# **LangChain na Prática: O primeiro passo com Agentes de IA**

**Igor Medeiros**

## **Informações de Copyright**

© 2025 Igor Medeiros. Todos os direitos reservados.

Este livro faz parte da série “IA na Prática”. Lançamento: 20 de Julho de 2025. ASIN: B0FJ7L9NTM Link do ebook na Amazon Brasil: https://www.amazon.com.br/dp/B0FJ7L9NTM

Nenhuma parte desta publicação pode ser reproduzida, distribuída ou transmitida por qualquer forma ou meio, incluindo fotocópia, gravação ou outros métodos eletrônicos ou mecânicos, sem a permissão prévia por escrito do autor, exceto no caso de breves citações incorporadas em resenhas críticas e outros usos não comerciais permitidos pela lei de direitos autorais.

Contato: igor@igormedeiros.com.br  
Website: https://igormedeiros.com.br

## **Aviso Legal**

Este livro e seu conteúdo são fornecidos “como estão”, sem garantias de qualquer tipo, expressas ou implícitas. O autor e o editor não se responsabilizam por quaisquer erros ou omissões, ou por quaisquer danos resultantes do uso das informações contidas neste livro. Os exemplos de código são fornecidos para fins educacionais e podem exigir modificação para uso em produção. A execução dos códigos e a utilização das chaves de API são de inteira responsabilidade do leitor.

## **Prefácio**

[Este prefácio será escrito por Felipe Gagliazzo.]

## **Dedicatória**

Aos meus pais, José Maurício e Genilda, a quem busco honrar todos os dias sendo um ser humano melhor.

À minha filha Melissa, a luz que ilumina minha jornada e me inspira todos os dias a ser melhor.

Ao Dr. Paulo H Pires de Aguiar, o médico e amigo que, com maestria e humanidade, orquestrou o salvamento da minha vida.

## **Sumário**

* **Página de Rosto**
* **Informações de Copyright**
* **Prefácio**
* **Dedicatória**
* **Agradecimentos**
* **Índice Detalhado**
* **Introdução**
  + A Revolução Silenciosa dos LLMs
  + Por que LangChain? O Poder da Orquestração
  + Uma Visão de Resolvedor de Problemas
  + Como Aproveitar ao Máximo Este Livro
* **Capítulo 1: Introdução ao LangChain — Fundamentos e Conceitos Essenciais**
  + O que é LangChain? Uma Breve História em Meio à Tempestade da IA
  + O Problema da Conversa Iterativa e o Nascimento das “Chains”
  + Exercício Prático: Hello, LangChain!
  + A Arquitetura Central: Os Componentes Essenciais
  + O Ecossistema LangChain: Mais que uma Biblioteca
  + Tabela 1: LangChain vs. Frameworks Concorrentes
  + Pontos Chave
  + Resumo do Capítulo
* **Capítulo 2: Configurando o Ambiente de Desenvolvimento Python para LangChain**
  + Meu Ambiente de Batalha: Por que Linux e Ferramentas de Linha de Comando
  + Gerenciando Versões do Python como um Profissional: pyenv
  + Um Terminal com Superpoderes: zsh e Oh My Zsh
  + Segurança e Conveniência com Git: Chaves SSH
  + Minhas Ferramentas de Batalha: VS Code
  + Gerenciando Dependências: A Revolução do uv e pyproject.toml
  + Gerenciando Segredos: Chaves de API e Variáveis de Ambiente
  + Considerações sobre Hardware
  + Troubleshooting Comum
  + Pontos Chave
  + Resumo do Capítulo
  + Teste seu Conhecimento
* **Capítulo 3: Manipulação de Prompts e Modelos de Linguagem com LangChain**
  + A Arte e a Ciência da Engenharia de Prompts
  + Templates de Prompt: Reutilização e Dinamismo
  + Integrando Modelos de Linguagem (LLMs)
  + Exercício Prático: Um Tradutor Multilíngue Dinâmico
  + Resumo do Capítulo
  + Pontos Chave
  + Teste seu Conhecimento
* **Capítulo 4: Construção de Pipelines: Da SequentialChain à LCEL**
  + A Evolução das Chains: Do Imperativo ao Declarativo
  + O Jeito Clássico: LLMChain e SequentialChain (OBSOLETO)
  + A Revolução da LCEL: Por que Devemos Usá-la?
  + LCEL na Prática: Exercícios Essenciais
  + Checklist de Construção de Pipelines
  + Pontos Chave
  + Teste seu Conhecimento
* **Capítulo 5: Desenvolvimento de Agentes Autônomos e Multiagentes**
  + O que é um Agente? O LLM como Cérebro
  + Engenharia de Contexto: A Evolução do Prompt
  + RAG (Retrieval-Augmented Generation): Ampliando o Conhecimento do LLM
  + Componentes de um Agente: Ferramentas e o Executor
  + Exercício Prático: Agente de Pesquisa Simples
  + Sistemas Multiagentes e a Magia do LangGraph
  + Pontos Chave
  + Resumo do Capítulo
  + Teste seu Conhecimento
* **Capítulo 6: Ferramentas e Agentes: Capacitando LLMs com Ações**
  + O que são Ferramentas (Tools)?
  + Agentes (Agents): O Cérebro por Trás das Ações
  + Exercício 1: Agente de Análise de Vendas com Ferramenta SQL
  + Resumo do Capítulo
  + Pontos Chave
  + Teste seu Conhecimento
* **Capítulo 7: Técnicas Avançadas: Memória, Feedback e Aprendizado Contínuo**
* **Capítulo 8: Testes, Debugging e Otimização de Aplicações LangChain**
* **Capítulo 9: LangChain na Produção: Escalabilidade, Observabilidade e Segurança**
  + Pontos Chave
  + Resumo do Capítulo
  + Teste seu Conhecimento
  + Pontos Chave
  + Resumo do Capítulo
  + Teste seu Conhecimento
* Próximos passos
* Mensagem final
* Sobre o autor
* Referências
* Apêndice: Glossário de Termos

## **Agradecimentos**

Nenhum livro é escrito no vácuo, e este não é exceção. Minha gratidão transborda por aqueles que, de formas distintas, mas igualmente importantes, tornaram esta obra possível.

A **Felipe Gagliazzo**, meu Mestre Jedi particular. Sua sabedoria e paciência são um farol. Obrigado pelo aprendizado contínuo e por me dar o apoio necessário para me sentir mais confiante no trabalho e na vida. Sua mentoria é um presente que valorizo imensamente.

A **Nicolli Botelho**, por enxergar além e me confiar desafios que são muito mais que uma declaração de confiança no meu trabalho; são oportunidades generosas de crescimento que me impulsionam a cada dia. Sua liderança inspiradora me motiva a ser um profissional melhor.

A **Marcus Borin**, pelo encorajamento constante e por sempre me empurrar a subir um degrau a mais na infinita escada do conhecimento. Sua crença no meu potencial é um motor poderoso que me faz acreditar que posso ir mais longe.

A **Emilly Gomes**, por uma amizade e um apoio incondicional que me fazem sentir parte de uma comunidade verdadeiramente acolhedora. Sua presença torna a jornada muito mais leve e divertida, e sou grato por cada conversa e risada.

A **Anne Aguiar**, pelo encorajamento sincero e por acreditar em meu trabalho mesmo quando eu duvidava. Sua fé foi um combustível essencial para que este projeto saísse do papel e se tornasse realidade.

Ao **Flavio Pires**, pela amizade inestimável e pela paciência de um monge ao escutar meus longos e intermináveis áudios no WhatsApp divagando sobre agentes de IA. Suas perguntas e insights sempre me ajudaram a clarear as ideias.

Ao **Fernando Fish**, por ser um exemplo tão forte de rigor técnico e científico. Seu profissionalismo, aliado a um carinho e uma inspiração contagianes, são uma referência que busco seguir em minha própria carreira.

À **Patricia Baena**, por acreditar em mim desde o início, por confiar no meu trabalho e por ter a generosidade de sempre buscar me apresentar portas que só eu poderia abrir. Sua mentoria e visão de futuro foram cruciais em minha trajetória.

À **Giselle Arruda**, por segurar minha mão em momentos difíceis, quando precisei de ajuda para me reerguer profissionalmente. Você foi a pessoa que me aparou e me guiou com uma força e uma empatia que jamais esquecerei.

Ao **Fernando Lopes Jr.**, por ser um amigo e um incentivador, sempre me apoiando nos momentos mais difíceis.

Ao **Eduardo Bregaida**, que confiou em mim e me apresentou uma possibilidade de recomeço quando eu mais precisei, enquanto enfrentava uma terrível fase de problemas de saúde mental e buscava um caminho de volta ao mercado de TI. Sua confiança foi um ponto de virada.

À **Marina**, **Adriana** e **Julio**, por confiarem sempre em mim para voltar ao cenário das palestras técnicas pelo Brasil e outros países, ainda que de forma online, no contexto interno corporativo. Sua crença no meu potencial foi fundamental para essa retomada.

À **Cristiane Tokarski**, por ter surgido como um anjo que agiu corrigindo o curso que a vida havia me conduzido, onde eu não me sentia “encaixado” na trilha do meu proposito profissional. Sua orientação foi um farol que me ajudou a reencontrar meu caminho e propósito.

Ao **Marcello Manzan**, um irmão de alma que a faculdade me deu e que hoje é como se fosse da minha família. Sua busca incansável pela excelência profissional é uma inspiração diária, um farol que me guia. Mais do que isso, sou grato pela sua amizade inabalável e pelas risadas que, ao longo dos anos, transformaram desafios em memórias preciosas.

À vasta e vibrante comunidade open source do LangChain e de todas as bibliotecas e ferramentas que tornaram este livro possível. Seu trabalho incansável e colaboração são a verdadeira força motriz por trás da inovação em IA. Reconhecemos e agradecemos a todos os projetos de código aberto e seus contribuidores, cujas licenças e termos de uso foram respeitados na criação deste material.

A cada um de vocês, meu mais profundo e sincero obrigado.

## **Agradecimentos**

Nenhum livro é escrito no vácuo, e este não é exceção. Minha gratidão transborda por aqueles que, de formas distintas, mas igualmente importantes, tornaram esta obra possível.

A **Felipe Gagliazzo**, meu Mestre Jedi particular. Sua sabedoria e paciência são um farol. Obrigado pelo aprendizado contínuo e por me dar o apoio necessário para me sentir mais confiante no trabalho e na vida. Sua mentoria é um presente que valorizo imensamente.

A **Nicolli Botelho**, por enxergar além e me confiar desafios que são muito mais que uma declaração de confiança no meu trabalho; são oportunidades generosas de crescimento que me impulsionam a cada dia. Sua liderança inspiradora me motiva a ser um profissional melhor.

A **Marcus Borin**, pelo encorajamento constante e por sempre me empurrar a subir um degrau a mais na infinita escada do conhecimento. Sua crença no meu potencial é um motor poderoso que me faz acreditar que posso ir mais longe.

A **Emilly Gomes**, por uma amizade e um apoio incondicional que me fazem sentir parte de uma comunidade verdadeiramente acolhedora. Sua presença torna a jornada muito mais leve e divertida, e sou grato por cada conversa e risada.

A **Anne Aguiar**, pelo encorajamento sincero e por acreditar em meu trabalho mesmo quando eu duvidava. Sua fé foi um combustível essencial para que este projeto saísse do papel e se tornasse realidade.

Ao **Flavio Pires**, pela amizade inestimável e pela paciência de um monge ao escutar meus longos e intermináveis áudios no WhatsApp divagando sobre agentes de IA. Suas perguntas e insights sempre me ajudaram a clarear as ideias.

Ao **Fernando Fish**, por ser um exemplo tão forte de rigor técnico e científico. Seu profissionalismo, aliado a um carinho e uma inspiração contagianes, são uma referência que busco seguir em minha própria carreira.

À **Patricia Baena**, por acreditar em mim desde o início, por confiar no meu trabalho e por ter a generosidade de sempre buscar me apresentar portas que só eu poderia abrir. Sua mentoria e visão de futuro foram cruciais em minha trajetória.

À **Giselle Arruda**, por segurar minha mão em momentos difíceis, quando precisei de ajuda para me reerguer profissionalmente. Você foi a pessoa que me aparou e me guiou com uma força e uma empatia que jamais esquecerei.

Ao **Fernando Lopes Jr.**, por ser um amigo e um incentivador, sempre me apoiando nos momentos mais difíceis.

Ao **Eduardo Bregaida**, que confiou em mim e me apresentou uma possibilidade de recomeço quando eu mais precisei, enquanto enfrentava uma terrível fase de problemas de saúde mental e buscava um caminho de volta ao mercado de TI. Sua confiança foi um ponto de virada.

À **Marina**, **Adriana** e **Julio**, por confiarem sempre em mim para voltar ao cenário das palestras técnicas pelo Brasil e outros países, ainda que de forma online, no contexto interno corporativo. Sua crença no meu potencial foi fundamental para essa retomada.

