A healthcare network implemented an AI agent to automate insurance claims processing, aiming to enhance efficiency and accuracy. However, this initiative inadvertently introduced compliance risks, highlighted by several key issues:

- The AI agent struggled with complex claims, leading to payment delays and provider frustration. Because of the inconsistency in handling these claims, claims processors spent more time verifying the AI's work than processing new claims.
- The error rate in complex cases raised alarms with the compliance team, especially critical given the stringent regulatory demands of healthcare claims processing.

## Functionality

The AI was designed to:

- Analyze medical codes
- Verify insurance coverage
- Check policy compliance
- Validate provider information
- Automatically assess claim completeness and compliance
- Calculate expected payments and generate preliminary approvals for straightforward claims

## Challenges

To counter these issues, the network focused on three key performance indicators to transform their AI agent's capabilities:

1. **LLM Call Error Rate**
   - **Problem:** API failures during claims analysis led to incomplete processing and incorrect approvals.
   - **Solution:** Implementing robust error recovery protocols and strict state management ensured accurate rollbacks and reprocessing.

2. **Task Completion Rate**
   - **Problem:** The agent incorrectly marked claims as 'complete' without conducting all necessary verifications.
   - **Solution:** Mandatory verification checklists and completion criteria were introduced to meet all regulatory requirements before finalizing claims.

3. **Number of Human Requests**
    - **Problem:** The agent took on complex cases beyond its capability, such as experimental procedures or cases requiring coordination of benefits across multiple policies.
    - **Solution:** Stricter escalation protocols automatically route high-risk cases to human experts based on claim complexity and regulatory requirements.

4. **Token Usage per Interaction**
    - **Problem:** Unnecessary inclusion of patient details in processing routine claims heightened privacy risks.
    - **Solution:** Strict data minimization protocols and context-cleaning practices were adopted to ensure that only essential protected health information is used

## Outcomes

The enhanced agent delivered:

- Faster claims processing
- Higher compliance accuracy
- Improved resource utilization
- Reduced rejection rates

# Case Study 2: Optimizing the Tax Audit Agent

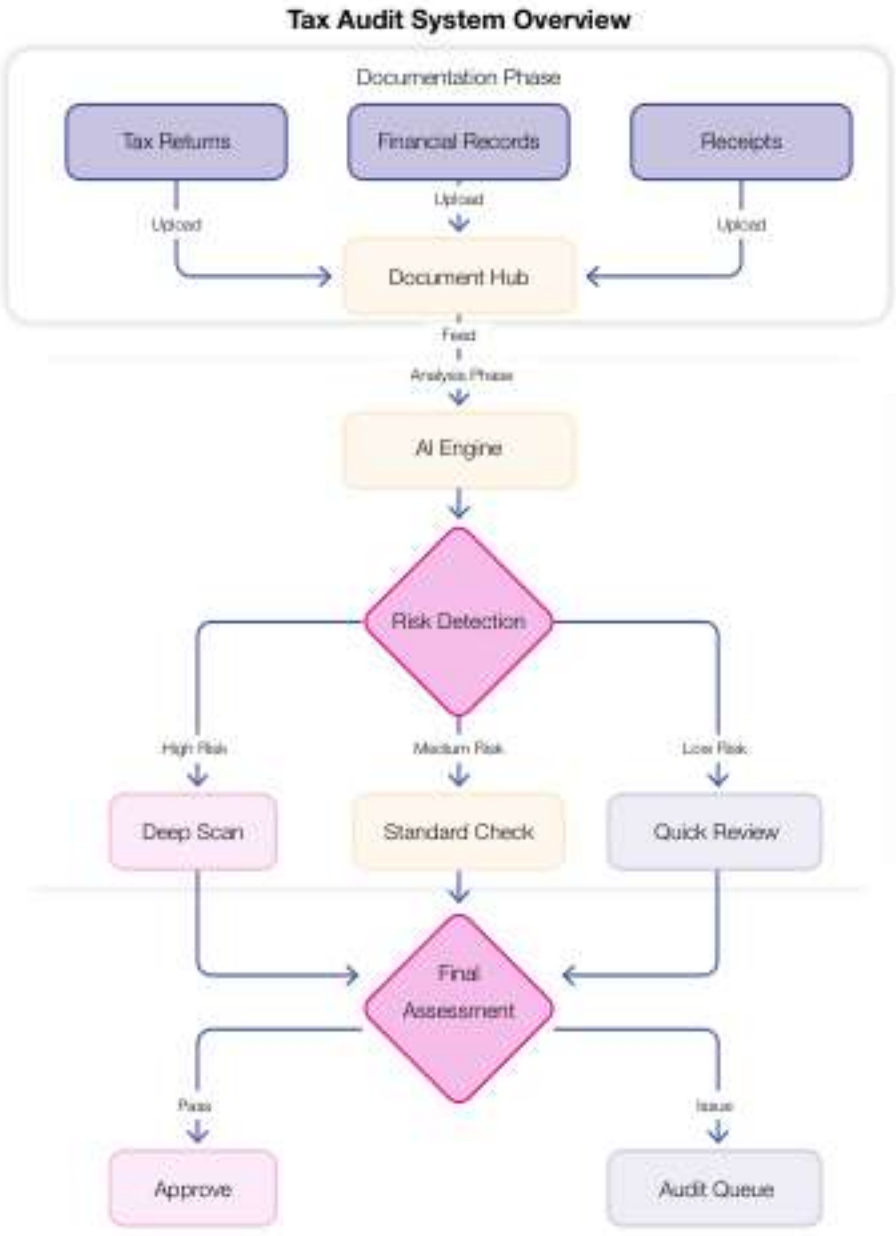


**Fig 4.3:** An overview of the Tax Auditing System

At a mid-sized accounting firm, their deployed AI audit agent created unexpected workflow bottlenecks. While the agent effectively handled routine tax document processing, the firm was concerned about three critical issues:

- Lengthy turnaround times for complex corporate audits
- Excessive computing costs from inefficient processing
- A growing backlog of partially completed audits requiring manual review

What should have streamlined their operations was instead causing senior auditors to spend more time supervising the AI's work than doing their specialized analysis. The firm needed to understand why its significant investment in AI wasn't delivering the anticipated productivity gains.

## Functionality

The AI audit agent was designed to:

- Process various tax documents, from basic expense receipts to complex corporate financial statements.
- Automatically extract and cross-reference key financial data in corporate tax returns.
- Systematically verify compliance across multiple tax years.
- Validate deduction claims against established rules and flag discrepancies for review.
- For simpler cases, it could generate preliminary audit findings and reports.
- The system was integrated with the firm's tax software and document management systems to access historical records and precedents.

## Challenges

The team focused on three critical metrics to reshape their agent's capabilities:

1. **Tool Success Rate**
   - **Problem:** The agent struggled with document processing efficiency, especially with complex document hierarchies.
   - **Solution:** Implementation of structured document classification protocols and validation frameworks improved handling of complex documents.

2. **Context Window Utilization**
   - **Problem:** The agent's processing of tax histories in their entirety was suboptimal, often missing connections between related transactions.

- **Solution:** Smart context segmentation was introduced, allowing the agent to focus on relevant time periods and maintain historical context. This enhanced the detection of subtle tax patterns.

3. **Steps per Task**
   - **Problem:** The agent applied the same level of analysis intensity to all tasks, regardless of complexity.
   - **Solution:** Adaptive workflows were implemented to adjust analytical depth based on the complexity of the task.

