Capítulo 1

Agentes de IA: Construção de Chatbot no Paradigma Multiagente com LangChain e LangGraph

Gutemberg P. A. da Silva, Jailson C. Zarur, Luis H. M. Queiroz, Raimundo S. Moura

Abstract

This chapter presents a theoretical and practical approach to building intelligent agents using Large Language Models (LLMs). Bridging the gap between classical AI theory and modern frameworks, we map fundamental concepts defined by Russell and Norvig to implementations in LangChain and LangGraph. We explore architectures ranging from simple reflex agents to multi-agent systems with state persistence, demonstrating how to orchestrate complex workflows in production environments.

Resumo

Este capítulo apresenta uma abordagem teórico-prática para a construção de agentes inteligentes utilizando Grandes Modelos de Linguagem (LLMs). Estabelecendo uma ponte entre a teoria clássica de IA e frameworks modernos, mapeamos conceitos fundamentais definidos por Russell e Norvig para implementações em LangChain e LangGraph. Exploramos arquiteturas que variam de agentes reativos simples a sistemas multiagentes com persistência de estado, demonstrando como orquestrar fluxos de trabalho complexos em ambientes de produção.

1.1. Introdução e Motivações

A engenharia de sistemas modernos atravessa uma mudança de paradigma impulsionada pela integração massiva de Grandes Modelos de Linguagem (LLMs). Estes modelos deixaram de ser apenas ferramentas de processamento de texto para se tornarem componentes centrais em arquiteturas complexas de software. Nesse cenário, frameworks de orquestração como LangChain[1] e LangGraph[2] emergem como a infraestrutura essencial,

[1]https://docs.langchain.com/

[2]https://docs.langchain.com/langgraph/

fornecendo as fundações arquiteturais para construir e implantar sistemas inteligentes [1]. A adoção de agentes autônomos neste contexto permite que os desenvolvedores aproveitem essa abundância de capacidade cognitiva, transformando LLMs em entidades ativas capazes de realizar planejamento dinâmico e coordenar fluxos de trabalho complexos, superando as limitações de scripts estáticos tradicionais [5].

A teoria de agentes inteligentes, formalizada extensivamente na literatura clássica de Inteligência Artificial, define um agente como uma entidade que percebe seu ambiente através de sensores e age sobre ele através de atuadores [4]. Durante décadas, a implementação destes agentes exigiu a codificação manual de regras ou o treinamento de modelos específicos para tarefas restritas. No entanto, o advento dos LLMs alterou fundamentalmente este paradigma. Na arquitetura de agentes modernos, consolida-se o entendimento de que o LLM atua como o *controlador central* ou "cérebro" do agente, possuindo capacidades de raciocínio generalista e conhecimento paramétrico prévio para orquestrar os demais módulos do sistema [5].

Apesar da disponibilidade de ferramentas poderosas, existe uma desconexão frequente entre os fundamentos teóricos da IA e a prática de desenvolvimento de software. Desenvolvedores muitas vezes criam "chains" (cadeias de execução) sem compreender que estão, na verdade, implementando agentes reativos simples, ou utilizam grafos de estado sem perceber a correlação com agentes baseados em modelo.

Este capítulo visa preencher essa lacuna, consolidando o conteúdo teórico-prático do minicurso. Não trataremos apenas da sintaxe das ferramentas, mas de como elas materializam conceitos clássicos de racionalidade, planejamento e sistemas multiagentes. Ao longo das próximas seções, demonstraremos como a função matemática abstrata de um agente se traduz em código Python moderno, permitindo a construção de *chatbots* que não apenas conversam, mas executam tarefas complexas de forma robusta e auditável.

1.2. Fundamentos Teóricos e Mapeamento Moderno

Para construir sistemas robustos com LangChain e LangGraph, é imperativo revisitar a definição formal de um agente. Segundo Russell e Norvig [4], o comportamento de um agente é descrito pela **função do agente** (f), que mapeia uma sequência completa de percepções (P^*) para uma ação (A):

$$f : P^* \to A \tag{1}$$

No contexto de desenvolvimento moderno com LLMs, podemos realizar um mapeamento direto destes componentes teóricos para artefatos de software:

- **Sensores (P):** Correspondem às interfaces de entrada do sistema. Em uma aplicação baseada em LangChain, a percepção não é apenas o *prompt* do usuário, mas inclui o histórico da conversa (`chat_history`), o contexto injetado via *Retrieval-Augmented Generation* (RAG) e os resultados de execuções anteriores de ferramentas.

- **Função do Agente (f):** É implementada pelo próprio LLM (como GPT-4, Gemini

ou Llama 3). O modelo atua como o motor de raciocínio, processando a sequência de percepções para determinar a próxima etapa.

- **Atuadores (*A*):** Tradicionalmente vistos como mecanismos físicos em robótica, no desenvolvimento de software eles se manifestam através do conceito de *Tool Calling*. Ferramentas são funções Python, chamadas de API ou consultas a bancos de dados que permitem ao agente alterar o estado do ambiente, expandindo seu espaço de ação para além da geração de texto [5, 1].

1.2.1. Racionalidade versus Onisciência

Um ponto crucial para a implementação de agentes eficazes é a distinção entre racionalidade e onisciência. Um agente onisciente saberia o resultado real de suas ações, o que é impossível na realidade [4]. Um agente racional, por sua vez, escolhe a ação que maximiza o desempenho esperado, dado o seu conhecimento atual.

Os LLMs não são oniscientes; eles sofrem de alucinações e possuem conhecimento desatualizado (o *cutoff* de treinamento) [5], o que exige mecanismos de grounding externo para garantir a factibilidade das respostas. Portanto, ao projetar um sistema com LangGraph, devemos focar na **racionalidade**: desenhar o fluxo para que o agente reconheça a insuficiência de suas informações internas e utilize ferramentas (atuadores) para buscar dados externos antes de responder, maximizando assim a utilidade da resposta.

1.2.2. Taxonomia de Arquiteturas: De Cadeias a Grafos

A evolução das ferramentas de orquestração reflete a hierarquia de complexidade dos agentes. A seguir, correlacionamos os tipos de agentes clássicos com os padrões de projeto utilizados neste minicurso.