À **Cristiane Tokarski**, por ter surgido como um anjo que agiu corrigindo o curso que a vida havia me conduzido, onde eu não me sentia “encaixado” na trilha do meu proposito profissional. Sua orientação foi um farol que me ajudou a reencontrar meu caminho e propósito.

Ao **Marcello Manzan**, um irmão de alma que a faculdade me deu e que hoje é como se fosse da minha família. Sua busca incansável pela excelência profissional é uma inspiração diária, um farol que me guia. Mais do que isso, sou grato pela sua amizade inabalável e pelas risadas que, ao longo dos anos, transformaram desafios em memórias preciosas.

À vasta e vibrante comunidade open source do LangChain e de todas as bibliotecas e ferramentas que tornaram este livro possível. Seu trabalho incansável e colaboração são a verdadeira força motriz por trás da inovação em IA. Reconhecemos e agradecemos a todos os projetos de código aberto e seus contribuidores, cujas licenças e termos de uso foram respeitados na criação deste material.

A cada um de vocês, meu mais profundo e sincero obrigado.

## **Introdução**

A vida nos conduz por caminhos únicos. Antes de mergulharmos no código e nos conceitos que preenchem as páginas a seguir, gostaria que você fizesse uma pequena pausa. Reflita sobre as razões que te trouxeram até aqui, até este livro. Foi a curiosidade sobre o boom da Inteligência Artificial? A necessidade de resolver um problema específico no seu trabalho? Ou talvez um desejo profundo de criar algo novo, algo que possa, de alguma forma, impactar o mundo?

Seja qual for a sua motivação, saiba que ela é a faísca inicial de uma jornada transformadora. Este livro nasceu de uma motivação parecida: a necessidade de criar uma ponte. Uma ponte entre a teoria complexa dos Modelos de Linguagem de Grande Escala (LLMs) e a prática diária de nós, desenvolvedores Python. O LangChain surgiu como uma ferramenta revolucionária, uma espécie de canivete suíço que nos permite orquestrar, conectar e dar superpoderes a esses LLMs. Mas, como toda ferramenta poderosa, ela vem com sua própria curva de aprendizado.

Minha intenção aqui não é apenas entregar um manual técnico. É compartilhar a experiência vivida na trincheira, criando agentes inteligentes que resolvem problemas reais de negócios e, o que mais me move, que têm o potencial de impactar vidas. Minha grande paixão é a área da saúde, um campo onde a tecnologia, quando aplicada com propósito e empatia, pode reescrever histórias.

Espero que este guia seja um farol na sua jornada. Que ele te dê a confiança para transformar teoria em código, e código em soluções elegantes e eficientes. E para que nossa conversa não termine na última página, criei um espaço para continuarmos essa troca: um “Gêmeo Digital” deste livro, um assistente com quem você pode conversar, tirar dúvidas e explorar os conceitos que veremos aqui. Ele é a alma viva deste projeto, e você pode acessá-lo em: http://chat.com/gemini-ebook-com-alma.

Vamos começar?

### **A Revolução Silenciosa dos LLMs**

Se você é um desenvolvedor Python, provavelmente sentiu a terra tremer nos últimos anos. O que antes parecia ficção científica — máquinas que conversam, escrevem código, criam imagens e raciocinam sobre dados complexos — de repente se tornou parte do nosso kit de ferramentas diário. Os Modelos de Linguagem de Grande Escala (LLMs), como os da família GPT, Gemini, Claude e Llama, não são apenas chatbots glorificados; eles representam uma mudança fundamental na forma como construímos software.

A ascensão dos LLMs não é apenas uma evolução tecnológica; é uma **revolução silenciosa** que está redefinindo a interação humano-computador e a própria engenharia de software. Eles não são apenas ferramentas, mas parceiros de raciocínio que amplificam nossa capacidade de criar e inovar.

Lembro-me do meu primeiro contato com um “prompt” em 1996, no MS-DOS. Aquela tela preta com um cursor piscando, esperando um comando. A palavra “prompt” vem do latim “promptus”, que significa “pronto”, “disposto”, “rápido”. No contexto da computação, era o sinal de que o sistema estava pronto para receber uma instrução. Hoje, com os LLMs, o conceito de “prompt” evoluiu, mas a essência permanece: é a nossa forma de dar uma instrução, de iniciar um diálogo com a máquina.

É importante notar que um produto como o ChatGPT não é apenas um LLM (como o GPT-4o) isolado. Ele é uma aplicação completa e complexa, que já utiliza internamente muitos dos conceitos que vamos aprender aqui: RAG para buscar informações atualizadas, ferramentas (Tools) para executar ações, e memória para manter o contexto da conversa. Essa complexidade por trás de uma interface aparentemente simples é o que torna a engenharia de LLMs tão fascinante e desafiadora.

Estamos saindo de uma era puramente imperativa, onde dizemos à máquina *como* fazer cada passo, para uma era mais declarativa, onde descrevemos *o que* queremos e o modelo nos ajuda a chegar lá. Essa mudança de paradigma exige uma nova forma de pensar e de construir software. Isso abre um universo de possibilidades, mas também traz um novo conjunto de desafios:

* **Conectar LLMs a dados privados:** Como fazer com que esses cérebros de IA poderosos, mas isolados, acessem e utilizem nossos dados internos e proprietários de forma segura e eficiente?
* **Interagir com APIs externas:** Como permitir que os LLMs executem ações no mundo real, como enviar e-mails, consultar bancos de dados ou interagir com sistemas legados?
* **Gerenciar memória e contexto:** Como dar-lhes memória para que possam manter uma conversa coerente e contextualizada ao longo do tempo, sem “esquecer” o que foi dito anteriormente?
* **Orquestrar tarefas complexas:** Como encadear múltiplos passos e decisões para resolver problemas que exigem mais do que uma única interação?

É crucial, no entanto, manter uma perspectiva crítica. Como o renomado filósofo Luciano Floridi adverte, a inteligência artificial, se mal compreendida ou aplicada sem ética, pode levar à “idiotice artificial” – a automação de processos sem discernimento ou sabedoria. Nosso objetivo com este livro é ir além da mera automação, buscando a amplificação da inteligência humana, não sua substituição cega.

É exatamente aqui que a nossa história começa, e onde o LangChain se torna uma ferramenta indispensável.

### **Por que LangChain? O Poder da Orquestração**

Em meio a essa explosão de possibilidades, surgiu o LangChain. Pense no seu LLM favorito como um motor de Fórmula 1: incrivelmente potente, mas inútil sem um chassi, um sistema de transmissão, fiação e um painel de controle. O LangChain é exatamente isso: o framework que fornece a “transmissão, a fiação e o sistema de controle” para o motor do LLM.

Lançado no final de 2022, o LangChain rapidamente se tornou o padrão *de fato* para desenvolvedores que desejam construir aplicações robustas e “conscientes de dados” com LLMs. Ele não é apenas mais uma biblioteca; é um framework de orquestração completo. Seu objetivo é simplificar cada estágio do ciclo de vida de uma aplicação de IA, desde a prototipagem rápida até a implantação em produção:

* **Desenvolvimento Acelerado:** O LangChain oferece uma vasta coleção de componentes modulares e reutilizáveis. Isso inclui modelos (para interagir com diferentes LLMs), prompts (para gerenciar e otimizar as instruções), chains (para encadear operações complexas) e agentes (para dar autonomia aos LLMs, permitindo que eles tomem decisões e usem ferramentas). Essa modularidade permite que você construa aplicações complexas de forma muito mais rápida e eficiente, reutilizando blocos de construção testados e otimizados.
* **Consciência de Dados (Data-Awareness):** Um dos maiores desafios ao trabalhar com LLMs é conectá-los aos seus dados privados e em tempo real. O LangChain se destaca por facilitar a integração com diversas fontes de dados, como bancos de dados, APIs, documentos e sistemas de armazenamento de vetores. Isso é crucial para construir aplicações que não apenas geram texto, mas que também compreendem e interagem com o mundo real através dos seus próprios dados.
* **Agentes Autônomos e Ferramentas:** O framework eleva o LLM de um simples gerador de texto para um “motor de raciocínio”. Com o LangChain, você pode capacitar seu LLM a usar ferramentas externas (como motores de busca, calculadoras, APIs de terceiros) e a tomar decisões sobre qual ação executar para atingir um objetivo. Isso abre as portas para a criação de agentes verdadeiramente autônomos e sistemas multiagentes complexos.
* **Observabilidade e Avaliação em Produção:** Através de ferramentas complementares como o LangSmith, o LangChain permite inspecionar, monitorar e avaliar suas aplicações em tempo real. Isso é absolutamente crucial para identificar gargalos, depurar comportamentos inesperados e garantir a qualidade e a confiabilidade das suas aplicações de IA em ambientes de produção.
* **Implantação Simplificada:** O framework facilita a transformação de suas lógicas complexas em APIs prontas para produção, permitindo que você integre suas aplicações de IA em sistemas existentes com facilidade.

Em resumo, o LangChain nos dá os blocos de construção e a cola para criar aplicações que vão muito além de uma simples chamada de API para um LLM. Seu sucesso e abordagem pioneira inspiraram o surgimento de diversos outros frameworks fantásticos, como CrewAI, Microsoft Autogen, Pydantic AI e Praison AI, que hoje compõem um ecossistema rico e vibrante para o desenvolvimento de agentes. O LangChain se mantém como uma base sólida e flexível para a inovação em IA.

### **Uma Visão de Resolvedor de Problemas**

Antes de escrever a primeira linha de código com LangChain, quero te propor uma mudança de mentalidade. Muitas vezes, ficamos fascinados com a tecnologia pela tecnologia. Mas a verdadeira magia acontece quando usamos essa tecnologia para resolver um problema real, seja ele seu, da sua empresa ou de outra pessoa.

Antes de codificar, é preciso ter uma visão de resolvedor de problemas.

### **Como Aproveitar ao Máximo Este Livro**

Este livro foi desenhado para ser mais do que uma leitura passiva; é um convite à ação. Para extrair o máximo valor desta jornada, sugiro as seguintes abordagens:

1. **Mão na Massa:** A teoria é importante, mas a prática é onde o aprendizado se solidifica. Execute todos os exemplos de código. Altere-os, quebre-os e conserte-os. A experimentação é a chave para a maestria.
2. **Não Pule Capítulos:** Os conceitos são construídos de forma progressiva. Cada capítulo se baseia no anterior. Se algo parecer complexo, revise os capítulos anteriores antes de seguir em frente.
3. **Use o Gêmeo Digital:** Lembre-se do assistente de IA que mencionei na introdução? Ele é seu companheiro de estudo. Use-o para tirar dúvidas, explorar conceitos e até mesmo para debater ideias. Ele foi treinado com o conteúdo deste livro e pode oferecer insights adicionais.
4. **Participe da Comunidade:** O aprendizado é uma via de mão dupla. Compartilhe suas descobertas, faça perguntas e ajude outros. A comunidade LangChain é vibrante e acolhedora. Juntos, somos mais fortes.
5. **Adapte e Inove:** Os exemplos fornecidos são pontos de partida. Pense em como você pode adaptar as técnicas e os conceitos para seus próprios projetos e desafios. A inovação começa com a adaptação criativa.

Estou animado para ver o que você vai construir. Vamos começar?

Cada novo conhecimento que você adquirir neste livro não é apenas um conceito teórico; é uma nova ferramenta na sua caixa. Pense em um problema que te incomoda. Um processo manual e repetitivo no seu trabalho? A dificuldade de encontrar informações em uma base de dados interna gigante? A vontade de criar um assistente pessoal que realmente te entenda?

Eu, por exemplo, aprendi sobre visão computacional e frameworks como o YOLO não por um interesse puramente acadêmico, mas porque queria resolver um problema que me toca profundamente. Vendo a luta diária do meu pai com o Alzheimer, comecei a desenvolver um projeto para usar câmeras de segurança internas para monitorar idosos com demência, alertando familiares ou cuidadores sobre quedas ou situações de risco diretamente em seus celulares. O problema real me deu a motivação e o contexto para aprender a ferramenta.

Mantenha essa abordagem em mente enquanto avança pelos capítulos. Tente conectar cada conceito — Chains, Agents, RAG — a uma possível solução para um problema que importa para você. Essa abordagem transforma o aprendizado passivo em uma busca ativa por soluções, tornando a jornada muito mais significativa e eficaz.

No fundo, a programação e a filosofia compartilham um terreno comum: ambas buscam entender e moldar a realidade através da lógica e da abstração. Assim como um filósofo constrói argumentos para desvendar verdades, um programador constrói algoritmos para resolver problemas. A IA, nesse sentido, é a materialização de um pensamento filosófico antigo: a busca por inteligência e autonomia. Ao programar com IA, estamos não apenas escrevendo código, mas participando de um diálogo contínuo com a natureza da inteligência e da criação.