## Outcomes

The refined capabilities of the AI agent led to:

- Decreased audit completion times
- Improved accuracy in discrepancy detection
- More efficient utilization of processing resources

# Case Study 3: Elevating the Stock Analysis Agent

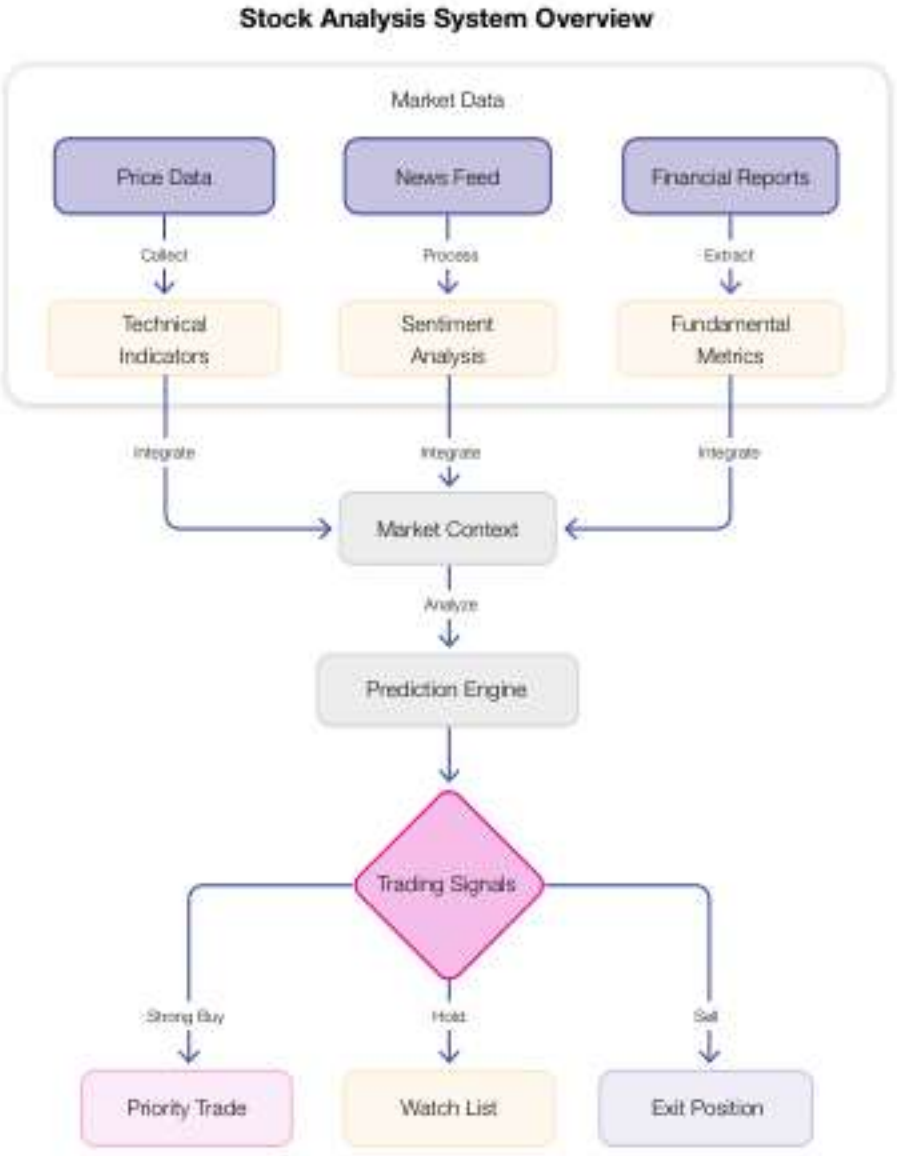


**Fig 4.4:** An overview of the Stock Analysis System

At a boutique investment firm, their AI-enhanced analysis service was under scrutiny as clients questioned its value. Portfolio managers were overwhelmed by redundant analysis requests and faced inconsistent reporting formats across client segments.

This situation undermined the firm's competitive edge of providing rapid market insights as analysts spent excessive time reformatting and verifying the AI's outputs. The inability of the AI to adjust its analysis depth based on varying market conditions resulted in either overly superficial or unnecessarily detailed reports, compromising client confidence in the service.

## Functionality

The AI analysis agent was developed to:

- Process multiple data streams, including market prices, company financials, news feeds, and analyst reports.
- Generate comprehensive stock analyses by evaluating technical indicators, assessing fundamental metrics, and identifying market trends across different timeframes.
- Generate customized reports combining quantitative data with qualitative insights for each analysis request.
- The system was integrated with the firm's trading platforms and research databases, providing real-time market intelligence.

## Challenges

Through analyzing three crucial metrics, the team improved the AI agent's performance:

1. **Total Task Completion Time**
   - **Problem:** The agent applied a uniform analysis depth across all stock types, regardless of their complexity.
   - **Solution:** Adaptive analysis frameworks based on stock characteristics were implemented to improve processing efficiency while maintaining insight quality.

2. **Output Format Success Rate**
   - **Problem:** Inconsistencies in how the agent presented market analysis for different user roles. Analysts and business managers received inappropriate levels of detail for their specific needs.

- **Solution:** Role-specific output templates and better parsing of output requirements were introduced, enabling the agent to format its analyses appropriately for different audiences while maintaining analytical accuracy.

3. **Token Usage per Interaction**
   - **Problem:** The agent inefficiently reprocessed entire documents for new queries, such as analyzing a company's quarterly earnings report multiple times for related questions.
   - **Solution:** Improved memory management and progressive analysis techniques were adopted, allowing the agent to reuse relevant insights across related queries while ensuring analytical precision.

## Outcomes

The enhancements to the AI agent delivered:

- More precise market analysis
- Faster processing times
- Improved resource utilization