1.2.2.1. Agentes Reativos Simples e Chains

Os agentes reativos simples selecionam ações com base apenas na percepção atual, ignorando o histórico de percepções [4]. Eles operam sob regras de condição-ação diretas. No ecossistema LangChain, isso equivale a uma **Chain** básica (como a `RunnableSequence`). O sistema recebe uma entrada, processa-a através de um *prompt template* e um LLM, e gera uma saída. Não há persistência de estado complexo nem loops de decisão. É um fluxo linear e funcional, ideal para tarefas atômicas como classificação de sentimento ou tradução, mas insuficiente para diálogos longos e complexos.

1.2.2.2. Agentes Baseados em Modelo e Grafos de Estado

Para operar em ambientes parcialmente observáveis e complexos, o agente precisa manter um **estado interno** que depende do histórico de percepções e reflete como o mundo evolui independentemente do agente. Aqui reside a necessidade do **LangGraph**. Diferente de uma *chain* linear, o LangGraph introduz o conceito de **State** (um objeto de estado compartilhado e persistente). Isso permite a construção de grafos cíclicos onde o agente pode:

1. Raciocinar (“Thought”);
2. Agir (“Action”);
3. Observar o resultado (“Observation”);
4. Atualizar seu estado interno;
5. Decidir se precisa agir novamente ou responder ao usuário.

Esta estrutura cíclica implementa o padrão **ReAct** [6], que materializa o ciclo de monitoramento e correção descrito na literatura de planejamento clássico. Essa dissonância entre as abordagens é ilustrada na Figura 1.1.

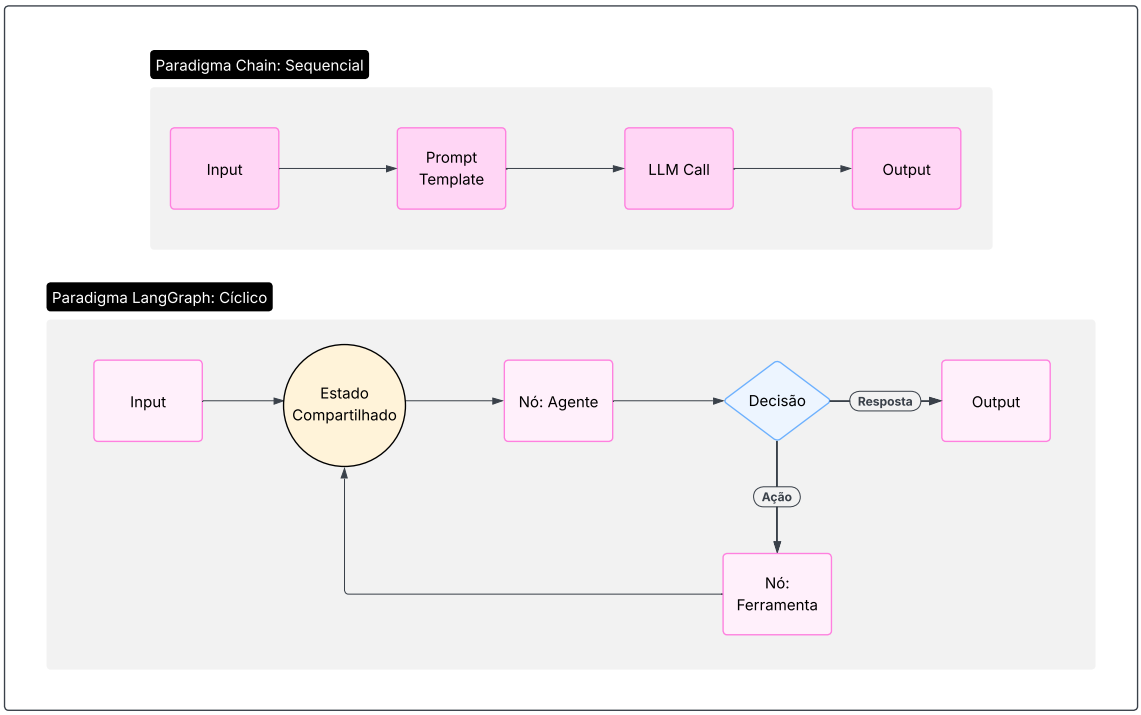

Figura 1.1: Comparação entre Agente Reativo Simples (Chain) e Agente Baseado em Modelo (Grafo)

1.2.3. O Problema da Persistência e Redutores

Um dos desafios centrais na IA clássica é o problema da representação e persistência: como descrever o que muda no mundo sem ter que listar tudo o que permaneceu inalterado? Na linguagem de planejamento PDDL (*Planning Domain Description Language*), isso é resolvido especificando apenas os efeitos de adição e remoção de uma ação [2].

O LangGraph adota uma abordagem análoga para o gerenciamento de memória através de **Redutores** (*Reducers*). Ao definirmos o estado do agente, utilizamos operações (como `operator.add`) que especificam como novas informações (ex: uma nova mensagem) devem ser mescladas ao estado existente. O framework garante que o histórico anterior persista, aplicando apenas as “deltas” (mudanças) necessárias. Isso garante a

integridade do contexto conversacional e permite funcionalidades avançadas como *Time Travel* (reverter o estado do agente para um ponto anterior) e *Human-in-the-loop* (intervenção humana antes da execução de uma ação crítica).

1.3. Implementação de Agentes com LangChain e LangGraph

Tendo estabelecido a fundamentação teórica de que um agente é uma função que mapeia um histórico de percepções para ações em um ambiente incerto, voltamo-nos agora para a implementação computacional destes conceitos. A biblioteca **LangChain** fornece as primitivas de abstração para interagir com Grandes Modelos de Linguagem (LLMs), enquanto o **LangGraph** oferece a infraestrutura de orquestração necessária para gerenciar estado e fluxos cíclicos de controle.

Esta seção disseca os componentes fundamentais para a construção de um agente, partindo da interface básica de chat até a definição de grafos de estado persistentes.

1.3.1. Componentes Atômicos: Modelos e Mensagens

Na arquitetura de agentes modernos, o LLM atua como o "cérebro" ou, mais formalmente, como o mecanismo de raciocínio. O LangChain abstrai as diferenças entre provedores (OpenAI, Google Gemini, Anthropic) através da classe `ChatModel`. Diferentemente de modelos de completação de texto antigos (que recebiam uma única *string*), os `ChatModels` operam sobre listas de **Mensagens**, que correspondem diretamente às **percepções** (P) na teoria de agentes.