## Capítulo 1: Introdução ao LangChain — Fundamentos e Conceitos Essenciais

**Neste capítulo, você vai aprender:**

* O que é LangChain e como ele surgiu para resolver problemas reais na era da IA.
* Os conceitos fundamentais de Chains, Prompts, Models e Agents.
* Como o LangChain se posiciona no ecossistema de IA, comparado a outros frameworks.
* Seu primeiro código prático com LangChain, usando a sintaxe moderna da LCEL.

Este capítulo serve como a porta de entrada para o universo LangChain. Abordaremos a história e a motivação por trás do framework, seus componentes essenciais e como ele se encaixa no cenário atual da inteligência artificial. Ao final, você terá uma compreensão sólida dos pilares do LangChain e estará pronto para construir suas primeiras aplicações.

### O que é LangChain? Uma Breve História em Meio à Tempestade da IA

Antes de mergulharmos, quero que saiba: é normal se sentir um pouco sobrecarregado no início. O mundo da IA avança a passos largos, e o LangChain, como uma ferramenta poderosa, pode parecer complexo. Mas lembre-se, cada grande jornada começa com um primeiro passo. E você já deu o seu ao abrir este livro. Vamos desmistificar cada conceito, juntos. Respire fundo, e vamos nessa!

Para entender o LangChain, vamos simplificar. Imagine que você tem um supercomputador que sabe conversar, escrever e até criar coisas, mas ele está isolado. Ele não consegue acessar a internet, nem seus documentos, nem interagir com outros programas. O LangChain é como a “ponte” que conecta esse supercomputador (o Modelo de Linguagem, ou LLM) ao mundo real.

Ele surgiu no final de 2022, um pouco antes do ChatGPT “explodir” e mostrar a todos o poder dos LLMs. Naquela época, um engenheiro chamado Harrison Chase percebeu que, para construir aplicações realmente úteis com esses modelos, precisávamos de uma forma fácil de:

* **Conectá-los a informações externas:** Como fazer o LLM “ler” seus e-mails ou documentos?
* **Permitir que eles usem ferramentas:** Como fazer o LLM “clicar” em um botão ou “enviar” uma mensagem?
* **Organizar tarefas complexas:** Como fazer o LLM seguir uma série de passos para resolver um problema grande?

O LangChain foi a resposta para essas perguntas. Ele se tornou incrivelmente popular porque oferecia uma maneira de “orquestrar” os LLMs, transformando-os de meros geradores de texto em “cérebros” capazes de interagir com o mundo. Em pouco tempo, o projeto de código aberto cresceu tanto que se tornou uma empresa, a LangChain AI, atraindo grandes investimentos.

Em resumo, o LangChain é a ferramenta que nos permite construir aplicações de IA que vão muito além de uma simples conversa, conectando os LLMs a dados e ferramentas para resolver problemas do dia a dia.

### O Problema da Conversa Iterativa e o Nascimento das “Chains”

Se você já usou o ChatGPT, provavelmente já passou por este processo: você tem uma ideia, mas a primeira resposta do modelo não é exatamente o que você queria. Então, você começa um diálogo para refinar o resultado.

Imagine que você quer um poema. Sua conversa pode ser algo assim:

* **Você:** “Escreva um poema sobre a chuva.”
* *O modelo responde com um poema de quatro estrofes.*
* **Você:** “Gostei, mas está um pouco longo. Você pode deixar mais conciso, com apenas duas estrofes?”
* *O modelo responde com uma versão mais curta.*
* **Você:** “Perfeito. Agora, reescreva essa versão mais curta, mas adicione um sentimento de melancolia e a imagem de alguém olhando pela janela.”
* *O modelo finalmente entrega o poema que você tinha em mente.*

O que você acabou de fazer foi um **refinamento iterativo**. Você “encadeou” seus pensamentos, usando a saída de um passo como entrada para o próximo, para guiar o modelo até o resultado desejado. Esse processo é poderoso, mas é manual.

É exatamente aqui que a genialidade do LangChain se revela. O nome **LangChain** significa literalmente “Language Chain” (Chain de Linguagem). A ideia central do framework é permitir que nós, desenvolvedores, automatizemos esse processo de encadeamento.

Em vez de você refinar manualmente o poema, você poderia construir uma “Chain” no LangChain que faz isso programaticamente:

1. **Passo 1:** Um prompt que gera um poema sobre um tópico.
2. **Passo 2:** A saída do Passo 1 (o poema) é automaticamente enviada para um segundo prompt que o torna mais conciso.
3. **Passo 3:** A saída do Passo 2 (o poema conciso) é enviada para um terceiro prompt que adiciona um sentimento específico.

Isso é uma **Chain**: uma sequência de operações ou chamadas a modelos de linguagem para formar um pipeline inteligente. É o conceito fundamental que nos permite construir aplicações complexas que vão muito além de uma única pergunta e resposta.

### Exercício Prático: Hello, LangChain!

Vamos colocar a mão na massa com o nosso primeiro código. Este será o “Hello, World!” do LangChain. Ele vai demonstrar, da forma mais simples possível, o conceito de chain que acabamos de discutir. Para este e os demais exemplos do livro, usaremos os modelos Gemini do Google, especificamente o gemini-1.5-flash, que é incrivelmente rápido e oferece uma camada gratuita generosa para desenvolvedores.

* **Objetivo:** Fazer a primeira chamada a um LLM usando a sintaxe de chain do LangChain para ver o conceito em ação.
* **Nome do Arquivo:** exercicios/capitulo\_01/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv
* **Comando de Instalação:** uv add langchain langchain-google-genai python-dotenv

# exercicios/capitulo\_01/exercicio\_1/main.py  
  
import os   
from dotenv import load\_dotenv   
from langchain\_google\_genai import ChatGoogleGenerativeAI   
from langchain\_core.prompts import ChatPromptTemplate   
from langchain\_core.output\_parsers import StrOutputParser  
  
# Carrega as variáveis de ambiente (necessário para a chave do Google)   
# Certifique-se de ter um arquivo .env como explicado no Capítulo 2   
load\_dotenv()  
  
# 1. Crie um template de prompt.   
# Pense nisso como o "molde" para a sua pergunta.   
# A variável {topico} será preenchida dinamicamente.   
prompt\_template = ChatPromptTemplate.from\_template(   
 "Escreva uma única frase engraçada sobre o tópico: {topico}"   
)  
  
# 2. Inicialize o LLM.   
# Este é o "cérebro" que vai gerar a resposta. Usaremos o Gemini Flash.   
model = ChatGoogleGenerativeAI(model="gemini-1.5-flash")  
  
# 3. Crie um parser de saída.   
# Ele vai extrair apenas o texto da resposta do modelo.   
output\_parser = StrOutputParser()  
  
# 4. Crie a "Chain" usando o operador pipe (|).   
# Esta é a mágica da LangChain Expression Language (LCEL)!  
# O fluxo é: prompt -> modelo -> parser   
chain = prompt\_template | model | output\_parser  
  
# 5. Invoque a chain com um tópico.   
# O LangChain cuida de passar o resultado de um passo para o outro.   
print("Executando a chain...")   
resposta = chain.invoke({"topico": "desenvolvedores Python"})  
  
# Imprime o resultado final   
print("\nResposta da Chain:")   
print(resposta)

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_01/exercicio\_1/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_01/exercicio\_1/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_01\exercicio\_1\run.bat

**Saída Esperada (pode variar):**

Executando a chain...  
  
Resposta da Chain:   
Um desenvolvedor Python não tem medo de cobras, mas treme na base quando vê um erro de indentação.

Parabéns! Você acabou de executar sua primeira Chain. Observe a linha chain = prompt\_template | model | output\_parser. Essa sintaxe elegante, chamada **LangChain Expression Language (LCEL)**, é a representação visual do encadeamento que discutimos. É a base sobre a qual construiremos aplicações muito mais poderosas.

**Troubleshooting Comum:**

* **AuthenticationError ou GOOGLE\_API\_KEY não configurada:** Certifique-se de que você obteve sua GOOGLE\_API\_KEY do Google AI Studio e a adicionou corretamente ao seu arquivo .env na raiz do projeto. Lembre-se de que o arquivo .env não deve ser versionado no Git.
* **ModuleNotFoundError:** Verifique se todas as dependências (langchain, langchain-google-genai, python-dotenv) foram instaladas corretamente usando uv add ou pip install.
* **Erros de Conexão:** Problemas de rede ou limites de taxa da API podem causar erros. Tente novamente após alguns segundos ou verifique sua conexão com a internet.

### A Arquitetura Central: Os Componentes Essenciais

Agora que você já viu uma chain em ação, vamos formalizar os blocos de construção fundamentais. Pense neles como peças de Lego que podemos combinar de infinitas maneiras.

**1. Models (LLMs e Chat Models)**

O coração de qualquer aplicação LangChain é um modelo de linguagem. O LangChain fornece uma interface padronizada para interagir com dezenas de modelos diferentes, desde os da OpenAI (GPT-3.5, GPT-4), Google (Gemini), Anthropic (Claude), até modelos de código aberto disponíveis no Hugging Face. Isso significa que você pode trocar o modelo subjacente da sua aplicação com pouquíssimas alterações no código, o que é fantástico para experimentação e otimização de custos.

Existem dois tipos principais de modelos no LangChain:

* **LLMs:** Modelos que recebem uma string como entrada e retornam uma string como saída.
* **Chat Models:** Modelos que recebem uma lista de mensagens (“system”, “human”, “ai”) e retornam uma mensagem. A maioria das aplicações modernas utiliza Chat Models.

**2. Prompts**

Um prompt é a instrução que damos ao modelo de IA. O LangChain facilita a criação de prompts dinâmicos e reutilizáveis com **Prompt Templates**.

**3. Chains**

Uma Chain é uma sequência declarativa de componentes (Runnables) conectados via LCEL (|), formando pipelines que processam dados de entrada até a saída final. Use LCEL para compor fluxos simples e eficientes.

**4. Agents (Agentes)**

Agentes são componentes que usam LLMs para decidir dinamicamente quais ações tomar, escolhendo ferramentas e caminhos conforme o contexto e objetivo. Use Agents para tarefas que exigem flexibilidade e tomada de decisão autônoma.

### O Ecossistema LangChain: Mais que uma Biblioteca

O LangChain evoluiu para um ecossistema completo:

* **langchain-core:** Abstrações e interfaces base (Runnable, BaseMessage, etc.).
* **langchain-community:** Integrações mantidas pela comunidade.
* **langchain:** Pacote principal com chains, agents e estratégias de recuperação.
* **LangGraph:** Framework para orquestração de agentes e workflows baseados em grafos, ideal para fluxos complexos e multiagentes.
* **LangSmith:** Plataforma de observabilidade, depuração e avaliação para aplicações LangChain.

### Tabela 1: LangChain vs. Frameworks Concorrentes

Para situar o LangChain no cenário atual, é útil compará-lo com outras ferramentas populares.

| Framework | Foco Principal | Pontos Fortes | Caso de Uso Típico |
| --- | --- | --- | --- |
| **LangChain** | Orquestração de LLMs e Agentes | Flexibilidade, ecossistema de agentes, vastas integrações, LangGraph | Chatbots complexos, agentes autônomos, assistentes de código |
| **LlamaIndex** | Indexação e Preparação de Dados para RAG | Otimização de dados para RAG, conectores de dados, índices hierárquicos | Aplicações RAG sobre grandes volumes de documentos |
| **CrewAI** | Orquestração de Agentes Colaborativos | Foco em colaboração entre agentes, definição de papéis e tarefas, processos sequenciais | Equipes de agentes autônomos para pesquisa, escrita e análise |
| **Microsoft Autogen** | Framework de Multiagentes Conversacionais | Agentes conversáveis, flexibilidade na definição de padrões de interação | Simulações complexas, resolução de problemas em grupo, jogos |
| **Pydantic AI** | Geração de Saídas Estruturadas (JSON) | Integração com Pydantic, garantia de conformidade com schemas JSON | Extração de dados, saídas de API, integração com sistemas legados |
| **Praison AI** | Orquestração de Multi-Agentes LLM (Low-Code) | Facilidade de uso, customização, integração com outros frameworks, suporte a múltiplos LLMs | Automação de processos, chatbots, pesquisa multi-agente, análise de dados |

É importante notar que, embora o LangChain seja uma ferramenta poderosa e flexível, a escolha do framework ideal depende muito do caso de uso específico. LlamaIndex, por exemplo, brilha em cenários de RAG complexos, enquanto CrewAI e Microsoft Autogen oferecem abordagens mais especializadas para orquestração multiagente. Pydantic AI foca na garantia de saídas estruturadas, crucial para integração com sistemas legados. Praison AI, por sua vez, se destaca pela facilidade de uso e orquestração de multi-agentes com foco em low-code.