# Case Study 4: Upgrading the Coding Agent

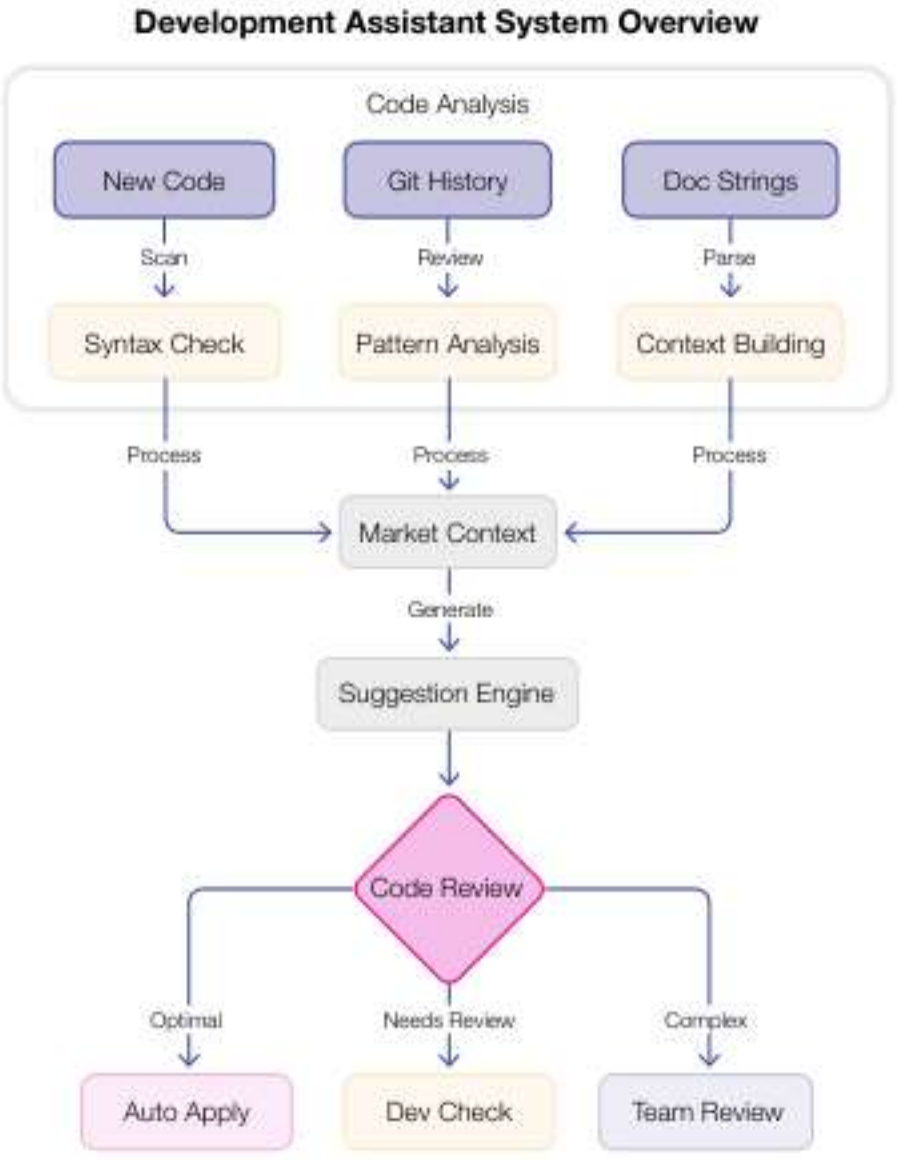


**Fig 4.5:** An overview of the Development Assistant System

A software development company implemented an AI coding assistant to enhance engineering productivity. However, rather than speeding up development cycles, the assistant became a source of frustration due to frequent disruptions and unreliable performance, especially during critical sprint deadlines.

Developers experienced delays as the agent struggled with large codebases and provided irrelevant suggestions that failed to consider project-specific requirements. Additionally, rising infrastructure costs from inefficient resource usage further exacerbated the situation, prompting a need for transformative improvements to make the AI assistant a genuine productivity tool.

## Functionality

The AI coding assistant was designed to:

- Analyze codebases to provide contextual suggestions, identify potential bugs, and recommend optimizations.
- Review code changes, ensuring compliance with project standards and generating documentation suggestions.
- Handle multiple programming languages and frameworks, adapting recommendations to specific project needs.
- The system integrated with common development tools and version control systems, supporting developers throughout the development cycle.

## Challenges

By optimizing three pivotal indicators, the team significantly enhanced the agent's capabilities:

1. **LLM Call Error Rate**
    - **Problem:** Frequent API timeouts when processing large code files and connection failures during peak usage.
    - **Solution:** Robust error handling, automatic retries, and request queuing mechanisms were implemented, greatly enhancing API call reliability and minimizing workflow disruptions.

2. **Task Success Rate**
    - **Problem:** Inconsistencies in the relevance and completeness of code suggestions. The agent sometimes provided overly complex rewrites for simple style fixes or inadequate details for required refactoring.
    - **Solution:** Standardized response templates for various code issues, including style guides, bug fixes, refactoring suggestions, and optimization recommendations, were introduced, making the agent's suggestions more consistently actionable.

3. **Cost per Task Completion**
    - **Problem:** Inefficient resource allocation in debugging workflows, using the same computational power for minor and major tasks.
    - **Solution:** Tiered processing was implemented based on the complexity and scope of code changes, optimizing resource usage while maintaining high analysis quality.

## Outcomes

The optimizations delivered:

- Enhanced code analysis accuracy
- Improved suggestion relevance
- More efficient resource utilization

# Case Study 5: Enhancing the Lead Scoring Agent

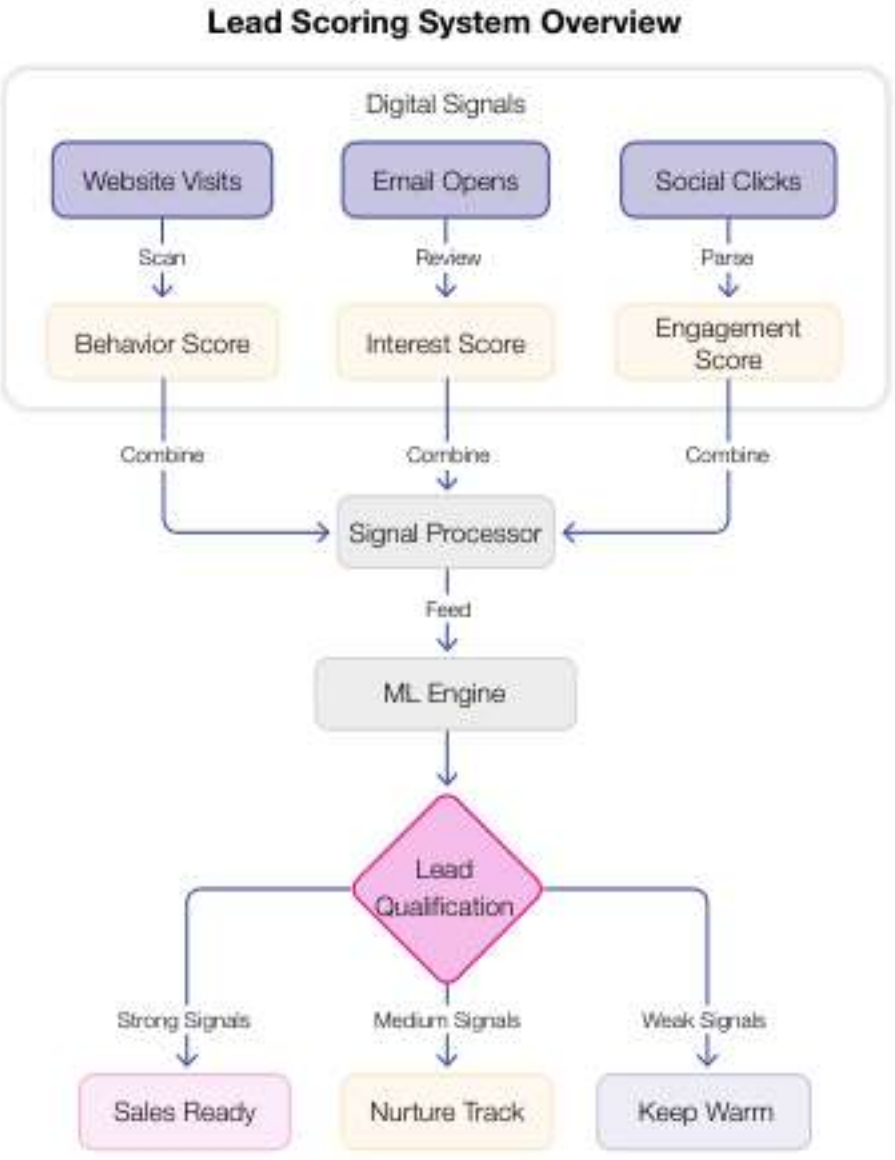


**Fig 4.6:** An overview of the Lead Scoring System

A software development company implemented an AI lead scoring agent to optimize sales strategies. Despite the promise of enhancing lead qualification efficiency, the agent was initially ineffective, leading to misclassification of prospects and declining conversion rates. Sales representatives found themselves pursuing low-potential leads due to outdated or inaccurate scores, especially during peak times, which resulted in increased costs per qualified lead and compromised growth targets.