As mensagens são objetos tipados que estruturam a comunicação:

- **SystemMessage:** Define a *persona*, instruções base e restrições do agente. Corresponde ao "conhecimento prévio" ou "design" do agente na teoria de Russell e Norvig.
- **HumanMessage:** Representa o *input* do usuário ou estímulo externo.
- **AIMessage:** Representa a resposta ou ação gerada pelo modelo.
- **ToolMessage:** (Crucial para agentes) Contém o resultado da execução de uma ferramenta, funcionando como uma nova percepção derivada de uma ação do agente.

Um exemplo básico de invocação de um modelo, que seria equivalente a um agente reflexo sem memória, pode ser implementado conforme a Listagem 1.1.

Este código implementa a função $f(P) \to A$, onde P é a lista `messages` e A é a `response`. Contudo, este sistema é **sem estado** (*stateless*). Cada invocação é independente, incapaz de manter um diálogo coerente ou realizar tarefas multi-etapas. Para evoluir para um agente completo, necessitamos de uma arquitetura que suporte persistência e ciclos.

```python
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, SystemMessage

model = ChatOpenAI(model="gpt-4o")

messages = [
    SystemMessage(content="Você é um assistente especializado em física teórica."),
    HumanMessage(content="Explique a segunda lei da termodinâmica.")
]

# Execução da função do agente (f:  P* -> A)
response = model.invoke(messages)
print(response.content)
```

Listing 1.1: Invocação básica de um Modelo de Chat

1.3.2. De Cadeias (Chains) a Grafos de Estado

Tradicionalmente, o LangChain popularizou o conceito de **Chains** (Cadeias) — sequências lineares de operações (Input $\rightarrow$ Passo 1 $\rightarrow$ Passo 2 $\rightarrow$ Output). Embora úteis para pipelines de dados determinísticos, as cadeias são insuficientes para agentes autônomos.

Um agente autônomo operando no mundo real frequentemente necessita de loops de controle: ele deve tentar uma ação, observar o resultado, corrigir se necessário e tentar novamente. Isso exige uma topologia de grafo, onde nós representam etapas de computação e arestas representam transições de controle, permitindo ciclos.

O **LangGraph** formaliza essa necessidade através da classe `StateGraph`. Um grafo no LangGraph é definido por três elementos principais:

1. **State (Estado):** Um esquema de dados compartilhado que persiste ao longo de toda a execução do grafo.
2. **Nodes (Nós):** Funções Python que recebem o estado atual, processam algo (ex: chamam o LLM), e retornam uma atualização para o estado.
3. **Edges (Arestas):** Lógica de controle que determina qual nó executar a seguir, podendo ser condicional (baseado na saída do LLM).

1.3.3. O Schema de Estado e Redutores

A gestão do estado é o diferencial crítico do LangGraph. Ao contrário de simplesmente sobrescrever variáveis, o framework utiliza o conceito de **Redutores** (inspirado em conceitos de programação funcional e sistemas como Redux).

Quando um nó retorna um valor, esse valor não substitui o estado global; ele é "reduzido" (mesclado) ao estado atual. Para listas de mensagens, a operação de redução mais comum é a **adição** (`operator.add`), garantindo que o histórico da conversa seja preservado incrementalmente. A Listagem 1.2 apresenta a definição canônica de um Estado para um agente conversacional.

```python
from typing import Annotated, TypedDict
from langgraph.graph.message import add_messages

# Definição do Schema do Estado
class AgentState(TypedDict):
    # 'messages' é a chave principal.
    # A anotação 'Annotated[list, add_messages]' instrui o grafo a:
    # 1. Permitir que nós retornem apenas a NOVA mensagem.
    # 2. Usar a função 'add_messages' para anexar essa nova mensagem
    # à lista existente, em vez de substituir a lista inteira.
    messages: Annotated[list, add_messages]
```

Listing 1.2: Definição de Estado com Redutores

1.3.4. Construindo o Grafo do Agente

Com o estado definido, a construção do agente segue um padrão declarativo. O grafo deve ser inicializado, os nós adicionados e as arestas conectadas. Um padrão comum é o grafo de nó único que cicla sobre si mesmo ou termina, dependendo da decisão do modelo. A Listagem 1.3 mostra a implementação de um agente que processa mensagens e mantém memória de curto prazo.

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

# 1. Definir a função do nó (o comportamento do agente)
def chatbot_node(state: AgentState):
    # O nó recebe o estado atual (histórico completo)
    # E invoca o modelo
    response = model.invoke(state["messages"])
    # Retorna apenas a atualização (a nova mensagem AI)
    return {"messages": [response]}

# 2. Inicializar o Grafo com o Schema de Estado
builder = StateGraph(AgentState)

# 3. Adicionar o nó ao grafo
builder.add_node("chatbot", chatbot_node)

# 4. Definir o fluxo de controle (Arestas)
# O fluxo inicia no ponto de entrada 'START' e vai para o nó 'chatbot'
builder.add_edge(START, "chatbot")
# Após responder, o agente encerra a execução
builder.add_edge("chatbot", END)