O LangChain se posiciona como uma ferramenta modular de propósito geral, permitindo a construção de uma vasta gama de aplicações de IA. A escolha entre esses frameworks envolve um trade-off entre flexibilidade, facilidade de uso e especialização. Enquanto frameworks como CrewAI e AutoGen oferecem soluções mais prontas para orquestração multiagente e colaboração, o LangChain fornece os blocos de construção fundamentais para criar soluções altamente customizadas e complexas, sendo a base para muitos desses outros frameworks. Compreender esses trade-offs é fundamental para escolher a ferramenta certa para o trabalho.

### Pontos Chave

* LangChain simplifica a orquestração de LLMs, permitindo a construção de aplicações complexas.
* O conceito de “Chain” automatiza sequências de chamadas a LLMs, tornando o desenvolvimento mais estruturado.
* Os componentes fundamentais (Models, Prompts, Chains, Agents) são blocos de construção modulares.
* A LCEL (LangChain Expression Language) é a forma moderna e eficiente de construir pipelines no LangChain.

### Resumo do Capítulo

Este capítulo serve como a porta de entrada para o universo LangChain. Abordamos a história, motivação, componentes essenciais e posicionamento do framework. Você está pronto para construir suas primeiras aplicações com LangChain.

### Teste seu Conhecimento

1. Qual foi o principal problema que o LangChain buscou resolver em sua criação?
   1. A falta de modelos de linguagem poderosos.
   2. A dificuldade de treinar novos LLMs.
   3. A necessidade de automatizar e estruturar sequências de chamadas para LLMs.
   4. A ausência de interfaces de chat como o ChatGPT.
2. O que é uma “Chain” no contexto do LangChain?
   1. Um tipo específico de LLM.
   2. Uma sequência de componentes (prompts, modelos, etc.) conectados para executar uma tarefa complexa.
   3. Uma ferramenta para buscar informações na internet.
   4. A interface de usuário de uma aplicação de IA.
3. Qual dos seguintes NÃO é um componente central da arquitetura LangChain?
   1. Models
   2. Prompts
   3. Agents
   4. Database
4. No exercício “Hello, LangChain!”, qual operador foi usado para conectar o prompt, o modelo e o parser?
   * + (adição)
   1. -> (seta)
   2. (pipe)
   3. & (e comercial)
5. Qual a principal diferença entre uma Chain e um Agent?
   1. Agents são mais rápidos que Chains.
   2. Chains usam múltiplos modelos, enquanto Agents usam apenas um.
   3. Chains seguem um fluxo predefinido, enquanto Agents usam um LLM para decidir dinamicamente qual ação tomar.
   4. Agents só podem ser usados para chatbots, enquanto Chains são de uso geral.

**Respostas:** 1. c 2. b 3. d 4. c 5. c

### Projeto Hands-on: Construindo um Chatbot Simples

Neste projeto, você vai integrar tudo o que aprendeu no Capítulo 1 para construir um chatbot simples que pode responder a perguntas básicas. Este será o seu primeiro passo para criar aplicações mais complexas com LangChain.

**Objetivo:** Criar um chatbot que interage com o usuário, mantendo um histórico de conversas e respondendo a perguntas gerais.

**Nome do Arquivo:** exercicios/capitulo\_01/exercicio\_2/main.py

**Dependências:** langchain, langchain-google-genai, python-dotenv, langchain-community

**Comando de Instalação:** uv add langchain langchain-google-genai python-dotenv langchain-community

# exercicios/capitulo\_01/exercicio\_2/main.py  
  
import os  
from dotenv import load\_dotenv  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder  
from langchain\_core.output\_parsers import StrOutputParser  
from langchain\_core.messages import HumanMessage, AIMessage  
  
# Carrega as variáveis de ambiente  
load\_dotenv()  
  
# Inicializa o LLM  
model = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0.7)  
  
# Cria um template de prompt com um placeholder para o histórico de chat  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "Você é um assistente prestativo. Responda às perguntas de forma concisa e útil."),  
 MessagesPlaceholder(variable\_name="chat\_history"),  
 ("human", "{input}"),  
])  
  
# Cria o parser de saída  
output\_parser = StrOutputParser()  
  
# Cria a chain  
chain = prompt | model | output\_parser  
  
# Histórico de chat (simulado por enquanto)  
chat\_history = []  
  
print("Chatbot Simples. Digite 'sair' para encerrar.")  
  
while True:  
 user\_input = input("Você: ")  
 if user\_input.lower() == 'sair':  
 break  
  
 # Invoca a chain com a entrada do usuário e o histórico de chat  
 response = chain.invoke({  
 "input": user\_input,  
 "chat\_history": chat\_history  
 })  
  
 print(f"Bot: {response}")  
  
 # Atualiza o histórico de chat  
 chat\_history.append(HumanMessage(content=user\_input))  
 chat\_history.append(AIMessage(content=response))  
  
print("Chatbot encerrado.")

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_01/exercicio\_2/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_01/exercicio\_2/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_01\exercicio\_2\run.bat

**Troubleshooting Comum:**

* **AuthenticationError ou GOOGLE\_API\_KEY não configurada:** Certifique-se de que você obteve sua GOOGLE\_API\_KEY do Google AI Studio e a adicionou corretamente ao seu arquivo .env na raiz do projeto. Lembre-se de que o arquivo .env não deve ser versionado no Git.
* **ModuleNotFoundError:** Verifique se todas as dependências (langchain, langchain-google-genai, python-dotenv, langchain-community) foram instaladas corretamente usando uv add ou pip install.
* **Erros de Conexão:** Problemas de rede ou limites de taxa da API podem causar erros. Tente novamente após alguns segundos ou verifique sua conexão com a internet.
* **Comportamento Inesperado do Chatbot:** Se o chatbot não estiver mantendo o contexto ou respondendo de forma estranha, revise o system prompt e a forma como o chat\_history está sendo passado para a chain. A clareza das instruções no prompt é fundamental.

## Capítulo 2: Configurando o Ambiente de Desenvolvimento Python para LangChain

**Neste capítulo, você vai aprender:**

* Como configurar um ambiente de desenvolvimento Python robusto e seguro para projetos de IA com LangChain.
* Vantagens do Linux/WSL, pyenv, zsh, Oh My Zsh e chaves SSH para produtividade e segurança.
* Gerenciamento moderno de dependências com uv e pyproject.toml.
* Práticas recomendadas para proteger chaves de API e variáveis de ambiente.
* Considerações sobre hardware para rodar LLMs locais.
* Exercícios práticos para instalar ferramentas essenciais e validar o ambiente.

## Vamos começar?

Antes de mergulharmos no mundo do LangChain, é fundamental garantir que nosso ambiente de desenvolvimento esteja pronto para a batalha. Configurar o ambiente certo não é apenas uma questão de conveniência, mas também de eficiência e segurança. Neste capítulo, vamos preparar o terreno para que você possa se concentrar no que realmente importa: construir aplicações incríveis com IA.

Então, antes de começarmos, respire fundo. Pegue um café (ou um chá, no meu caso, já que ironicamente não sou fã de café) e vamos passar por isso juntos, passo a passo. Lembro-me de um colega que passou horas depurando um erro complexo, apenas para descobrir que era um ponto e vírgula faltando. Ou, pior, um espaço a mais na indentação em Python. A máquina é implacável com a sintaxe, mas a satisfação de encontrar o erro é indescritível! A paciência que você exercita aqui será sua maior aliada em toda a jornada com IA. Prometo que, ao final deste capítulo, você terá uma base sólida e organizada para construir todos os projetos incríveis que virão.

### **Por que um bom ambiente de desenvolvimento é crucial?**

Um ambiente de desenvolvimento bem configurado é a fundação sobre a qual você construirá suas aplicações. Ele não apenas facilita o trabalho diário, mas também garante que você possa escalar seus projetos de forma eficiente e segura.

Basicamente, é importante que você desenvolva em um ambiente que seja o mais próximo possível do ambiente de produção. Isso reduz problemas de compatibilidade e garante que o que funciona no seu computador também funcionará quando você for para o *deploy*.

Aqui estão alguns outros pontos-chave: \* **Consistência:** Um ambiente padronizado reduz problemas de compatibilidade e garante que todos os membros da equipe estejam na mesma página. \* **Eficiência:** Ferramentas como pyenv e Oh My Zsh aceleram o fluxo de trabalho, permitindo que você se concentre no código, não na configuração. \* **Segurança:** Proteger suas chaves de API e variáveis de ambiente é essencial para evitar vazamentos de dados e garantir a integridade do seu projeto. \* **Hardware adequado:** Considerar as especificações do seu computador é crucial, especialmente ao trabalhar com modelos de linguagem grandes (LLMs) que exigem recursos significativos. Embora esse aspecto seja mais relevante quando você for rodar LLMs locais, é importante ter em mente que um ambiente bem configurado pode fazer a diferença entre um projeto que roda suavemente e outro que trava constantemente.

### **Meu Ambiente de Batalha: Por que Linux e Ferramentas de Linha de Comando**

Sei que muitos desenvolvedores, especialmente no mundo corporativo, estão acostumados com o Windows e o PowerShell. E eu entendo perfeitamente, são ferramentas poderosas e familiares. No entanto, para o tipo de desenvolvimento que faremos aqui, e para o desenvolvimento de software em geral, eu recomendo fortemente que você abrace o ambiente Linux.

Para alguns, um terminal de computador é uma tela preta, ainda que colorida, é fria e sem vida. Para mim, quando estou no meu terminal Linux Kali com o shell Zsh todo customizado, me sinto como se estivesse tocando o solo de guitarra de “Hotel California” dos Eagles. Existe uma fluidez, uma arte, um prazer em fazer as ferramentas responderem exatamente como você quer, com o mínimo de esforço. É uma dança entre o homem e a máquina.

**Por que Linux?** A grande maioria das ferramentas de desenvolvimento, servidores de produção, contêineres (Docker) e tecnologias de nuvem rodam nativamente em Linux. Desenvolver em um ambiente semelhante ao de produção economiza uma quantidade enorme de dores de cabeça com compatibilidade, permissões de arquivo e pequenas diferenças que podem quebrar sua aplicação quando você for para o *deploy*.

**“Mas Igor, eu uso Windows!”** Sem problemas! A melhor invenção da Microsoft para desenvolvedores nos últimos anos foi o **Windows Subsystem for Linux (WSL)**. Ele permite que você rode uma distribuição Linux completa diretamente no seu Windows, com integração total. É o melhor dos dois mundos. Pessoalmente, eu uso o Kali Linux, que está disponível gratuitamente na Microsoft Store, por sua robustez e conjunto de ferramentas, mas distribuições como Ubuntu também são excelentes escolhas.

### **Gerenciando Versões do Python como um Profissional: pyenv**

Outro ponto crucial é o gerenciamento de versões do Python. Um projeto pode precisar do Python 3.11, outro do 3.12, e o seu sistema operacional pode depender de uma versão específica para funcionar. Instalar múltiplas versões globalmente é uma receita para o desastre.

A solução para isso é o **pyenv**. Ele é uma ferramenta fantástica que permite instalar e gerenciar múltiplas versões do Python no seu espaço de usuário, sem interferir com o Python do sistema. Com um simples comando, você pode definir qual versão do Python usar globalmente, por pasta de projeto ou até mesmo por sessão de terminal.

**Vantagens do pyenv:**

* **Isolamento:** Cada versão do Python é instalada em seu próprio diretório, evitando conflitos.
* **Flexibilidade:** Teste seu código em diferentes versões do Python com facilidade.
* **Consistência:** Garanta que toda a sua equipe use exatamente a mesma versão do Python para um projeto.

Exercício Prático: Instalando pyenv e Python no Kali Linux (WSL)  
Este script automatiza a instalação de todas as dependências necessárias, do pyenv e da versão mais recente do Python no Kali Linux.

* **Objetivo:** Preparar um ambiente Linux robusto com a versão correta do Python gerenciada pelo pyenv.
* **Nome do Arquivo:** setup\_python\_kali.sh
* **Dependências:** git, curl, build-essential e outras dependências de compilação.
* **Comando de Execução:** setup\_python\_kali.sh

#!/bin/bash  
  
# Script DEFINITIVO para instalar Python 3.12.9 no Kali Linux com pyenv   
# Versão 3: Lida com a ausência do comando 'gpg'.  
  