## Functionality

- Evaluate data from multiple sources like website interactions, email responses, social media engagement, and CRM records to assess potential customers.
- Analyze company profiles, assess engagement patterns, and generate lead scores based on predefined criteria.
- Automatically categorize prospects by industry, company size, and potential deal value, updating scores in real-time as new information became available.
- Integrate with the company's sales tools, providing sales representatives with prioritized lead lists and engagement recommendations.

## Challenges

1. **Token Usage per Interaction**
   - **Problem:** The agent repetitively generated new analyses for similar company profiles instead of leveraging existing insights.
   - **Solution:** Implementation of intelligent pattern matching and context reuse improved processing efficiency while maintaining lead quality assessment accuracy.

2. **Latency per Tool Call**
   - **Problem:** Performance bottlenecks arose from sequential database querying patterns, causing delays.
   - **Solution:** Introduction of parallel processing and smart data caching transformed the agent's analysis speed.

3. **Tool Selection Accuracy**
   - **Problem:** The agent inefficiently selected between similar analysis methods, using more computationally expensive tools for basic tasks.

- **Solution:** Developing smarter selection criteria allowed the agent to match tool complexity with the analysis needs, using simpler tools for straightforward tasks and reserving intensive tools for complex cases.

## Outcomes

- Faster prospect analysis processing
- Higher lead qualification accuracy
- Improved resource utilization efficiency

These use cases reveal a crucial truth: **effective AI agents require careful measurement and continuous optimization.** As these systems become more sophisticated, the ability to measure and improve their performance becomes increasingly important.

**Here's a quick takeaway:**

- Metric-driven optimization must align with business objectives
- Human workforce transformation is crucial for AI success
- Clear outcome targets drive better optimization decisions
- Regular measurement and adjustment cycles are essential
- Balance between automation and human oversight is critical

# 05
# CHAPTER

# WHY MOST AI AGENTS FAIL & HOW TO FIX THEM

# CHAPTER 5
# WHY MOST AI AGENTS FAIL & HOW TO FIX THEM

In the previous chapter, we looked at different metrics for evaluating our AI agents, namely along four core dimensions: Technical efficiency, Task Completion, Quality Control, and Tool interaction. In our journey, we've also seen how agents are powerful tools capable of automating complex tasks and processes with many frameworks that make it possible to build complex agents in a few lines of code. However, many AI agents fail to deliver the expected outcomes despite their potential.

In this chapter, we'll examine why agents fail, providing insights into common pitfalls and strategies to overcome them.

# AI Agent Challenges and Solutions

## DEVELOPMENT ISSUES

Poorly Defined Prompts
- Define Clear Objectives
- Craft Detailed Personas
- Use Effective Prompting

Evaluation Challenges
- Continuous Evaluation
- Use Real-World Scenarios
- Incorporate Feedback Loops

## LLM ISSUES

Difficult to Steer
- Specialized Prompts
- Hierarchical Design
- Fine-Tuning Models

High Cost of Running
- Reduce Context Size
- Use Smaller Models
- Cloud-Based Solutions

Planning Failures
- Task Decomposition
- Multi-Plan Selection
- Reflection and Refinement

Reasoning Failures
- Enhance Reasoning Capabilities
- Fine-Tune LLMs with Feedback
- Use Specialized Agents

Tool Calling Failures
- Define Clear Parameters
- Validate Tool Outputs
- Tool Selectio VerificationLoops

## PRODUCTION ISSUES

Guardrails
- Rule-Based Filters & Validation
- Human-in-the-Loop Oversight
- Ethical & Compliance Frameworks

Agent Scaling
- Scalable Architectures
- Resource Management
- Monitor Performance

Fault Tolerance
- Redundancy
- Automated Recovery
- Stateful Recovery

Infinite Looping
- Clear Termination Conditions
- Enhance Reasoning & Planning
- Monitor Agent Behavior

# Development Issues

## Poorly Defined Task or Persona

A well-defined task or persona is essential for effectively operating your AI agents. It clarifies the agent›s objectives, constraints, and expected outcomes, ensuring that your agent can make appropriate decisions and perform effectively. Without it, agents may struggle to make appropriate decisions, leading to suboptimal performance.

**Define Clear Objectives**

You should specify the goals, constraints, and expected outcomes for each agent.

**Craft Detailed Personas**

Develop personas that outline the agents role, responsibilities, and behavior for you.

**Prompting**

Use research-backed prompting techniques to reduce hallucinations for your agents.

## Evaluation Issues

Evaluation helps you identify weaknesses and ensures your agents operate reliably in dynamic environments. However, evaluating agents› performance is inherently challenging. Unlike traditional software, where outputs can be easily validated against expected results, agents operate in dynamic environments with complex interactions, making it difficult for you to establish clear metrics for success.

**Continuous Evaluation**

Implement an ongoing evaluation system to assess your agents performance and identify areas for improvement.

**Use Real-World Scenarios**

Test your agents in real-world scenarios to understand their performance in dynamic environments.

**Feedback Loops**

Incorporate feedback loops to allow for continuous improvement based on performance data.

# LLM Issues

## Difficult to Steer

You can steer LLMs towards specific tasks or goals for consistent and reliable performance. Effective steering ensures that agents can perform their intended functions accurately and efficiently. LLMs are influenced by vast amounts of training data, which can lead to unpredictable behavior, and fine-tuning them for specific tasks requires significant expertise and computational resources.

### Specialized Prompts

Use specialized prompts to guide the LLM toward specific tasks.

### Hierarchical Design

Implement a hierarchical design where specialized agents handle specific tasks, reducing the complexity of steering a single agent. **(See Fig 5.1)**

### Fine-Tuning

Continuously fine-tune the LLM based on task-specific data to improve performance.


**Fig 5.1:** Hierarchical design with specialized agents performing specific tasks

## High Cost of Running

Running LLMs, especially in production environments, can be prohibitively expensive. The computational resources required for inference, particularly for large models, can lead to high operational costs. This makes it difficult for organizations to scale their agent deployments cost-effectively.

### Reduce Context

Agents can run for a while in their iterative loops. Introduce mechanisms to use as low context as possible to reduce the tokens.

### Use Smaller Models

Where possible, use smaller models or distill larger models to reduce costs.

### Cloud Solutions

Use cloud-based solutions to manage and scale computational resources efficiently. Design a serverless system to save wasting of resources. **(See Fig 5.2.)**