# 5. Compilar o grafo
# A compilação transforma a definição em um 'Runnable' executável
agent_graph = builder.compile()
```

Listing 1.3: Construção de Grafo com Memória

Este código materializa o conceito de **Agente Reativo Simples com Estado**. A função `compile()` cria uma estrutura otimizada que gerencia a passagem de mensagens. Quando executamos este grafo, o LangGraph cria automaticamente um *snapshot* do estado inicial, executa o nó `chatbot`, aplica o redutor `add_messages` para inserir a resposta do LLM no histórico e, finalmente, transita para o estado `END`.

Esta arquitetura é a fundação sobre a qual agentes mais complexos — capazes de usar ferramentas e tomar decisões de roteamento — são construídos. A introdução de **Arestas Condicionais** é o próximo passo lógico para permitir que o agente decida *se* deve parar ou *se* deve executar uma ação (ferramenta), implementando o verdadeiro paradigma de autonomia.

1.4. Agentes Autônomos: Do Reflexo à Razão e Ação

A arquitetura de Agente Reflexivo Simples com Estado, discutida anteriormente, permite a manutenção de um histórico de conversação, mas carece de uma característica fundamental para a verdadeira autonomia: a capacidade de intervir no ambiente para alterar seu estado ou recuperar informações externas. Na taxonomia de Russell e Norvig [4], agentes racionais operam através de sensores para perceber o ambiente e atuadores para agir sobre ele. No contexto de *Large Language Models* (LLMs), os "atuadores" são materializados através do mecanismo de *Tool Calling* (chamada de ferramentas).

A transição de um sistema conversacional passivo para um agente ativo é fundamentada teórica e arquiteturalmente no paradigma **ReAct** (*Reasoning and Acting*). Yao et al. [6] demonstram que a capacidade de raciocínio de um LLM é significativamente ampliada quando o modelo é instruído a gerar traços de raciocínio (*thoughts*) intercalados com ações específicas (*actions*) e a observação de seus resultados (*observations*). Diferente da abordagem *Chain-of-Thought* (CoT) isolada, que opera apenas no espaço latente do modelo, o ReAct permite a atualização dinâmica do conhecimento do agente através da interação com ferramentas. A Figura 1.2 detalha este fluxo de execução: o modelo raciocina (Reasoning), decide atuar (Action), recebe o retorno da ferramenta (Observation) e repete o ciclo se necessário.

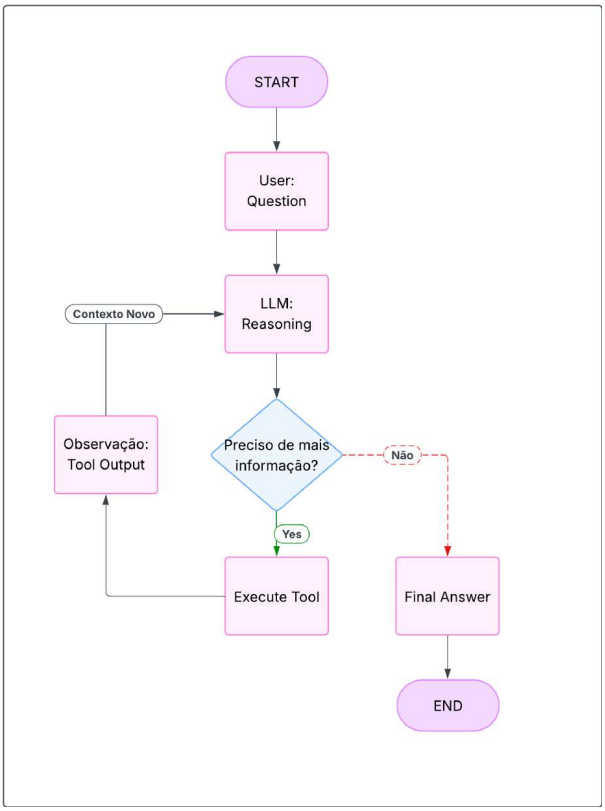

Figura 1.2: Fluxo de execução do paradigma ReAct: Raciocínio, Ação e Observação

1.4.1. Implementação do Ciclo Cognitivo no LangGraph

Para implementar essa autonomia no LangGraph, a topologia do grafo deve evoluir de uma sequência linear para um ciclo condicional. O agente deve decidir, a cada passo, se possui informações suficientes para responder ao usuário (retornando ao estado `END`) ou se necessita invocar uma ferramenta (transitando para um nó de ferramentas).

A materialização deste comportamento exige a definição de ferramentas como funções Python tipadas e a vinculação destas ao modelo de linguagem. A Listagem 1.4 apresenta a configuração essencial para um agente ReAct.

A materialização deste comportamento autônomo no LangGraph repousa sobre pilares técnicos que visam mitigar a natureza estocástica dos LLMs. O primeiro pilar é o **Vínculo de Ferramentas** (`bind_tools`). Ao injetar as assinaturas das funções diretamente no esquema da API do modelo (em vez de apenas descritas no prompt textual), reduz-se significativamente a alucinação na geração de argumentos.

Concomitantemente, a configuração da **Temperatura Determinística** (*temperature=0*) é mandatória para agentes que interagem com sistemas externos. Segundo Russell e Norvig [4], um agente racional deve selecionar a ação que maximiza sua medida de desempenho; em termos de engenharia, isso implica que, dado um mesmo estado e input, o agente deve selecionar consistentemente a mesma ferramenta.

```python
from langchain.tools import tool
from langgraph.prebuilt import ToolNode, tools_condition

# Definição de ferramentas com docstrings rigorosas
@tool
def search_database(query: str):
    """Realiza busca no banco de dados vetorial para informações contextuais."""
    # Implementação da busca...
    pass

# Vinculação das ferramentas ao modelo
# O método .bind_tools() injeta as assinaturas das funções
# no esquema da API do modelo, permitindo que o LLM
# estruture a saída JSON correta para execução.
llm_with_tools = llm.bind_tools([search_database])

def agent_node(state: AgentState):
    # O modelo decide se gera texto ou uma chamada de ferramenta
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}

# Construção do Grafo ReAct
workflow = StateGraph(AgentState)
workflow.add_node("agent", agent_node)
workflow.add_node("tools", ToolNode([search_database]))

workflow.add_edge(START, "agent")