# O comando 'set -e' garante que o script pare se algum comando falhar.   
set -e  
  
echo "--- PASSO 1: Corrigindo o 'apt' (Lidando com a falta de 'gpg') ---"   
# Temporariamente, dizemos ao apt para confiar no repositório sem verificar a assinatura.   
# ISSO É INSEGURO, mas necessário para quebrar o ciclo e poder instalar o gpg.   
echo "deb [trusted=yes] http://http.kali.org/kali kali-rolling main contrib non-free" | sudo tee /etc/apt/sources.list  
  
echo ""   
echo "--- PASSO 2: Instalando 'gpg' e 'wget' ---"   
sudo apt update   
# Agora que o apt funciona (de forma insegura), instalamos o gnupg (que contém o gpg) e o wget.   
sudo apt install -y gnupg wget  
  
echo ""   
echo "--- PASSO 3: Consertando a segurança do 'apt' DE FORMA PERMANENTE ---"   
# 3.1: Agora que temos o 'gpg', podemos baixar e instalar a chave de segurança oficial.   
wget -q -O - https://archive.kali.org/archive-key.asc | sudo gpg --dearmor -o /usr/share/keyrings/kali-archive-keyring.gpg  
  
# 3.2: Reescrevemos o arquivo de repositórios para o modo SEGURO, forçando a verificação com a chave que acabamos de baixar.   
echo "deb [signed-by=/usr/share/keyrings/kali-archive-keyring.gpg] http://http.kali.org/kali kali-rolling main contrib non-free" | sudo tee /etc/apt/sources.list  
  
echo "Segurança do APT restaurada."  
  
echo ""   
echo "--- PASSO 4: Atualizando pacotes (agora de forma segura) e instalando TODAS as dependências ---"   
sudo apt update   
sudo apt install -y build-essential libssl-dev zlib1g-dev \  
libbz2-dev libreadline-dev libsqlite3-dev curl llvm \  
libncurses5-dev libncursesw5-dev xz-utils tk-dev libffi-dev \  
liblzma-dev python3-openssl git  
  
echo ""   
echo "--- PASSO 5: Instalando o pyenv (se necessário) ---"   
if [ ! -d "$HOME/.pyenv" ]; then   
 curl https://pyenv.run | bash   
else   
 echo "O pyenv já está instalado. Pulando a instalação."   
fi  
  
echo ""   
echo "--- PASSO 6: Configurando o ambiente do pyenv ---"   
if ! grep -qF 'PYENV\_ROOT' ~/.zshrc; then   
 echo -e '\n# Configuração do Pyenv' >> ~/.zshrc   
 echo 'export PYENV\_ROOT="$HOME/.pyenv"' >> ~/.zshrc   
 echo 'command -v pyenv >/dev/null || export PATH="$PYENV\_ROOT/bin:$PATH"' >> ~/.zshrc   
 echo 'eval "$(pyenv init -)"' >> ~/.zshrc   
fi   
export PYENV\_ROOT="$HOME/.pyenv"   
export PATH="$PYENV\_ROOT/bin:$PATH"   
eval "$(pyenv init -)"  
  
echo ""   
echo "--- PASSO 7: Instalando o Python 3.12.9 (Isso pode levar alguns minutos) ---"   
pyenv install 3.12.9  
  
echo ""   
echo "--- PASSO 8: Definindo o Python 3.12.9 como padrão global ---"   
pyenv global 3.12.9  
  
echo "--- PASSO 9: Atualizando o pip"   
pip install --upgrade pip  
  
echo ""   
echo "-------------------------------------------------------------"   
echo " SUCESSO! Ambiente corrigido e Python 3.12.9 instalado."   
echo "-------------------------------------------------------------"   
echo ""   
echo "Feche e reabra seu terminal para que tudo funcione corretamente, então verifique com:"   
echo "python --version"   
echo ""

### Hands-on: Exercício 1 — Instalando pyenv e Python no Kali Linux (WSL)

**Comando de Execução (Linux/macOS):**

chmod +x setup\_python\_kali.sh && ./setup\_python\_kali.sh

**Comando de Execução (Windows):** Execute o script acima dentro do WSL (Ubuntu/Kali).

**Saída Esperada (pode variar):**

SUCESSO! Ambiente corrigido e Python 3.12.9 instalado.  
python --version  
Python 3.12.9

### **Um Terminal com Superpoderes: zsh e Oh My Zsh**

Sou muito exigente com meu terminal. É onde passo boa parte do meu dia, e ele precisa ser rápido, inteligente e visualmente agradável. Por isso, a primeira coisa que faço em qualquer ambiente novo é substituir o bash padrão pelo zsh e turbiná-lo com o framework **Oh My Zsh**.

O zsh por si só já é um shell mais poderoso, mas com Oh My Zsh e alguns plugins, ele se transforma. Meus plugins indispensáveis são:

* **zsh-syntax-highlighting:** Colore os comandos em tempo real, te mostrando se um comando existe ou se você digitou algo errado antes mesmo de apertar Enter.
* **zsh-autosuggestions:** Sugere comandos com base no seu histórico enquanto você digita, como um autocompletar mágico.

Exercício Prático: Instalando o Terminal Perfeito  
Este script instala o zsh, o Oh My Zsh e os plugins que eu uso.

* **Objetivo:** Configurar um terminal moderno e productivo.
* **Nome do Arquivo:** setup\_zsh.sh
* **Dependências:** zsh, git, curl
* **Comando de Execução:** setup\_zsh.sh

#!/bin/bash  
  
# Atualiza os pacotes e instala o Zsh   
sudo apt update   
sudo apt install -y zsh  
  
# Define o Zsh como o shell padrão para o usuário atual   
# Pode ser necessário inserir a senha aqui, dependendo das permissões do sudo   
sudo usermod -s /usr/bin/zsh $(whoami)  
  
# Instala o Oh My Zsh sem iniciar um novo shell   
export RUNZSH=no   
sh -c "$(curl -fsSL https://raw.githubusercontent.com/ohmyzsh/ohmyzsh/master/tools/install.sh)" "" --unattended  
  
# Clona os plugins do zsh-syntax-highlighting e zsh-autosuggestions   
git clone https://github.com/zsh-users/zsh-syntax-highlighting.git ${ZSH\_CUSTOM:-~/.oh-my-zsh/custom}/plugins/zsh-syntax-highlighting   
git clone https://github.com/zsh-users/zsh-autosuggestions ${ZSH\_CUSTOM:-~/.oh-my-zsh/custom}/plugins/zsh-autosuggestions  
  
# Edita a linha de plugins no ~/.zshrc usando sed   
sed -i 's/^plugins=(.\*/plugins=(git zsh-syntax-highlighting zsh-autosuggestions)/' ~/.zshrc  
  
source ~/.zshrc  
  
echo "Instalação e configuração do Zsh e Oh My Zsh concluídas!"   
echo "Por favor, reinicie seu terminal ou faça logout e login para que as alterações entrem em vigor."  
  
# Inicia uma nova sessão Zsh para aplicar as alterações (opcional)   
exec zsh

### Hands-on: Exercício 2 — Instalando o Terminal Perfeito (zsh + Oh My Zsh)

**Comando de Execução (Linux/macOS):**

chmod +x setup\_zsh.sh && ./setup\_zsh.sh

**Comando de Execução (Windows):** Execute o script acima dentro do WSL.

**Saída Esperada (pode variar):**

Instalação e configuração do Zsh e Oh My Zsh concluídas!

### **Segurança e Conveniência com Git: Chaves SSH**

Quando você clona, faz *push* ou *pull* de um repositório no GitHub, você precisa se autenticar. A forma mais comum é via HTTPS, que te pede o nome de usuário e senha (ou um token de acesso pessoal) toda vez. É seguro, mas repetitivo.

Uma forma muito mais segura e conveniente é usar **chaves SSH**. Você gera um par de chaves: uma pública, que você adiciona à sua conta do GitHub, e uma privada, que fica segura no seu computador. Quando você se conecta, o Git usa esse par de chaves para te autenticar sem que você precise digitar nada.

**Vantagens de usar SSH:**

* **Conveniência:** Chega de digitar senhas. Uma vez configurado, é automático.
* **Segurança Aprimorada:** Chaves SSH são criptograficamente muito mais fortes que senhas. É praticamente impossível alguém adivinhar sua chave privada.
* **Gerenciamento:** Você pode ter múltiplas chaves para diferentes máquinas e revogar o acesso de uma delas a qualquer momento sem afetar as outras.

**Exercício Prático: Gerando sua Chave SSH para o GitHub**

* **Objetivo:** Criar um par de chaves SSH e exibi-lo para que possa ser adicionado ao GitHub.
* **Nome do Arquivo:** generate\_ssh\_key.sh
* **Dependências:** openssh-client
* **Comando de Execução:** generate\_ssh\_key.sh

#!/bin/bash   
# Substitua o email pelo seu email do GitHub   
ssh-keygen -t rsa -b 4096 -C "igor@igormedeiros.com.br"  
  
# Inicia o ssh-agent em background   
eval "$(ssh-agent -s)"  
  
# Adiciona sua chave SSH privada ao ssh-agent   
ssh-add ~/.ssh/id\_rsa  
  
# Exibe a chave pública para que você possa copiá-la   
echo "Copie a chave pública abaixo e cole nas configurações do seu GitHub:"   
cat ~/.ssh/id\_rsa.pub

### Hands-on: Exercício 3 — Gerando sua Chave SSH para o GitHub

**Comando de Execução (Linux/macOS):**

chmod +x generate\_ssh\_key.sh && ./generate\_ssh\_key.sh

**Comando de Execução (Windows):** Execute o script acima dentro do WSL.

**Saída Esperada (pode variar):**

Copie a chave pública abaixo e cole nas configurações do seu GitHub:  
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAr...

### **Gerenciando Dependências: A Revolução do uv e pyproject.toml**

Agora que temos o Python certo e um terminal turbinado, precisamos falar sobre como gerenciar as bibliotecas do nosso projeto (as dependências). Por muito tempo, o mundo Python viveu com o arquivo requirements.txt e o comando pip. Funcionava, mas era lento e, às vezes, levava a inconsistências.

A comunidade Python, sempre em evolução, nos deu uma solução muito mais moderna e robusta. A conversa começou com a PEP 518, que introduziu o arquivo **pyproject.toml**. A ideia era simples: ter um único arquivo para declarar as dependências de construção do projeto. Isso resolveu o problema de “como instalar as ferramentas para construir o seu projeto?”. Depois, a PEP 621 expandiu isso, permitindo que a gente definisse quase todas as metadados do projeto (nome, versão, autor, e claro, as dependências) nesse mesmo arquivo. O pyproject.toml se tornou o padrão para projetos Python modernos.

Mas faltava uma peça: uma ferramenta que fosse rápida e inteligente para ler esse arquivo e gerenciar nosso ambiente. E então, os criadores do ruff (um linter de Python absurdamente rápido) nos deram o **uv**.

uv é um gerenciador de pacotes e ambientes virtuais escrito em Rust. E, meu amigo, ele é rápido. Coisas que levavam minutos com pip agora levam segundos. Ele foi projetado para ser um substituto direto para pip e virtualenv, mas com uma performance de 10 a 100 vezes melhor.

**Por que usar uv?**

* **Velocidade:** É a maior vantagem. Instalar e resolver dependências é incrivelmente rápido.
* **Tudo em um:** uv cria e gerencia ambientes virtuais automaticamente. Você não precisa mais de python -m venv e pip como ferramentas separadas.
* **Cache Inteligente:** Ele usa um cache global para evitar baixar a mesma dependência várias vezes, economizando disco e tempo.
* **Compatibilidade:** Ele entende pyproject.toml e também os antigos requirements.txt, então a migração é super tranquila.

A partir de agora, todos os nossos comandos de instalação usarão uv. Ele vai criar um ambiente virtual para nós na primeira vez que adicionarmos uma dependência e manter tudo organizado no pyproject.toml.

### Hands-on: Exercício 4 — Gerenciando Dependências com uv

**Comando de Execução (Linux/macOS/WSL):**

uv add langchain

**Saída Esperada (pode variar):**

Added langchain to pyproject.toml

### **Gerenciando Segredos: Chaves de API e Variáveis de Ambiente**

Para usar modelos de IA como os do Google Gemini, você precisará de uma **chave de API (API Key)**. É extremamente importante que você **nunca, jamais, em hipótese alguma**, coloque sua chave de API diretamente no seu código-fonte, especialmente se você planeja compartilhar esse código ou versioná-lo com o Git. Isso seria como deixar a chave da sua casa debaixo do tapete da porta.

A prática correta é usar **variáveis de ambiente**. A biblioteca python-dotenv nos ajuda a carregar essas variáveis de um arquivo para o nosso ambiente.

Como obter sua chave de API gratuita do Google Gemini:  
O Google oferece uma camada gratuita muito generosa para desenvolvedores que querem experimentar a API Gemini. A maneira mais fácil de obter uma chave é através do Google AI Studio.

1. Vá para aistudio.google.com/apikey.
2. Faça login com sua conta do Google.
3. Clique em “Create API key in new project”.
4. Copie a chave gerada. É isso! Você não precisa configurar um projeto complexo no Google Cloud ou adicionar um cartão de crédito para começar.

**Configurando a chave no seu projeto:**

1. **Crie um arquivo .env:** Na raiz da pasta do seu projeto, crie um arquivo chamado .env.
2. **Adicione sua chave de API ao arquivo .env:** Abra o arquivo .env e adicione sua chave da seguinte forma:  
   GOOGLE\_API\_KEY=“sua\_chave\_api\_aqui”
3. **Ignore o arquivo .env no Git:** Para garantir que você nunca envie acidentalmente seus segredos para um repositório público, crie um arquivo chamado .gitignore na raiz do seu projeto e adicione a seguinte linha a ele:  
   # Ambiente virtual  
   .venv/

# Arquivo de segredos

* .env

# Cache do Python

* **pycache**/

É fundamental ir além de apenas esconder as chaves. Adote o **princípio do menor privilégio**: suas chaves de API devem ter apenas as permissões mínimas necessárias para a tarefa que irão executar.