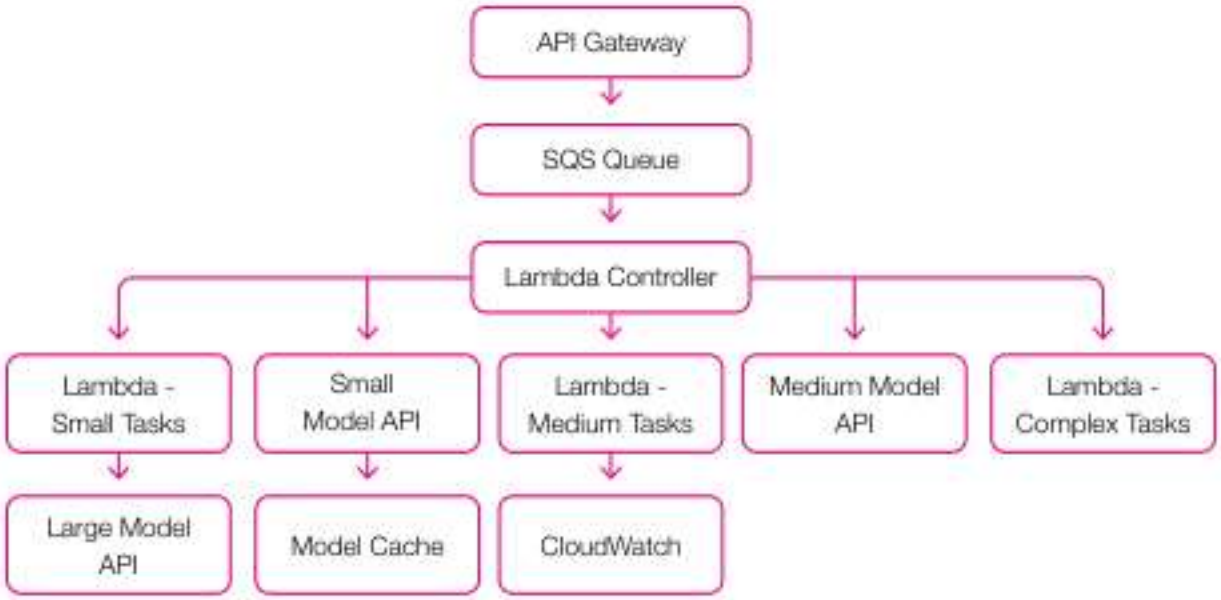


**Fig 5.2:** A serverless architecture where Lambda Controller makes intelligent decisions about request handling

Components of **Fig 5.2**

- The **SQS Queue** acts as our request buffer.
- The **Lambda Controller** makes intelligent decisions about request handling.
- **Small Model API** for simple completions and basic tasks
- **Medium Model API** for moderate complexity tasks
- **Large Model API** for complex reasoning tasks
- **Model Cache** for storing frequently used responses to reduce API calls
- **CloudWatch** to monitor system health and costs

# Planning Failures

Effective planning is crucial for agents to perform complex tasks. Planning enables agents to anticipate future states, make informed decisions, and execute tasks in a structured manner. Without effective planning, agents may struggle to achieve desired outcomes. However, LLMs often struggle with planning, as it requires strong reasoning abilities and the ability to anticipate future states.

### Task Decomposition

Break down tasks into smaller, manageable subtasks.

### Multi-Plan Selection

Generate multiple plans and select the most appropriate one based on the context.

### Reflection and Refinement

Continuously refine plans based on new information and feedback.and scale computational resources efficiently. Design a serverless system to save wasting of resources. **(See Fig 5.2.)**

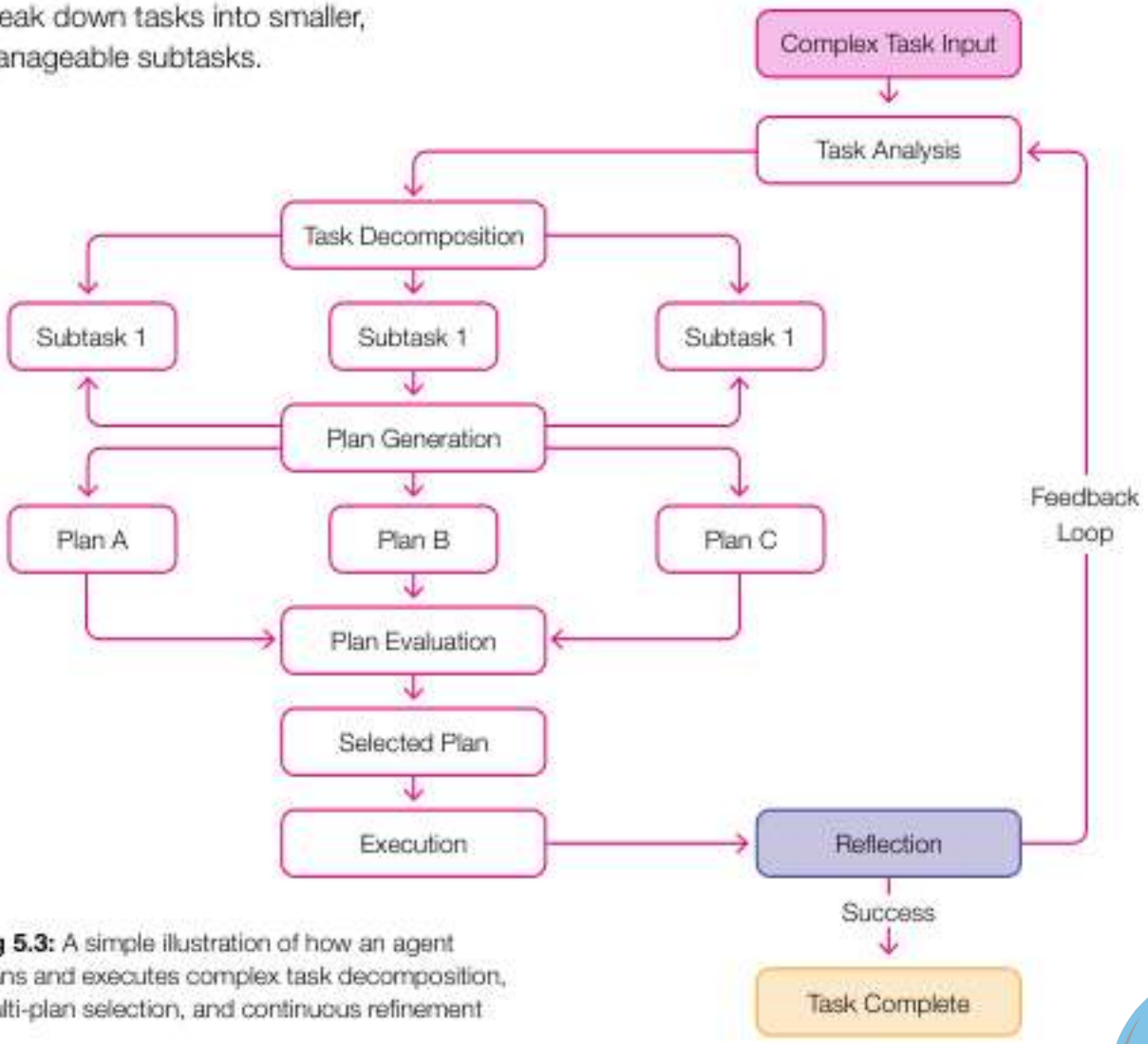


**Fig 5.3:** A simple illustration of how an agent plans and executes complex task decomposition, multi-plan selection, and continuous refinement

## Reasoning Failures

Reasoning is a fundamental capability that enables agents to make decisions, solve problems, and understand complex environments. Strong reasoning skills are essential for agents to interact effectively with complex environments and achieve desired outcomes. LLMs lacking strong reasoning skills may struggle with tasks that require multi-step logic or nuanced judgment. **(See Fig 5.4)**

### Enhance Reasoning Capabilities

Use prompting techniques like Reflexion to enhance the reasoning capabilities. Incorporate external reasoning modules that can assist the agent in complex decision-making processes. These modules can include specialized algorithms for logical reasoning, probabilistic inference, or symbolic computation.

### Finetune LLM

Establish training with data generated with a human in the loop. Feedback loops allow the agent to learn from its mistakes and refine its reasoning over time. You can use data with traces of reasoning that teach the model to reason or plan in various scenarios.

### Use Specialized Agents

Develop specialized agents that focus on specific reasoning tasks to improve overall performance.