# A função tools_condition verifica se a última mensagem contém 'tool_calls'.
# Se sim, roteia para o nó "tools"; caso contrário, encerra em END.
workflow.add_conditional_edges("agent", tools_condition)
workflow.add_edge("tools", "agent")
```

Listing 1.4: Agente ReAct com Ferramentas

1.5. Sistemas Multiagentes: Colaboração e Orquestração

À medida que a complexidade das tarefas aumenta, a capacidade de um único agente (monolítico) de manter o contexto e raciocinar sobre múltiplos domínios diminui. O paradigma de Sistemas Multiagentes (SMA) aborda essa limitação decompondo problemas complexos em subproblemas tratáveis por unidades especializadas.

Teoricamente, a transição para SMA no contexto de LLMs resgata conceitos de Inteligência Artificial Distribuída (DAI). No LangGraph, isso é implementado através de arquiteturas de grafo onde o **Estado** atua como um "Quadro Negro" (*Blackboard*), um padrão clássico de engenharia de software onde múltiplos especialistas leem e escrevem em uma memória compartilhada[1].

1.5.1. Paralelização e o Padrão Map-Reduce

Enquanto o ciclo ReAct introduz a autonomia sequencial, certas classes de problemas exigem a execução simultânea de sub-tarefas independentes para otimização da latência total do sistema. No contexto de Agentes Cognitivos, a paralelização permite que um LLM decomponha uma consulta complexa em múltiplos tópicos de investigação, delegue-os a instâncias de agentes (operários) e agregue os resultados subsequentes.

A justificativa matemática para esta arquitetura reside na redução do tempo total de execução. Em um sistema sequencial,$T_{seq} = \sum_{i=1}^{n} t_i$. Em uma topologia paralela ideal,

$T_{par} \approx \max(t_i) + \delta_{\text{agregação}}$, uma lógica visualizada na topologia da Figura 1.3.

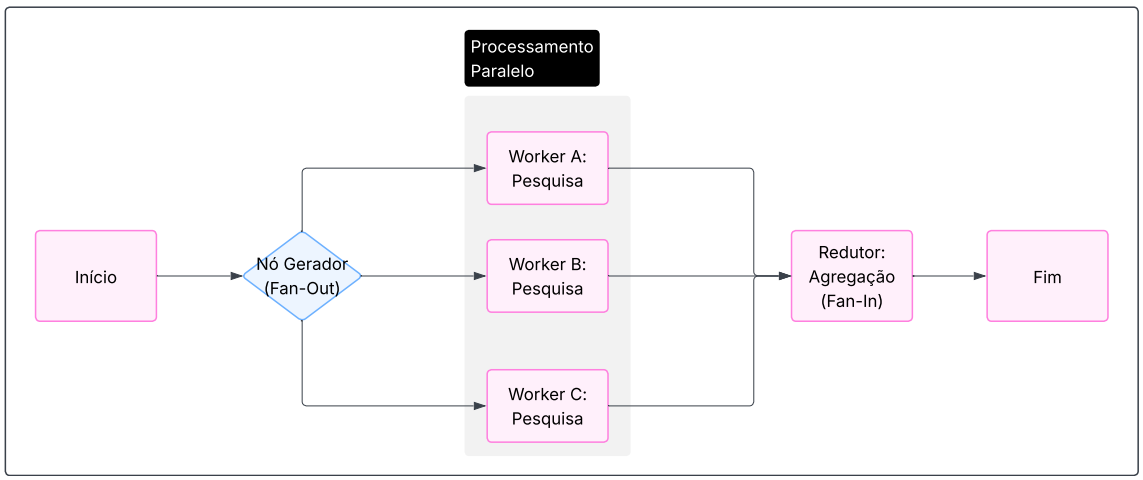

Figura 1.3: Topologia Map-Reduce para paralelização de tarefas em Agentes

Para evitar condições de corrida na escrita do estado paralelo, utiliza-se redutores especiais. A Listagem 1.5 demonstra o uso de redutores para garantir que escritas paralelas sejam fundidas corretamente.

```python
import operator
from typing import Annotated, TypedDict

class ResearchState(TypedDict):
    # O redutor operator.add garante que escritas paralelas
    # sejam fundidas em uma lista única, preservando todos os resultados.
    results: Annotated[list[str], operator.add]
```

Listing 1.5: Estado com Redutor para Paralelização

1.5.2. Arquiteturas Hierárquicas: Padrão Supervisor

A arquitetura de **Supervisor** introduz uma hierarquia onde um agente orquestrador delega tarefas a subagentes especialistas, mantendo a visão global do processo, um padrão observado em sistemas como MetaGPT e ChatDev [5]. Diferente de um roteamento estático (baseado em regras *if-else* rígidas), o roteamento em sistemas baseados em LLMs é semântico e dinâmico.Conforme ilustrado na Figura 1.4, o nó central (Supervisor) não executa o trabalho, mas atua como um roteador semântico, encaminhando a solicitação para o especialista mais adequado (Matemático ou Informações).

Para implementar este padrão de forma determinística, utiliza-se a técnica de *Structured Output* (Saída Estruturada). A Listagem 1.6 demonstra a definição do esquema de decisão e o nó supervisor.

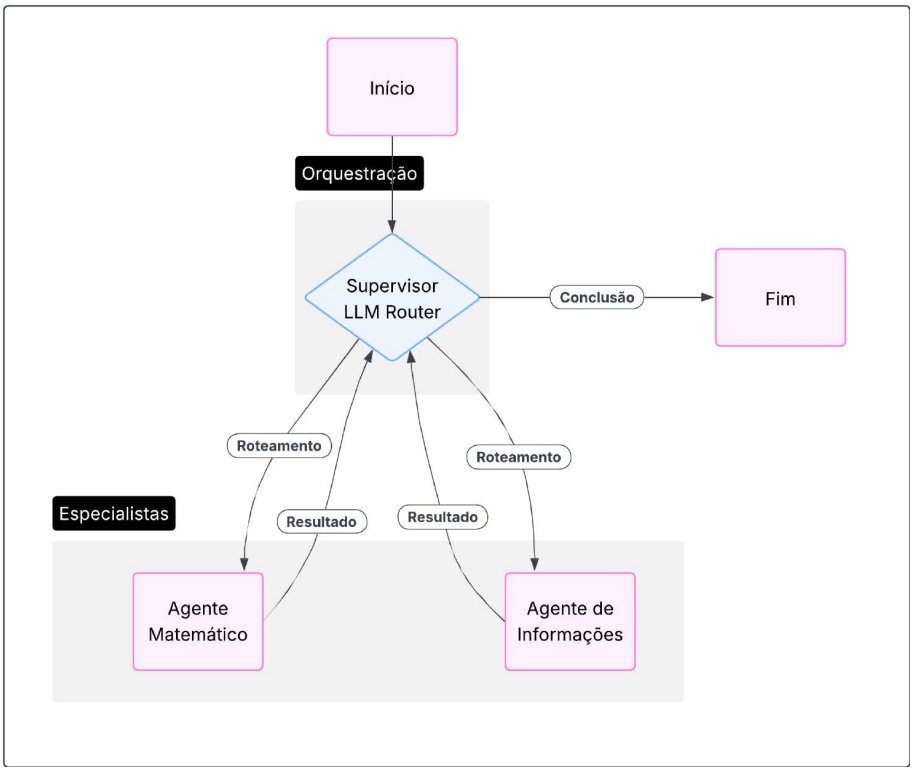

Figura 1.4: Arquitetura Hierárquica: O Supervisor atua como orquestrador central

```python
from typing import Literal
from pydantic import BaseModel

class RouteResponse(BaseModel):
    next: Literal["matematico", "informacoes", "finalizar"]

def supervisor_node(state: Estado):
    # O método .with_structured_output() acopla o esquema Pydantic
    # garantindo que o retorno seja um objeto Python validado
    chain = prompt | llm.with_structured_output(RouteResponse)
    decision = chain.invoke(state)