Por exemplo, se uma chave só precisa ler dados, ela não deve ter permissão para escrever ou deletar. Para ambientes de produção, considere o uso de serviços de gerenciamento de segredos como Azure KeyVault ou AWS Secret Manager.

É crucial entender que o ambiente de estudos e prototipagem, onde a conveniência é prioridade, difere significativamente de um ambiente de produção, que exige rigorosas práticas de segurança corporativa, como auditorias regulares, controle de acesso baseado em função (RBAC) e monitoramento contínuo. Além disso, a **rotação periódica de chaves** é uma boa prática de segurança. Defina um cronograma para gerar novas chaves e invalidar as antigas, minimizando o risco em caso de vazamento.

### **Considerações sobre Hardware**

Antes de encerrar este capítulo sobre ambiente, é importante tocar em um ponto que muitos desenvolvedores de IA enfrentam: o hardware. Especialmente se você planeja experimentar com modelos de linguagem locais (como os que rodam via Ollama, que abordaremos mais adiante), a capacidade da sua máquina se torna um fator crucial.

Eu, por exemplo, tenho um PC servidor de LLM em casa. É uma máquina modesta, com uma RTX 4060 de 8GB de VRAM. Para muitos, 8GB pode parecer pouco, e de fato, limita o tamanho dos modelos que consigo rodar eficientemente (geralmente até modelos de 8 bilhões de parâmetros). Mas mesmo com essa configuração, consigo gerar respostas a uma taxa de 40+ tokens por segundo, o que é incrivelmente rápido para experimentação e desenvolvimento local. Essa experiência me ensinou que não é preciso ter um supercomputador para começar a explorar o mundo dos LLMs locais, mas entender as limitações do seu hardware é fundamental para gerenciar as expectativas e otimizar seus experimentos.

### Hands-on: Exercício — Hello, Ambiente Python!

* **Objetivo:** Validar o ambiente Python e executar o primeiro código com LangChain.
* **Nome do Arquivo:** exercicios/capitulo\_02/main.py
* **Dependências:** langchain, python-dotenv
* **Comando de Instalação:**

uv add langchain python-dotenv

# exercicios/capitulo\_02/exercicio\_1/main.py  
import os  
from dotenv import load\_dotenv  
from langchain\_core.prompts import ChatPromptTemplate  
  
load\_dotenv()  
prompt = ChatPromptTemplate.from\_template("Diga olá para o mundo do LangChain!")  
print(prompt.format())

**Comando de Execução (Linux/macOS):**

chmod +x exercicios/capitulo\_02/exercicio\_1/run.sh  
./exercicios/capitulo\_02/exercicio\_1/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_02\exercicio\_1\run.bat

**Saída Esperada (pode variar):**

Diga olá para o mundo do LangChain!

**Troubleshooting Comum:** \* ModuleNotFoundError: Verifique se todas as dependências foram instaladas corretamente usando uv add. \* Problemas de permissão: Use chmod +x para scripts .sh. \* Variáveis de ambiente não carregadas: Certifique-se de que o arquivo .env está na raiz do projeto.

### Pontos Chave

* Um ambiente de desenvolvimento bem configurado (Linux/WSL, pyenv, zsh) é crucial para produtividade em IA.
* Gerenciamento de dependências com uv e pyproject.toml oferece velocidade e consistência.
* A segurança das chaves de API (variáveis de ambiente, .env, .gitignore) é fundamental para qualquer projeto de IA.
* Compreender as limitações de hardware é importante ao trabalhar com LLMs locais.

### Resumo do Capítulo

Neste capítulo, montamos um ambiente de desenvolvimento Python profissional, robusto e seguro, preparando o terreno para construir aplicações de IA de alta qualidade.

* Ambiente Linux: Vantagens de desenvolver em Linux (via WSL) para garantir compatibilidade com ferramentas e servidores de produção.
* Gerenciamento de Versões com pyenv: Instalação e uso do pyenv para múltiplas versões do Python sem conflitos.
* Terminal e Git: Terminal turbinado com zsh e Oh My Zsh, chaves SSH para interagir com o GitHub de forma segura.
* Gerenciamento de Dependências com uv: Evolução do gerenciamento de pacotes em Python, adoção do uv por sua velocidade e simplicidade.
* Gerenciamento de Segredos: Passo a passo para obter e configurar uma chave gratuita do Google AI Studio usando um arquivo .env.

|  |
| --- |
| ### Teste seu Conhecimento |
| 1. Qual é a principal vantagem de usar o WSL (Windows Subsystem for Linux) para desenvolvimento Python? a) Permite rodar aplicativos Windows no Linux b) Proporciona um ambiente Linux nativo dentro do Windows, facilitando compatibilidade e produtividade c) Melhora a performance do Windows d) Instala automaticamente todas as dependências Python |
| 2. Para que serve a ferramenta pyenv? a) Gerenciar ambientes virtuais b) Atualizar pacotes Python automaticamente c) Instalar e gerenciar múltiplas versões do Python sem conflitos d) Proteger variáveis de ambiente |
| 3. Qual arquivo é o padrão moderno para definir as dependências e metadados de um projeto Python? a) requirements.txt b) setup.py c) environment.yml d) pyproject.toml |
| 4. Por que é recomendado usar chaves SSH em vez de HTTPS para interagir com o GitHub? a) HTTPS é mais rápido b) Chaves SSH oferecem mais segurança e praticidade, evitando digitação de senhas c) SSH permite editar arquivos diretamente no GitHub d) HTTPS não funciona em ambientes Linux |
| 5. Qual comando você usaria com uv para adicionar uma nova dependência ao projeto? a) uv install b) pip add c) uv add d) python -m uv |

**Respostas:**  
1. b  
2. c  
3. d  
4. b  
5. c

### Projeto Hands-on: Melhorando o Chatbot Simples

Crie um projeto Python que utilize LangChain para construir um chatbot simples. Use uv para gerenciar dependências, proteja sua chave de API com .env e documente todos os comandos usados. Experimente rodar o projeto em diferentes versões do Python usando pyenv e compartilhe o repositório via SSH no GitHub.

## Capítulo 3: Manipulação de Prompts e Modelos de Linguagem com LangChain

**Neste capítulo, você vai aprender:**

* Os fundamentos da engenharia de prompts e como ela impacta a qualidade das respostas dos LLMs.
* Como criar e utilizar PromptTemplates dinâmicos e reutilizáveis no LangChain.
* Integração de modelos de linguagem (LLMs) com prompts usando a LCEL.
* Exercício prático de tradutor multilíngue com exemplos de código, instruções de execução e troubleshooting.
* Tabelas, fluxogramas, dicas avançadas e teste de conhecimento para consolidar o aprendizado.
* Como criar prompts para diferentes tipos de tarefas: geração, classificação, extração, tradução e sumarização.
* Estratégias para depuração e avaliação de prompts.
* Boas práticas para modularização e reuso de templates em projetos maiores.

### A Arte e a Ciência da Engenharia de Prompts

Se os modelos de linguagem são como gênios poderosos, os prompts são os nossos pedidos. A forma como fazemos o pedido — as palavras que usamos, a estrutura que damos, o contexto que fornecemos — determina drasticamente a qualidade da resposta que recebemos. Isso é o que chamamos de **Engenharia de Prompts**: a disciplina de projetar e otimizar as instruções dadas aos modelos de linguagem para obter os resultados desejados.

Não se trata de “adivinhar” as palavras mágicas. É uma disciplina que mistura criatividade, lógica e experimentação. Um bom prompt é claro, conciso e específico. Ele guia o modelo, em vez de deixá-lo vagando. O LangChain nos oferece ferramentas para elevar nossa engenharia de prompts de simples strings para templates dinâmicos e reutilizáveis.

**Dicas Avançadas:** - Teste diferentes variações de prompts e compare resultados usando LangSmith ou logs. - Use exemplos (few-shot) dentro do prompt para orientar o modelo em tarefas complexas. - Adote padrões como Chain-of-Thought para estimular raciocínio passo a passo. - Documente e versiona seus templates para facilitar manutenção e evolução do projeto.

### Templates de Prompt: Reutilização e Dinamismo

Imagine que você está construindo um assistente que resume artigos. Uma abordagem ingênua seria escrever o prompt toda vez no código:

# Abordagem ingênua (não faça isso!)  
artigo = "..." # texto longo do artigo  
prompt\_texto = f"Resuma o seguinte artigo em três pontos principais: {artigo}"

Isso funciona, mas é inflexível e ruim de manter. E se você quiser mudar o estilo do resumo? Ou o número de pontos? Você teria que caçar e alterar essa string em todo o seu código.

É aqui que os PromptTemplate do LangChain entram em cena. Eles nos permitem criar um modelo para nossos prompts, com espaços reservados (variáveis) que preenchemos depois.

Vamos ver como funciona. A classe principal para isso é a ChatPromptTemplate, projetada para os modernos modelos de chat.

from langchain\_core.prompts import ChatPromptTemplate  
  
# Criando um template para um sistema de resumo  
template\_string = """Você é um assistente especialista em resumir textos.  
Resuma o texto a seguir em {numero\_de\_pontos} pontos principais, em um tom {tom}.  
  
Texto:  
{texto\_do\_artigo}  
"""  
  
prompt\_template = ChatPromptTemplate.from\_template(template\_string)  
  
# Agora, podemos usar o template para formatar um prompt dinamicamente  
artigo\_exemplo = "A inteligência artificial está transformando indústrias, desde a saúde até as finanças, automatizando tarefas e gerando novos insights."  
prompt\_formatado = prompt\_template.format(  
 numero\_de\_pontos="dois",  
 tom="profissional",  
 texto\_do\_artigo=artigo\_exemplo  
)  
  
print(prompt\_formatado)

A saída será o prompt completo, pronto para ser enviado ao modelo:

text=‘Você é um assistente especialista em resumir textos.\nResuma o texto a seguir em dois pontos principais, em um tom profissional.\n\nTexto:\nA inteligência artificial está transformando indústrias, desde a saúde até as finanças, automatizando tarefas e gerando novos insights.\n’

A beleza disso é a modularidade. Podemos facilmente mudar o número de pontos, o tom ou o artigo sem tocar na estrutura do template.

**Outros usos de PromptTemplates:** - Tradução automática entre idiomas. - Geração de respostas personalizadas para diferentes perfis de usuário. - Extração de informações estruturadas (ex: dados em JSON). - Classificação de sentimentos ou tópicos.

A engenharia de prompts é sobre encontrar a combinação de palavras e estrutura que faz o modelo responder de forma precisa e criativa. Um prompt bem construído resulta em respostas melhores e mais úteis.

Neste capítulo, vamos aprender a afinar nossos templates e técnicas para criar prompts que realmente funcionam.

### Integrando Modelos de Linguagem (LLMs)

Um prompt, por si só, é apenas texto. Ele precisa ser enviado a um modelo para ganhar vida. Vamos usar o Gemini do Google.

# Supondo que você já configurou seu ambiente como no Capítulo 2  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
  
# Inicializa o modelo  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.7)  
# 'temperature' é um parâmetro que controla a criatividade da resposta.  
# 0 é mais determinístico, 1 é mais criativo.

Para conectar o prompt\_template com o llm, usamos a **LangChain Expression Language (LCEL)**, com o operador pipe (|):

# Conectando o prompt ao modelo usando LCEL  
chain = prompt\_template | llm  
  
# Agora, podemos invocar a "chain" diretamente com as variáveis  
response = chain.invoke({  
 "numero\_de\_pontos": "três",  
 "tom": "entusiasmado",  
 "texto\_do\_artigo": "O LangChain permite construir aplicações de IA complexas de forma modular e eficiente, abrindo um novo mundo de possibilidades para desenvolvedores Python."  
})  
  
print(response.content)

O pipeline criado pelo operador | torna o fluxo de dados entre componentes simples e legível.

**Dicas de Integração:** - Combine PromptTemplates com parsers de saída para garantir respostas no formato desejado. - Use o operador pipe (|) para criar pipelines flexíveis e testáveis. - Experimente diferentes modelos (Gemini, Claude, GPT) e compare resultados. - Adote logs e tracing para depurar e otimizar o fluxo de dados.

### Hands-on: Exercício — Tradutor Multilíngue Dinâmico com LangChain

* **Objetivo:** Criar uma chain que usa um ChatPromptTemplate dinâmico para instruir um LLM a realizar traduções.
* **Nome do Arquivo:** exercicios/capitulo\_03/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv
* **Comando de Instalação:**

uv add langchain langchain-google-genai python-dotenv

**Exemplo de Código: exercicios/capitulo\_03/exercicio\_1/main.py**

import os  
from dotenv import load\_dotenv  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_core.output\_parsers import StrOutputParser  
  
# Carrega variáveis de ambiente (.env)  
load\_dotenv()  
GOOGLE\_API\_KEY = os.getenv("GOOGLE\_API\_KEY")  
if not GOOGLE\_API\_KEY:  
 raise EnvironmentError("GOOGLE\_API\_KEY não encontrada. Adicione ao seu arquivo .env.")  
  