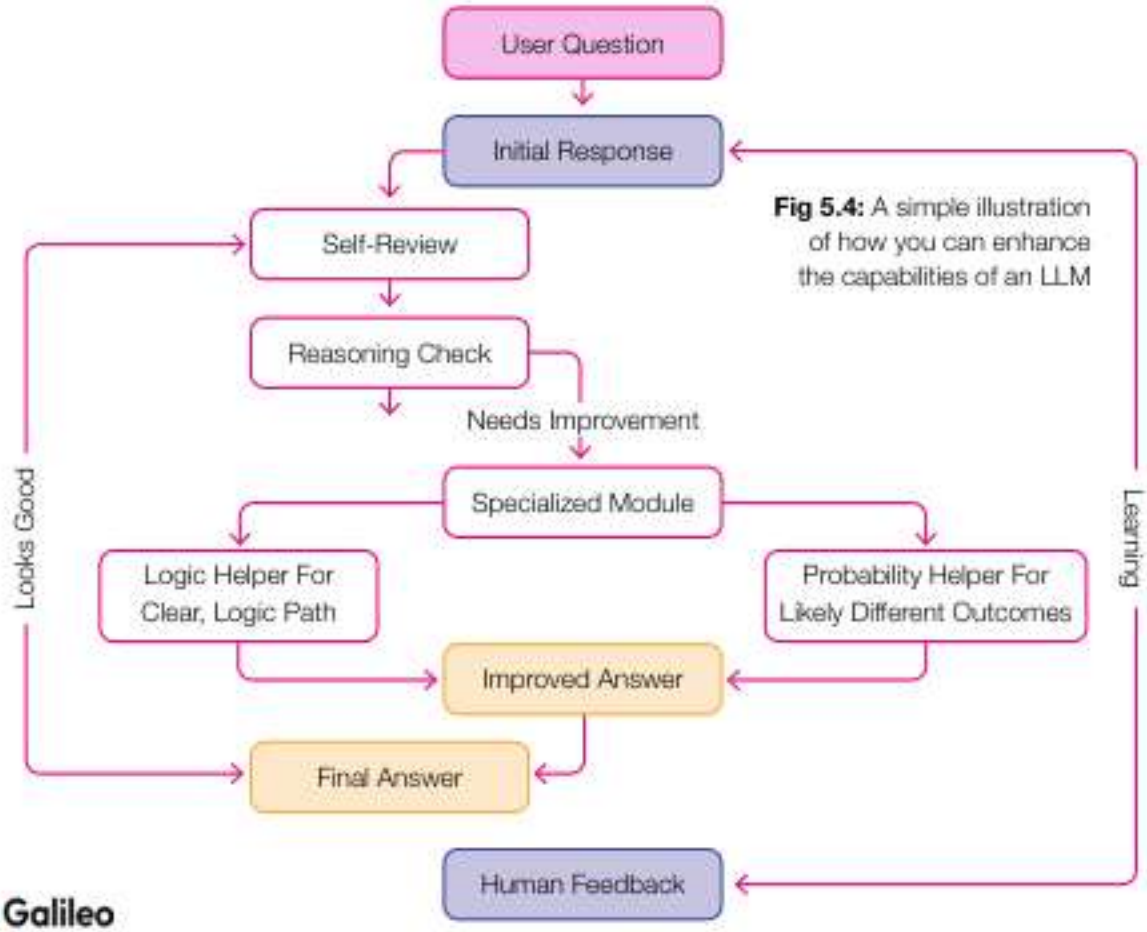


**Fig 5.4:** A simple illustration of how you can enhance the capabilities of an LLM

# Tool Calling Failures

One key benefit of agent abstraction over prompting base language models is the ability to solve complex problems by calling multiple tools to interact with external systems and data sources. Robust tool calling mechanisms ensure agents can perform complex tasks by leveraging various tools accurately and efficiently. However, agents often face challenges in effectively calling and using these tools. Tool calling failures can occur due to incorrect parameter passing, misinterpretation of tool outputs, or failures in integrating tool results into the agent's workflow.

**Define Clear Parameters**

Ensure that tools have well-defined parameters and usage guidelines for you.

**Validate Tool Outputs**

Implement validation checks to ensure that tool outputs are accurate and relevant.

**Tool Selection Verification**

Use a verification layer to check if the tool selected is correct for the job.

# Production Issues

## Guardrails

Guardrails help ensure that agents adhere to safety protocols and regulatory requirements. This is particularly important in sensitive domains such as healthcare, finance, and legal services, where non-compliance can have severe consequences. Guardrails define the operational limits within which agents can function.

Implement rule-based filters and validation mechanisms to monitor and control the actions and outputs of AI agents.

**Content Filters**

Use predefined rules to filter inappropriate, offensive, or harmful content. For example, content filters can scan the agent's outputs for prohibited words or phrases and block or modify responses that contain such content.

**Input Validation**

Before processing, inputs received by the agent must be validated to ensure they meet specific criteria. This can prevent malicious or malformed inputs from causing unintended behavior.

**Action Constraints**

Define constraints on the actions that agents can perform. For example, an agent managing financial transactions should have rules that prevent it from initiating transactions above a certain threshold without additional authorization.

Incorporate human-in-the-loop mechanisms to provide oversight and intervention capabilities.

**Approval Workflows:**
Implement workflows where certain actions or outputs require human approval before execution. For example, an agent generating legal documents can have its drafts reviewed by a human expert before finalization.

**Feedback Loops:**
Allow humans to provide feedback on the agent's performance and outputs. You can use this feedback to refine the agent's behavior and improve future interactions.

**Escalation Protocols:**
Establish protocols for escalating complex or sensitive tasks to human operators. For example, if an agent encounters a situation it cannot handle, it can escalate the issue to a human supervisor for resolution.

Develop and enforce ethical and compliance frameworks to guide the behavior of AI agents.

**Ethical Guidelines:**
Establish ethical guidelines that outline the principles and values the agent must adhere to. These guidelines can cover areas such as fairness, transparency, and accountability.

**Compliance Checks:**
Implement compliance checks to ensure that the agent's actions and outputs align with regulatory requirements and organizational policies. For example, an agent handling personal data must comply with data protection regulations such as GDPR.

**Audit Trails:**
Maintain audit trails that record the agent's actions and decisions. This allows for retrospective analysis and accountability, ensuring that any deviations from ethical or compliance standards can be identified and addressed.

# Agent Scaling

Scaling agents to handle increased workloads or more complex tasks is a significant challenge. As the number of agents or the complexity of interactions grows, the system must efficiently manage resources, maintain performance, and ensure reliability.

### Scalable Architectures

Design architectures that can efficiently manage increased workloads and complexity. Implement a microservices architecture where each agent or group of agents operates as an independent service. This allows for easier scaling and management of individual components without affecting the entire system.

### Resource Management

Integrate load balancers to distribute incoming requests evenly across multiple agents. This prevents any single agent service from becoming overwhelmed and ensures a more efficient use of resources.