    # A decisão é persistida no estado
    return {"proximo_agente": decision.next}
```

Listing 1.6: Nó Supervisor com Saída Estruturada

1.6. Persistência de Estado e Memória de Longo Prazo

Um desafio fundamental na engenharia de agentes sobre LLMs é a natureza *stateless* (sem estado) dos modelos de fundação: a função $f(prompt) \rightarrow texto$ não retém memória de invocações anteriores. Contudo, aplicações reais exigem continuidade, permitindo que o usuário retome conversas, faça referências a interações passadas e mantenha transações de longa duração.

No framework LangGraph, a persistência não é apenas um registro de logs, mas

um mecanismo ativo de **Checkpointing**. O *checkpointer* intercepta o fluxo de execução a cada transição de nó, serializando o objeto `State` e armazenando-o em uma base durável (como SQLite ou PostgreSQL). Isso permite não apenas a memória conversacional, mas também funcionalidades avançadas como "Time Travel" (reverter o estado para um ponto anterior) e "Human-in-the-loop" (pausar para aprovação humana e retomar).

A implementação da persistência exige a configuração de um gerenciador de memória no momento da compilação do grafo e a utilização de identificadores de sessão (`thread_id`) durante a invocação, conforme a Listagem 1.7.

```
from langgraph.checkpoint.memory import MemorySaver

# Inicialização do Checkpointer
# Em produção, substitui-se MemorySaver (RAM) por AsyncSqliteSaver
memory = MemorySaver()

# O parâmetro 'checkpointer' habilita o versionamento do estado
app = workflow.compile(checkpointer=memory)

# Configuração da Sessão
# O 'thread_id' atua como chave primária para recuperação do contexto
config = {"configurable": {"thread_id": "sessao_usuario_123"}}

# Invocação
# O grafo carregará automaticamente as mensagens prévias associadas ao
    ID
response = app.invoke(inputs, config=config)
```

Listing 1.7: Compilação com Persistência

Tecnicamente, o *checkpointer* resolve o problema da **persistência do planejamento** descrito por Ghallab et al. [2]. Ele assegura que o plano parcial construído pelo agente (o conjunto de mensagens e resultados de ferramentas acumulados) sobreviva a interrupções do sistema ou à latência entre interações do usuário.

1.7. Estudo de Caso: Agente de E-commerce

Consolidando os conceitos apresentados, esta seção demonstra um Assistente de E-commerce que executa alterações de estado via banco de dados SQL. A inclusão da camada de persistência transcende a mera funcionalidade de histórico, habilitando capacidades cognitivas e operacionais avançadas no sistema:

- **Conversação Multi-Turno:** O agente mantém acesso a fatos e entidades citados em interações anteriores, simulando uma memória episódica de curto prazo.
- **Human-in-the-loop:** O sistema pode interromper sua execução antes de ações sensíveis e aguardar aprovação humana.
- **Depuração Temporal:** Permite aos engenheiros inspecionar estados passados da conversa e "rebobinar" a execução para testar caminhos alternativos de raciocínio (*Time Travel*).

A Figura 1.5 detalha a arquitetura completa deste estudo de caso, demonstrando a interação entre os agentes especialistas e o banco de dados.

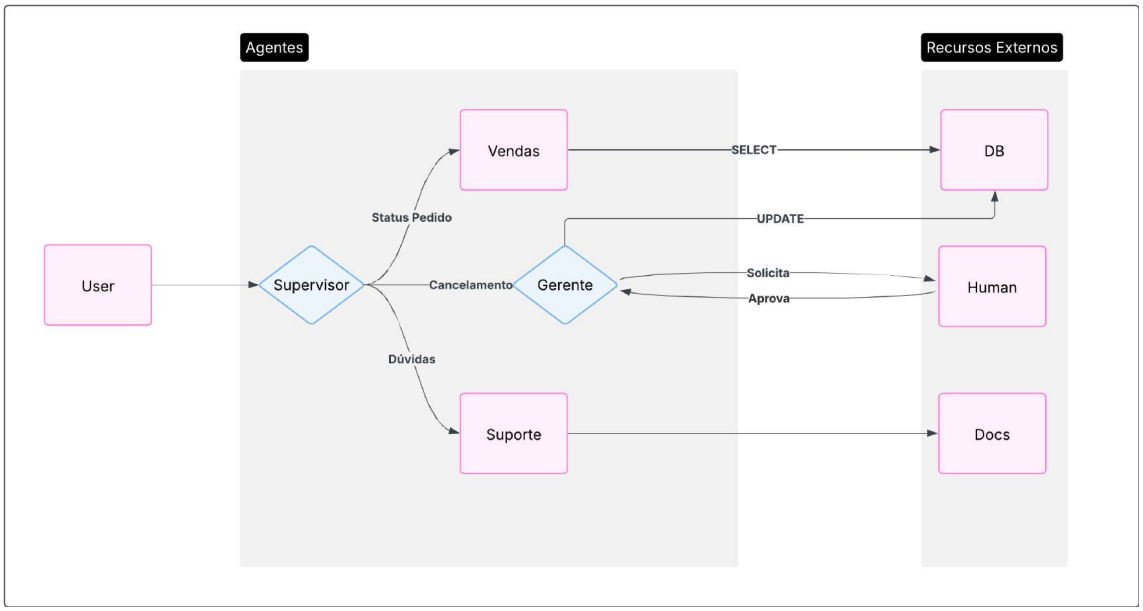

Figura 1.5: Arquitetura do Agente de E-commerce com persistência e ferramentas SQL

A integração com banco de dados SQL via *Tool Calling* exige rigor na definição das ferramentas. O LLM deve ser capaz de inferir corretamente os parâmetros da consulta. A Listagem 1.8 ilustra como uma ferramenta de consulta SQL é exposta ao agente.

Neste cenário, a persistência é vital. Se um usuário pergunta "Onde está meu pedido?" (acionando o agente de Vendas) e, em seguida, diz "Cancele-o" (acionando o agente de Gerência), o sistema deve manter o contexto de *qual* pedido está sendo discutido durante a transição entre os especialistas.