# Inicializa o modelo Gemini  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.7)  
  
# Template dinâmico para tradução  
template = """Você é um tradutor multilíngue especialista.  
Traduza o texto abaixo para {idioma\_destino}.  
Responda apenas com a tradução.  
  
Texto:  
{texto\_origem}  
"""  
  
prompt = ChatPromptTemplate.from\_template(template)  
parser = StrOutputParser()  
  
# Pipeline LCEL: prompt -> modelo -> parser  
chain = prompt | llm | parser  
  
# Exemplos de uso  
exemplos = [  
 {"texto\_origem": "A inteligência artificial está mudando o mundo.", "idioma\_destino": "Francês"},  
 {"texto\_origem": "Python is a powerful programming language.", "idioma\_destino": "Japonês"},  
 {"texto\_origem": "Hello, world!", "idioma\_destino": "Klingon"},  
]  
  
for exemplo in exemplos:  
 print(f"Traduzindo '{exemplo['texto\_origem']}' para {exemplo['idioma\_destino']}...")  
 resultado = chain.invoke(exemplo)  
 print(f"Resultado: {resultado}\n")

**Comando de Execução (Linux/macOS):**

chmod +x exercicios/capitulo\_03/exercicio\_1/run.sh  
./exercicios/capitulo\_03/exercicio\_1/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capulo\_03\exercicio\_1\run.bat

**Saída Esperada (pode variar):** Traduzindo ‘A inteligência artificial está mudando o mundo.’ para Francês… Resultado: L’intelligence artificielle change le monde.

Traduzindo ‘Python is a powerful programming language.’ para Japonês… Resultado: パイソンは強力なプログラミング言語です。

Traduzindo ‘Hello, world!’ para Klingon… Resultado: nuqneH, yaj!

**Troubleshooting Comum:** \* AuthenticationError ou GOOGLE\_API\_KEY não configurada: Certifique-se de que você obteve sua chave do Google AI Studio e a adicionou corretamente ao arquivo .env. \* Respostas Inesperadas: Use temperature=0 para traduções literais. Se persistir, revise o template ou o idioma solicitado. \* Erros de Conexão: Verifique sua internet ou limites de uso da API. \* Prompt não retorna o formato esperado: Adicione instruções claras sobre o formato desejado (ex: “Responda apenas com a tradução”). \* Traduções para idiomas fictícios: Teste limites do modelo e documente resultados curiosos.

### Pontos Chave

* Engenharia de Prompts é crucial para obter respostas de qualidade dos LLMs.
* PromptTemplates no LangChain permitem prompts dinâmicos e reutilizáveis.
* LLMs são integrados e orquestrados eficientemente com o LangChain.
* A LCEL simplifica a construção de pipelines, tornando-os mais legíveis e eficientes.
* Exercícios práticos solidificam o aprendizado dos conceitos de prompts e modelos.
* Modularize templates para facilitar reuso e manutenção.
* Use logs, tracing e testes para depurar e evoluir seus prompts.

### Resumo do Capítulo

Neste capítulo, mergulhamos na arte e ciência da engenharia de prompts e como o LangChain nos ajuda a dominar essa habilidade.

* Engenharia de Prompts: A qualidade da interação com LLM depende da clareza e estrutura das instruções.
* Templates de Prompt: Modularidade e reutilização com PromptTemplates.
* Integração de LLMs: Inicialização e conexão de modelos ao template.
* LCEL: Pipeline elegante com operador pipe (|).
* Exercício Prático: Tradutor multilíngue usando variáveis dinâmicas em prompts.
* Estratégias para depuração, avaliação e evolução de prompts em projetos reais.

### Teste seu Conhecimento

1. Qual é o principal benefício de usar PromptTemplates em vez de strings formatadas (f-strings)?
   1. PromptTemplates são mais rápidos de executar.
   2. Eles permitem criar prompts modulares, reutilizáveis e mais fáceis de manter.
   3. Apenas PromptTemplates podem ser usados com modelos de chat.
   4. Eles usam menos tokens de API.
2. No exemplo do tradutor, qual componente foi responsável por transformar a saída do modelo de um objeto de mensagem para uma string simples?
   1. ChatGoogleGenerativeAI
   2. ChatPromptTemplate
   3. StrOutputParser
   4. load\_dotenv
3. O que o parâmetro temperature em um LLM geralmente controla?
   1. A velocidade da resposta.
   2. O custo da chamada de API.
   3. O quão criativa ou determinística é a resposta (aleatoriedade).
   4. O idioma da resposta.
4. Qual é a principal função da LangChain Expression Language (LCEL) e seu operador |?
   1. Realizar operações matemáticas.
   2. Conectar (encadear) diferentes componentes do LangChain, como prompts e modelos, em um pipeline.
   3. Definir variáveis de ambiente.
   4. Imprimir a saída no console.
5. No exemplo do tradutor, se você quisesse que a tradução fosse o mais literal e menos criativa possível, qual valor você escolheria para temperature?
   1. 1.0
   2. 0.7
   3. 0.5
   4. 0.0

**Respostas:** 1. b 2. c 3. c 4. b 5. d

### Projeto Hands-on: Melhorando o Chatbot Simples

Coloque em prática os conceitos do capítulo criando um chatbot simples com LangChain, PromptTemplates e LCEL. Implemente funcionalidades como tradução automática, ajuste de tom e número de pontos em respostas. Documente comandos de execução, proteja chaves de API com .env e compartilhe o projeto via GitHub usando SSH.

## Capítulo 4: Construção de Pipelines: Da SequentialChain à LCEL

**Neste capítulo, você vai aprender:**

* O papel dos pipelines em aplicações reais (chatbots, RAG, automação).
* A evolução das chains no LangChain: do imperativo ao declarativo.
* Como construir pipelines com LCEL, RunnablePassthrough, RunnableParallel e RunnableLambda.
* Diferenças entre abordagens clássicas (LLMChain/SequentialChain) e modernas (LCEL).
* Exercícios práticos para criar pipelines eficientes, paralelos, com streaming e tracing.
* Troubleshooting, checklist de construção, teste de conhecimento e projeto hands-on.

### A Evolução das Chains: Do Imperativo ao Declarativo

Nos primeiros dias do LangChain, a forma de construir fluxos de trabalho era através de classes como LLMChain e SequentialChain. Essa abordagem, embora funcional, era o que chamamos de **imperativa**: você instanciava objetos e os conectava de forma mais verbosa, o que tornava o código mais difícil de ler e manter. Era como dar instruções passo a passo para montar um móvel, detalhando cada parafuso.

O time do LangChain percebeu que poderia haver uma maneira melhor, mais intuitiva e mais poderosa. Uma maneira que fosse **declarativa**, onde você simplesmente descreve o fluxo de dados — como um diagrama de montagem — e o framework se encarrega da execução otimizada. LCEL permite propagação automática de configurações (RunnableConfig), facilitando logging, tracing e debugging em pipelines complexos.

Essa percepção deu origem à **LangChain Expression Language (LCEL)**, lançada em agosto de 2023. A LCEL é, sem dúvida, uma das inovações mais importantes do framework e é a maneira moderna e recomendada de construir qualquer tipo de pipeline.

Neste capítulo, vamos fazer uma viagem no tempo. Primeiro, vamos construir uma chain “à moda antiga” para entender as dores que a LCEL veio resolver. Depois, vamos mergulhar de cabeça na LCEL e ver como ela torna nossa vida muito mais fácil.

### O Jeito Clássico: LLMChain e SequentialChain (OBSOLETO)

Para entendermos o salto que a LCEL representa, vamos primeiro ver como as coisas eram feitas. É importante notar que as classes LLMChain e SequentialChain são consideradas **legadas** e **deprecadas** na versão 0.2+ do LangChain. Elas são apresentadas aqui apenas para fins de contexto histórico e para ilustrar os problemas que a LCEL veio resolver. **Não as utilize em novos projetos.**

A LLMChain era o bloco de construção mais básico: um prompt, um LLM e uma saída.

# O jeito antigo com LLMChain (OBSOLETO)  
from langchain.chains import LLMChain  
from langchain\_core.prompts import PromptTemplate  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
  
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash")  
prompt = PromptTemplate.from\_template("Qual a capital de {pais}?")  
  
# Criando a LLMChain  
chain = LLMChain(llm=llm, prompt=prompt)  
# print(chain.run("França")) # .run() era o método de invocação

Agora, e se quiséssemos fazer aquilo que discutimos no Capítulo 1: gerar uma pergunta e depois respondê-la? Precisávamos da SequentialChain.

**Observação:** O uso de APIs obsoletas pode dificultar manutenção e escalabilidade. Prefira LCEL para novos projetos.

**Exercício Prático: Chain Sequencial (O Jeito Antigo - OBSOLETO)**

* **Objetivo:** Demonstrar como as chains eram construídas antes da LCEL usando LLMChain e SequentialChain. Este exemplo é **apenas para fins educacionais** e não deve ser replicado em código de produção.
* **Nome do Arquivo:** exercicios/capitulo\_04/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv
* **Comando de Instalação:** uv add langchain langchain-google-genai python-dotenv

# exercicios/capitulo\_04/exercicio\_1/main.py (OBSOLETO)  
from dotenv import load\_dotenv  
from langchain.chains import LLMChain, SequentialChain  
from langchain\_core.prompts import PromptTemplate  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
  
load\_dotenv()  
  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.7)  
  
# Chain 1: Gera uma pergunta  
template\_pergunta = "Gere uma pergunta criativa sobre o tópico: {topico}"  
prompt\_pergunta = PromptTemplate.from\_template(template\_pergunta)  
chain\_pergunta = LLMChain(llm=llm, prompt=prompt\_pergunta, output\_key="pergunta")  
  
# Chain 2: Responde a pergunta gerada  
template\_resposta = "Responda a seguinte pergunta de forma elaborada: {pergunta}"  
prompt\_resposta = PromptTemplate.from\_template(template\_resposta)  
chain\_resposta = LLMChain(llm=llm, prompt=prompt\_resposta, output\_key="resposta\_final")  
  
# Unindo as duas com SequentialChain  
# Note como precisamos definir explicitamente as chaves de entrada e saída  
overall\_chain = SequentialChain(  
 chains=[chain\_pergunta, chain\_resposta],  
 input\_variables=["topico"],  
 output\_variables=["pergunta", "resposta\_final"],  
 verbose=True # Para vermos o que está acontecendo  
)  
  
# Executando a chain  
resultado = overall\_chain({"topico": "a relatividade de Einstein"})  
print("\n--- RESULTADO FINAL ---")  
print(resultado)

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_04/exercicio\_1/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_04/exercicio\_1/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_04\exercicio\_1\run.bat

**Saída Esperada (pode variar):** Resultado final com pergunta gerada e resposta elaborada sobre o tópico informado.

**Troubleshooting Comum:** \* DeprecationWarning: Aviso esperado ao usar classes legadas. \* AuthenticationError ou GOOGLE\_API\_KEY não configurada: Verifique o arquivo .env. \* Erros de Conexão: Verifique rede ou limites de uso da API.

### Por que usar LCEL?

A LCEL resolve muitos dos problemas das classes legadas, como LLMChain e SequentialChain, que são consideradas obsoletas na versão 0.2+ do LangChain. Com LCEL, você constrói pipelines de forma **declarativa**, focando no fluxo de dados e não na implementação detalhada de cada etapa. Configurações como logging e tracing são propagadas automaticamente.

A abordagem moderna da LCEL permite criar pipelines claros, concisos e escaláveis. Você descreve o que deseja fazer e o LangChain executa de forma otimizada, facilitando a leitura, manutenção e evolução do código.

Além da clareza, a LCEL traz ganhos de performance: é otimizada para execução eficiente, suporta paralelismo nativo (com RunnableParallel), streaming de respostas e integração direta com ferramentas como LangSmith. Isso permite lidar com grandes volumes de dados e operações complexas sem comprometer a velocidade ou a robustez.

### Hands-on: Exercício 2 — LCEL na Prática: Sinopse de Filme

A melhor forma de entender a LCEL é colocando a mão na massa. Vamos passar por uma série de exercícios que demonstram seus principais recursos.

**Dica:** Funções Python são automaticamente convertidas em RunnableLambda ao usar o operador |. Dicionários são convertidos em RunnableParallel para execução concorrente.

**Exemplo de streaming:**

for chunk in chain.stream({"genero": "Comédia"}):  
 print(chunk, end="|")

**Exercício 1: A Chain Mais Simples (com LCEL)**

* **Objetivo:** Criar uma chain básica que gera uma sinopse de filme a partir de um gênero.