### Monitor Performance

Implement real-time monitoring tools to track each agent's performance. Metrics such as response time, resource utilization, and error rates should be continuously monitored to identify potential issues. **(See Fig 5.5)**

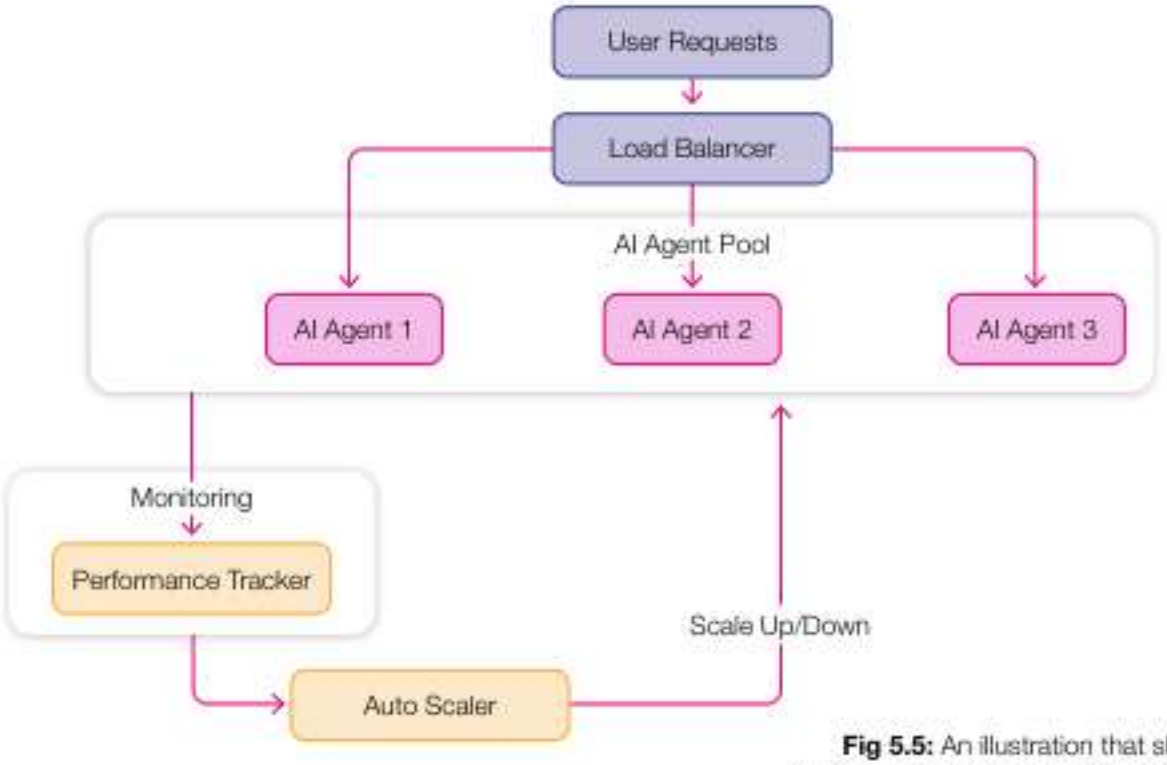


**Fig 5.5:** An illustration that shows how you can add monitoring and load balancers for easy scale-up and down

# Fault Tolerance

AI agents need to be fault-tolerant to ensure that they can recover from errors and continue operating effectively. Without robust fault tolerance mechanisms, agents may fail to handle unexpected situations, leading to system crashes or degraded performance. **(See Fig 5.6)**

### Redundancy

Deploy multiple instances of AI agents running in parallel. If one instance fails, the other instances can continue processing requests without interruption. This approach ensures high availability and minimizes downtime.

### Automated Recovery

Incorporate intelligent retry mechanisms that automatically attempt to recover from transient errors. This includes exponential backoff strategies, where the retry interval increases progressively after each failed attempt, reducing the risk of overwhelming the system. Develop self-healing mechanisms that automatically restart or replace failed agent instances.

### Stateful Recovery

Ensure that AI agents can recover their state after a failure. This involves using persistent storage to save the agent's state and context, allowing it to resume operations from the last known good state after a restart.

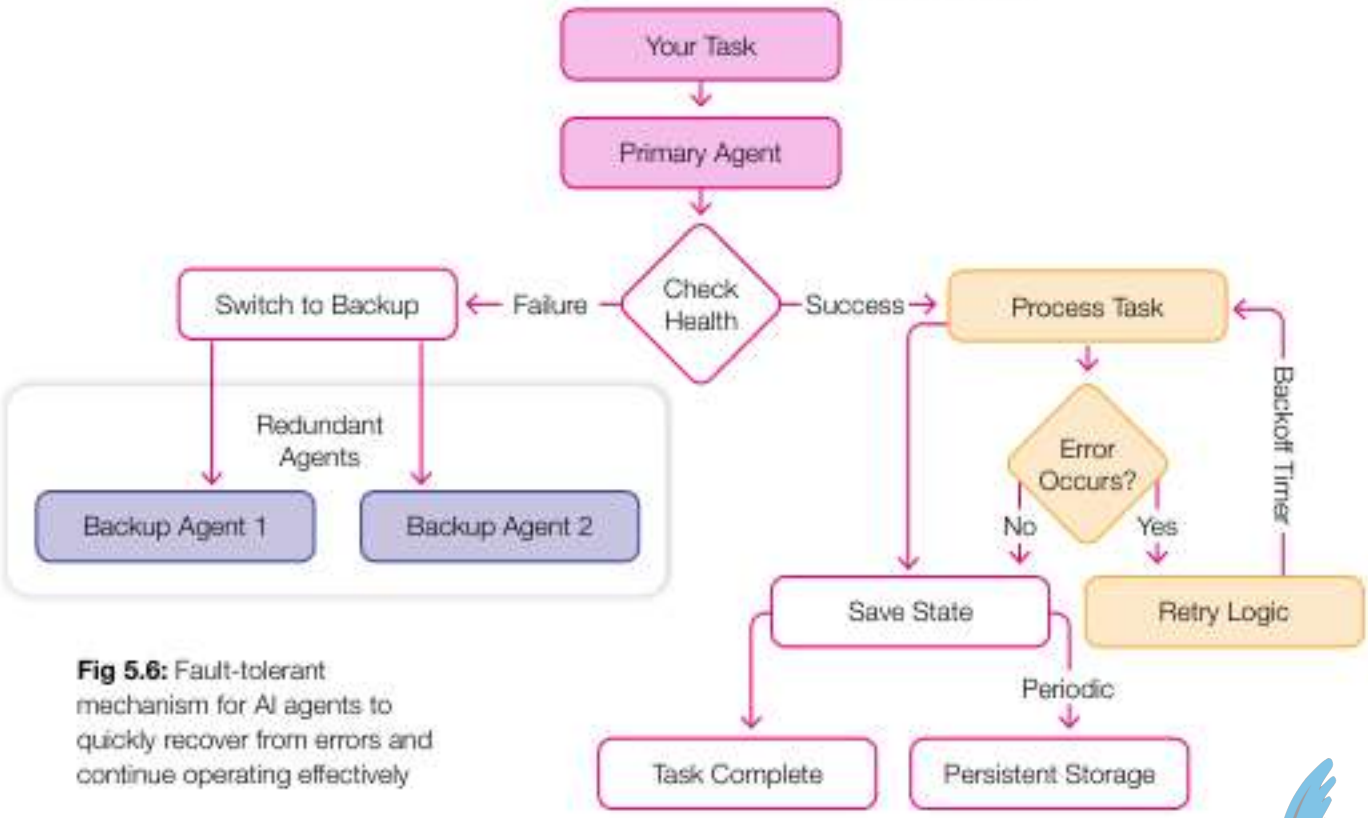


**Fig 5.6:** Fault-tolerant mechanism for AI agents to quickly recover from errors and continue operating effectively

# Infinite Looping

Looping mechanisms are essential for agents to perform iterative tasks and refine their actions based on feedback. Agents can sometimes get stuck in loops, repeatedly performing the same actions without progressing toward their goals. **(See Fig 5.7)**

### Clear Termination Conditions

Implement clear criteria for success and mechanisms to break out of loops.

### Enhance Reasoning and Planning

Improve the agent's reasoning and planning capabilities to prevent infinite looping.