```python
import sqlite3
from langchain.tools import tool

@tool
def consultar_pedidos(cliente: str) -> list:
    """
    Consulta o status dos pedidos de um cliente específico no banco de dados.
    Retorna uma lista de tuplas contendo produto, status e valor.
    """
    conn = sqlite3.connect('ecommerce_v2.db')
    cursor = conn.cursor()
    cursor.execute(
        "SELECT produto, status, valor FROM pedidos WHERE cliente = ?",
        (cliente,)
    )
    return cursor.fetchall()

# A docstring é consumida pelo LLM para entender a semântica da ferramenta.
# A tipagem (cliente: str) define o esquema JSON esperado na chamada.
```

Listing 1.8: Ferramenta SQL com Tipagem Rigorosa

1.8. Observabilidade e Avaliação Sistemática

A transição de um protótipo em *notebook* para um sistema de produção impõe um desafio metodológico: a natureza estocástica dos *Large Language Models*. Diferente de software determinístico, onde a função $f(x) = y$ é constante, em agentes generativos $P(y|x)$ é uma distribuição de probabilidade. Isso torna testes unitários tradicionais baseados em asserções exatas (`assert result == expected`) insuficientes.

Para mitigar a incerteza e garantir a confiabilidade do agente, a engenharia deve adotar práticas de **Observabilidade** (Tracing) e **Avaliação Baseada em Modelo** (*LLM-as-a-Judge*).

1.8.1. Tracing de Cadeias e Grafos

A observabilidade em sistemas baseados em grafos (como LangGraph) não se resume ao registro de *logs* textuais. É necessário capturar a **Árvore de Execução** completa, detalhando a latência, o consumo de tokens e, crucialmente, os estados intermediários em cada transição de nó, conforme ilustrado na Figura 1.6.

Ferramentas de rastreamento, como o LangSmith, permitem decompor a falha na resposta final em seus componentes causais primários, indo além da análise superficial do texto gerado. Através da inspeção do *trace*, é possível discernir se o erro originou-se de uma **Recuperação Falha**, onde os documentos relevantes não foram encontrados no banco vetorial; de um **Raciocínio Incorreto**, onde o modelo alucinou conclusões falsas mesmo possuindo os dados corretos no contexto; ou de uma **Falha na Ferramenta**, decorrente de exceções de *runtime* ou erros de sintaxe em consultas SQL e APIs externas.

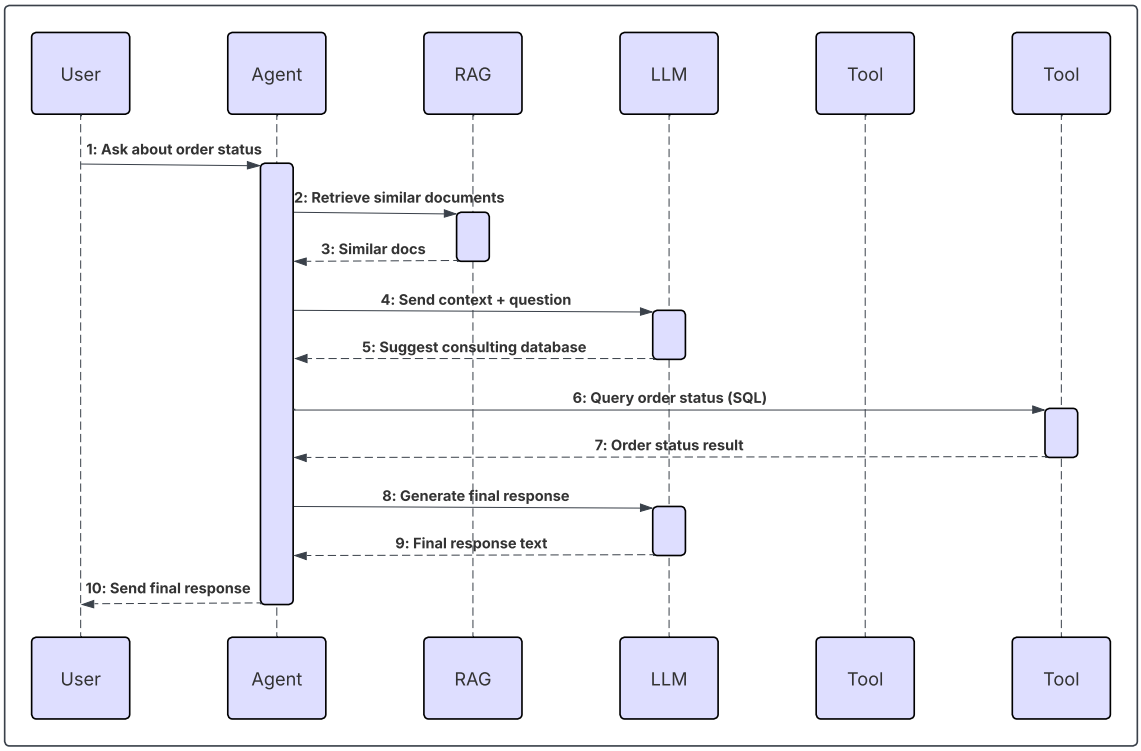

Figura 1.6: Visualização de trace de execução com decomposição temporal

1.8.2. Avaliação Automatizada (LLM-as-a-Judge)

A avaliação escalável de agentes conversacionais utiliza um LLM mais robusto (o "Juiz") para avaliar as saídas do agente em desenvolvimento (o "Aluno").Esta técnica, conhecida como avaliação subjetiva baseada em modelo, tem sido adotada para mitigar os alto custos da avaliação humana [5].

A implementação de um avaliador no ecossistema LangChain envolve a criação de um *dataset* de referência (perguntas e respostas ideais) e a execução de uma cadeia de avaliação. A Listagem 1.9 demonstra como definir um critério de correção factual.

A justificativa técnica para o uso de `labeled_criteria` em vez de comparação de strings (como distância de Levenshtein) reside na capacidade do LLM de entender equivalência semântica.

```python
from langsmith import Client
from langchain.evaluation import load_evaluator

# Definição do critério de avaliação
# O avaliador "labeled_criteria"compara a previsão com a referência
evaluator = load_evaluator("labeled_criteria", criteria="correctness")

def avaliar_agente(run, example):
    # Extrai a resposta do agente e a resposta esperada do dataset
    predicao = run.outputs["messages"][-1].content
    referencia = example.outputs["resposta_esperada"]

    # O LLM "Juiz"analisa semanticamente se a predição atende à
       referência
    score = evaluator.evaluate_strings(
        prediction=predicao,
        reference=referencia,
        input=example.inputs["pergunta"]
    )
    return {"key": "correctness", "score": score["score"]}

# A execução deste script sobre um dataset gera métricas quantitativas
# (ex:  85% de acurácia) permitindo comparar diferentes versões do
   grafo
```

Listing 1.9: Avaliador de Correção Factual

1.9. Arquitetura de Implantação

A etapa final do ciclo de vida do agente é a exposição de sua lógica como um serviço consumível por interfaces de usuário (Frontend Web, Mobile ou WhatsApp). A arquitetura recomendada segue o padrão de **Microsserviços Assíncronos** alinhando-se ao paradigma emergente de Agent-as-a-Service (AaaS), onde agentes encapsulam capacidades cognitivas expostas via APIs padronizadas [1].

O objeto compilado do LangGraph (`app = graph.compile()`) adere à interface `Runnable`, o que facilita sua integração com servidores web assíncronos como FastAPI. O framework **LangServe** automatiza a criação de endpoints RESTful padronizados, suportando não apenas requisições síncronas, mas também *streaming* de tokens, essencial para a experiência do usuário em chatbots (efeito de digitação). O fluxo completo de dados nesta arquitetura de microsserviços é apresentado na Figura 1.7.

A Listagem 1.10 ilustra a conversão do grafo desenvolvido nas seções anteriores em uma API de produção.

Esta arquitetura de microsserviços permite que o agente seja escalado horizontalmente, com múltiplas instâncias do servidor processando requisições em paralelo, garantindo alta disponibilidade e baixa latência mesmo sob carga elevada.

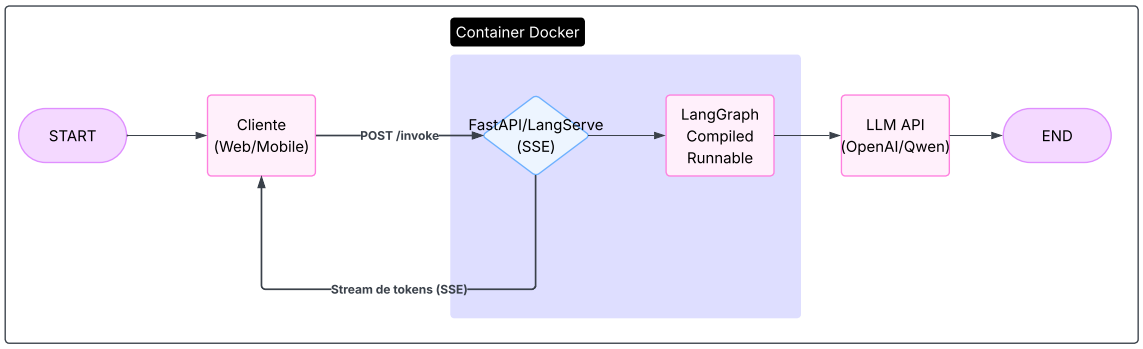

Figura 1.7: Diagrama de Implantação e fluxo de dados via Microsserviços

```python
from fastapi import FastAPI
from langserve import add_routes

app = FastAPI(
    title="Servidor do Agente de E-commerce",
    version="1.0",
    description="API para interação com Agente Multiagente via
        LangGraph"
)

# Criação automática de rotas:
# POST /agente/invoke - Execução síncrona
# POST /agente/stream - Streaming de tokens e atualizações de estado
# POST /agente/batch - Processamento em lote
add_routes(app, graph_app, path="/agente")

if __name__ == "__main__":
    import uvicorn
    # Execução do servidor em host local
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

Listing 1.10: Exposição via LangServe

1.10. Considerações Finais

A evolução dos sistemas conversacionais, partindo de cadeias lineares simples para arquiteturas multiagente complexas baseadas em grafos, representa uma mudança de paradigma na Inteligência Artificial Aplicada: a transição da "Engenharia de Prompt"para a "Engenharia de Fluxo"(Flow Engineering). Neste capítulo, demonstramos que a construção de agentes robustos exige a integração de conceitos fundamentais da Ciência da Computação:

1. **Racionalidade e Autonomia:** Via ciclo ReAct.
2. **Sistemas Distribuídos e Coordenação:** Modelados via arquiteturas de Supervisor e Paralelismo
3. **Persistência:** Via Checkpointers e Banco de Dados

Desafios persistentes incluem **Latência e Custo**, onde arquiteturas multiagente multiplicam chamadas ao LLM, e **Segurança e Alinhamento**, garantindo que agentes autônomos não executem ações destrutivas.

Apesar dos avanços significativos na orquestração de agentes, pesquisadores e engenheiros enfrentam desafios persistentes para a adoção massiva dessa tecnologia. O primeiro obstáculo reside no binômio Latência e Custo; arquiteturas multiagente multiplicam exponencialmente o número de chamadas ao LLM para uma única resolução. O uso de modelos menores e especializados (Small Language Models), ajustados via Fine-Tuning para tarefas específicas (como geração de SQL), apresenta-se como uma solução viável para a sustentabilidade econômica. Simultaneamente, impõe-se o desafio da Segurança e Alinhamento [5]. Garantir que um agente autônomo com acesso a ferramentas de escrita não execute ações destrutivas ou indesejadas requer o desenvolvimento de guardrails determinísticos e conformidade ética rigorosa [3] que operem externamente ao espaço latente do modelo, assegurando que a autonomia do agente permaneça circunscrita às regras de negócio definidas.

Conclui-se, portanto, que o desenvolvimento de Agentes de IA é uma disciplina interdisciplinar, exigindo rigor tanto no design algorítmico quanto na modelagem do conhecimento e na arquitetura de software.

Agradecimentos

O presente trabalho foi financiado parcialmente pela SSP/PI, com apoio da Fundação Cultural e de Fomento à Pesquisa, Ensino, Extensão e Inovação (FADEX). Agradecemos também à equipe técnica da SSP e Trinhub, pelo apoio e disponibilidade constante em conversar com os pesquisadores do Departamento de Computação da UFPI no desenvolver das habilidades da equipe nas tecnologias adotadas nesse minicurso.

Referências

[1] Hana Derouiche, Haithem Mezni, and Zaki Brahmi. Agentic ai frameworks: Architectures, protocols, and design challenges. *arXiv preprint arXiv:2508.10146*, 2025.

[2] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 1998.

[3] Satyadhar Joshi. Advancing innovation in financial stability: A comprehensive review of ai agent frameworks, challenges and applications. *World Journal of Advanced Engineering Technology and Sciences*, 14(2):117–126, 2025.

[4] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4th edition, 2020.

[5] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.

[6] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Wen, and Mike Lewis. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.