### Monitor Agent Behavior

Monitor agent behavior and adjust to prevent looping issues.

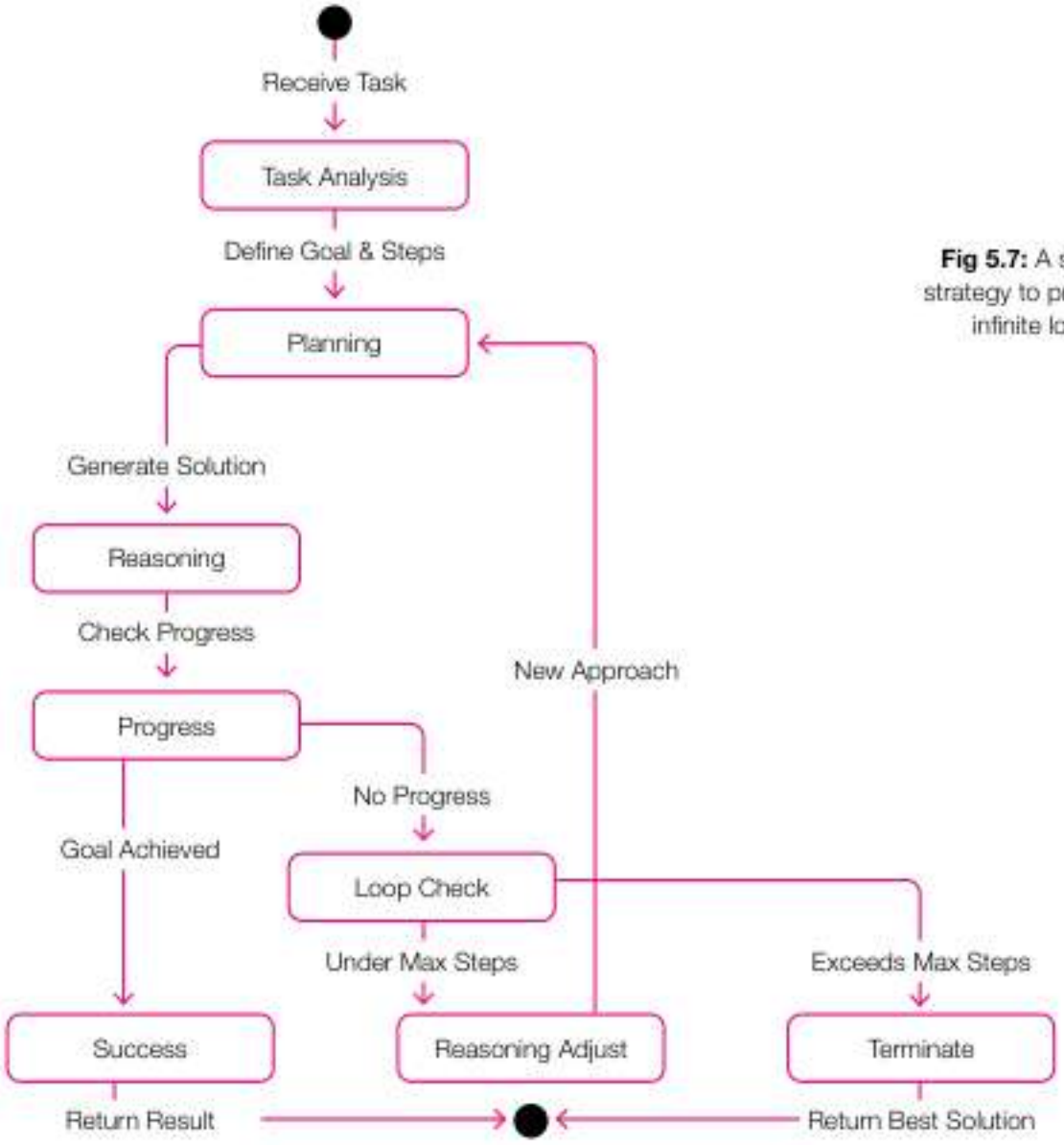


**Fig 5.7:** A simple strategy to prevent infinite looping

Through the above examples and workflow diagrams **(Fig 5.1** to **Fig 5.6**), you'll notice that while building AI agents presents numerous challenges, understanding and addressing these common failure points is necessary for success.

By implementing proper guardrails, ensuring robust error handling, and designing scalable architectures, you can create agents that work reliably and provide real value in production environments.

That said, remember that building effective agents is an iterative process.

Always start small, test thoroughly, and gradually expand your agent's capabilities as you learn from real-world usage. Pay special attention to the fundamentals we've covered—from clear task definition and evaluation to proper planning and reasoning capabilities. This will help you establish a strong foundation when you begin to experiment with your AI agents.

# Glossary

| Term | Description |
|---|---|
| Large Language Model (LLM) | An advanced AI model that can understand and generate human-like text by predicting the next word in a sequence. |
| AI Agent | Software application powered by large language models that autonomously perform specific tasks and makes complex decisions. |
| Agent-Based System | An approach where software agents work independently to solve problems through decision-making and interactions. |
| Task Automation | The process of using AI to perform repetitive or complex tasks without human intervention. |
| System Latency | The time delay between when an AI agent receives input and when it provides a response. |
| Entity Memory | A specialized form of AI memory that maintains detailed information about specific entities (people, organizations, concepts) across multiple interactions. |
| Human-in-the-Loop (HITL) | A system design approach that integrates human oversight and intervention points within automated AI processes. |
| Multi-Agent Pattern | A structured approach to organizing multiple AI agents› interactions, including hierarchical, sequential, and dynamic patterns. |
| Role-Based Agent Design | An architectural approach where AI agents are assigned specific roles with defined responsibilities, tools, and interaction patterns within a larger system. |
| State Management | Systematically tracking and controlling an AI agent›s internal conditions, memory, and context throughout its operation cycle. |
| Context Window Utilization | A metric measuring how efficiently an AI agent uses its available processing capacity for analyzing and retaining information |
| LLM Call Error Rate | A critical reliability metric tracking the frequency of failed API requests and processing errors when an AI agent interacts with its underlying language model. |
| Latency per Tool Call | A performance indicator measuring the time delay between an AI agent›s request to use a specific tool and receiving the tool›s response. |

| | |
|---|---|
| Output Format Success Rate | A quality metric assessing how accurately an AI agent adheres to specified formatting requirements and presentation standards across different user roles and contexts. |
| Steps per Task | An efficiency metric tracking the number of discrete operations an AI agent requires to complete a given task. |
| Task Completion Rate | A comprehensive performance indicator measuring the percentage of assignments an AI agent successfully completes without human intervention. |
| Tool Selection Accuracy | A metric evaluating how appropriately an AI agent chooses specific tools or methods from its available toolkit based on task requirements and complexity. |
| Token Usage per Interaction | A resource efficiency metric tracking how many computational units (tokens) an AI agent consumes during task processing |
| Hierarchical Design | A system architecture where specialized AI agents handle specific tasks, reducing the complexity of steering a single agent. |
| Prompting Techniques | Research-backed methods to guide LLM behavior and reduce hallucinations in AI agents. |
| Reflexion | A specialized prompting technique that enhances an AI agent's reasoning capabilities through self-reflection and improvement. |
| Serverless Architecture | A cloud-based system design where computational resources are dynamically allocated based on AI agent workload demands. |
| Task Decomposition | The process of breaking down complex assignments into smaller, manageable subtasks for AI agents to handle effectively. |
| Tool Calling | The mechanism by which AI agents interact with external systems and data sources to solve complex problems through multiple tool interactions. |