# exercicios/capitulo\_04/exercicio\_2/main.py   
from dotenv import load\_dotenv   
from langchain\_google\_genai import ChatGoogleGenerativeAI   
from langchain\_core.prompts import ChatPromptTemplate   
from langchain\_core.output\_parsers import StrOutputParser  
  
load\_dotenv()  
  
prompt = ChatPromptTemplate.from\_template("Crie uma sinopse de filme de uma frase para o gênero: {genero}")   
model = ChatGoogleGenerativeAI(model="gemini-2.5-flash")   
parser = StrOutputParser()  
  
chain = prompt | model | parser  
  
print(chain.invoke({"genero": "Comédia de Ficção Científica"}))

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_04/exercicio\_2/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_04/exercicio\_2/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_04\exercicio\_2\run.bat

**Saída Esperada (pode variar):** Sinopse de filme gerada para o gênero informado.

**Troubleshooting Comum:** \* AuthenticationError ou GOOGLE\_API\_KEY não configurada. \* ValidationError ou erros de schema. \* Erros de Conexão. \* Saída inesperada do LLM: ajuste o prompt ou a temperature.

### Hands-on: Exercício 3 — Passando Dados com RunnablePassthrough

**Dica:** Use RunnablePassthrough.assign para empacotar outputs em dicionários, útil para debugging e integração.

**Exercício 2: Passando Dados com RunnablePassthrough**

* **Objetivo:** Criar uma chain que gera uma pergunta sobre um tópico e depois responde a essa pergunta, usando o tópico original na resposta final.

# exercicios/capitulo\_04/exercicio\_3/main.py   
from dotenv import load\_dotenv   
from langchain\_core.prompts import ChatPromptTemplate   
from langchain\_google\_genai import ChatGoogleGenerativeAI   
from langchain\_core.output\_parsers import StrOutputParser   
from langchain\_core.runnables import RunnablePassthrough  
  
load\_dotenv()   
model = ChatGoogleGenerativeAI(model="gemini-2.5-flash")   
parser = StrOutputParser()  
  
prompt\_pergunta = ChatPromptTemplate.from\_template("Gere uma pergunta interessante sobre {topico}.")   
prompt\_resposta = ChatPromptTemplate.from\_template(   
 "Responda a pergunta: {pergunta}. Contexto original do tópico: {topico}"   
)  
  
chain\_pergunta = prompt\_pergunta | model | parser  
  
chain\_completa = (   
 {"pergunta": chain\_pergunta, "topico": RunnablePassthrough()}   
 | prompt\_resposta   
 | model   
 | parser   
)  
  
print(chain\_completa.invoke("a filosofia estoica"))

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_04/exercicio\_3/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_04/exercicio\_3/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_04\exercicio\_3\run.bat

**Saída Esperada (pode variar):** Pergunta gerada sobre o tópico e resposta contextualizada.

**Troubleshooting Comum:** \* AuthenticationError ou GOOGLE\_API\_KEY não configurada. \* ValidationError ou erros de schema. \* Erros de Conexão. \* Saída inesperada do LLM: ajuste o prompt ou a temperature.

### Hands-on: Exercício 4 — Execução Paralela com RunnableParallel

**Dica:** O uso de RunnableParallel permite que cada etapa receba o mesmo input e execute em paralelo, retornando um dicionário com os resultados.

**Exercício 3: Execução Paralela com RunnableParallel**

* **Objetivo:** Para um determinado país, buscar em paralelo sua capital, sua população e uma curiosidade.

# exercicios/capitulo\_04/exercicio\_4/main.py   
from dotenv import load\_dotenv   
from langchain\_core.prompts import ChatPromptTemplate   
from langchain\_google\_genai import ChatGoogleGenerativeAI   
from langchain\_core.output\_parsers import StrOutputParser   
from langchain\_core.runnables import RunnableParallel  
  
load\_dotenv()   
model = ChatGoogleGenerativeAI(model="gemini-2.5-flash")   
parser = StrOutputParser()  
  
chain\_capital = ChatPromptTemplate.from\_template("Qual é a capital de {pais}?") | model | parser   
chain\_populacao = ChatPromptTemplate.from\_template("Qual é a população aproximada de {pais}?") | model | parser   
chain\_curiosidade = ChatPromptTemplate.from\_template("Conte uma curiosidade sobre {pais}.") | model | parser  
  
mapa\_paralelo = RunnableParallel(   
 capital=chain\_capital,   
 populacao=chain\_populacao,   
 curiosidade=chain\_curiosidade   
)  
  
resultado = mapa\_paralelo.invoke({"pais": "Egito"})   
print(resultado)

**Comando de Execução (Linux/macOS):**

# Dê permissão de execução ao script  
chmod +x exercicios/capitulo\_04/exercicio\_4/run.sh  
  
# Execute o exercício  
./exercicios/capitulo\_04/exercicio\_4/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_04\exercicio\_4\run.bat

**Saída Esperada (pode variar):** Retorno paralelo: capital, população e curiosidade sobre o país informado.

**Troubleshooting Comum:** \* AuthenticationError ou GOOGLE\_API\_KEY não configurada. \* ValidationError ou erros de schema. \* Erros de Conexão. \* Saída inesperada do LLM: ajuste o prompt ou a temperature.

### Troubleshooting Comum

* DeprecationWarning: Aviso esperado ao usar classes legadas.
* AuthenticationError ou GOOGLE\_API\_KEY não configurada: Verifique o arquivo .env.
* Erros de Conexão: Verifique rede ou limites de uso da API.
* Ative tracing com LangSmith usando LANGSMITH\_TRACING=true.
* Proteja chaves de API com Pydantic SecretStr ou arquivos .env.
* Implemente testes automatizados para chains customizadas.

### Pontos Chave

* LCEL é a abordagem moderna e recomendada para pipelines no LangChain.
* Streaming, paralelismo e integração com LangSmith são nativos na LCEL.
* Chains legadas são úteis para contexto histórico, mas não devem ser usadas em produção.
* Exercícios práticos demonstram as vantagens da LCEL em clareza, modularidade e performance.
* O uso de RunnableParallel e RunnablePassthrough facilita a modularização e reuso de componentes.

### Resumo do Capítulo

Neste capítulo, você aprendeu a diferença entre pipelines imperativos e declarativos, explorou a LCEL e suas vantagens, e colocou em prática a construção de chains modernas, paralelas e eficientes. Explore integrações avançadas como LangGraph e LangServe para projetos de maior escala.

### Teste seu Conhecimento

1. Qual é a principal vantagem da LCEL em relação às chains clássicas?
   1. Permite integração com LangSmith e streaming nativo.
   2. É mais difícil de usar.
   3. Não suporta paralelismo.
   4. Só funciona com LLMChain.
2. O que o RunnablePassthrough permite em uma chain LCEL?
   1. Executar múltiplos modelos em paralelo.
   2. Passar dados do input diretamente para etapas subsequentes.
   3. Validar a saída do modelo.
   4. Gerar prompts automaticamente.
3. Qual componente LCEL permite executar múltiplas chains ao mesmo tempo?
   1. RunnableLambda
   2. RunnableParallel
   3. RunnablePassthrough
   4. SequentialChain
4. Qual é o método recomendado para obter respostas token a token em LCEL?
   1. .run()
   2. .stream()
   3. .ainvoke()
   4. .output()
5. Por que as chains legadas como LLMChain e SequentialChain são consideradas obsoletas?
   1. Não funcionam com Python 3.12.
   2. Não suportam streaming, paralelismo e integração moderna.
   3. São mais rápidas que LCEL.
   4. Foram removidas do LangChain.
6. Como ativar tracing para inspeção de pipelines?
   1. Definindo LANGSMITH\_TRACING=true no ambiente.
   2. Usando apenas SequentialChain.
   3. Removendo o arquivo .env.
   4. Desabilitando logs.

**Respostas:** 1. a 2. b 3. b 4. b 5. b 6. a

### Projeto Hands-on: Pipeline Inteligente

Coloque em prática os conceitos do capítulo criando um pipeline inteligente com LCEL. Implemente etapas paralelas, streaming de respostas e integração com LangSmith. Siga a estrutura sugerida:

* Estrutura de diretórios recomendada para projetos LCEL.
* Checklist de boas práticas: modularização, logging, versionamento, testes.
* Fluxograma do pipeline.
* Dicas para integração com APIs externas e testes automatizados.
* Orientação para uso de streaming e paralelismo.
* Passos para ativar tracing com LangSmith e compartilhar logs via GitHub.
* Proteja chaves de API com .env.
* Documente comandos de execução e compartilhe o projeto via GitHub usando SSH.

## Capítulo 5: Desenvolvimento de Agentes Autônomos e Multiagentes

**Neste capítulo, você vai aprender:**

* O papel dos agentes autônomos e multiagentes em aplicações de IA modernas.
* Como construir agentes que tomam decisões dinâmicas usando ferramentas (Tools) e o padrão ReAct.
* Engenharia de contexto, uso de memória e o papel do RAG para ampliar o conhecimento dos agentes.
* Exercício prático de agente de pesquisa com integração de ferramentas externas e adaptação para LangGraph.
* Introdução ao LangGraph para orquestração de sistemas multiagentes complexos.
* Pontos chave, troubleshooting, checklist e teste de conhecimento para consolidar o aprendizado.

Seja bem-vindo a um dos territórios mais fascinantes e, admito, mais complexos do LangChain: os agentes. Se o conceito parecer um pouco abstrato ou intimidador no começo, não se preocupe. É um salto conceitual significativo. Pense nisso como aprender a andar de bicicleta depois de só ter usado patinetes. Nos capítulos anteriores, construímos “patinetes”: fluxos de trabalho lineares e previsíveis. Agora, vamos dar equilíbrio e autonomia à nossa criação.

O equilíbrio, assim como na bicicleta, vem com a prática. Vamos começar com calma, entender cada peça, e logo você estará pedalando por conta própria, criando sistemas que pensam e agem. Confie no processo, e vamos passo a passo.

### O que é um Agente? O LLM como Cérebro

Até agora, usamos LLMs para executar tarefas bem definidas dentro de um pipeline (uma Chain). O fluxo era fixo. Um agente, por outro lado, vira esse jogo de cabeça para baixo. Em um sistema agêntico, o LLM não é apenas um executor de tarefas, ele é o **cérebro**, o **motor de raciocínio** que decide o que fazer a seguir.

A principal diferença entre uma Chain e um Agente é a **autonomia**. Um agente é um sistema que usa um LLM como seu “motor de raciocínio” para determinar a sequência de ações a serem tomadas. O padrão ReAct (Reason + Act) é implementado nativamente via create\_react\_agent no LangChain e LangGraph, permitindo raciocínio iterativo e uso dinâmico de ferramentas.

* Uma **Chain** segue um caminho predeterminado. Ex: Prompt -> LLM -> Parse. O caminho não muda.
* Um **Agente** usa o LLM para escolher um caminho dinamicamente a partir de um conjunto de opções disponíveis (as ferramentas). O caminho pode ser diferente a cada execução.

Essa capacidade de tomar decisões em tempo de execução é o que permite que agentes resolvam problemas complexos, interajam com o mundo exterior e executem tarefas que não podem ser roteirizadas de antemão.

Pense na cena icônica de Matrix (1999), onde Morpheus oferece a Neo a pílula azul ou a pílula vermelha. Neo (o usuário) está buscando a verdade. Morpheus (o Agente) é o sistema que, com base em seu “raciocínio” e acesso a “ferramentas” (como o conhecimento da Matrix e a capacidade de manipular programas), decide qual “ação” tomar para guiar Neo. As “ferramentas” são os programas que Morpheus pode usar, e a “Matrix” é o ambiente onde essas ações acontecem. A pílula vermelha, que revela a verdade e dá a Neo a autonomia para ver o mundo como ele realmente é, pode ser comparada à autonomia que damos aos nossos agentes. Eles não apenas seguem instruções, mas “enxergam” o ambiente e tomam decisões para alcançar um objetivo.

Os efeitos inovadores dessa autonomia são visíveis em modelos como o Google vO3, que demonstram capacidades de raciocínio e interação com o ambiente que eram impensáveis há poucos anos.

### Engenharia de Contexto: A Evolução do Prompt

Quando falamos de agentes, especialmente sistemas com múltiplos agentes que colaboram e compartilham informações, a simples “Engenharia de Prompts” evolui para algo mais sofisticado: a **Engenharia de Contexto**. Componentes de memória e checkpointers (ex: MemorySaver) permitem que agentes mantenham histórico e colaboratividade, facilitando fluxos multiagentes.

Não se trata mais apenas de criar a instrução perfeita para uma única tarefa. Trata-se de gerenciar e moldar dinamicamente o contexto completo que cada agente recebe. Esse contexto pode incluir:

* O objetivo geral da missão.
* O histórico da conversa até o momento (memória de curto prazo).
* Os resultados e observações das ferramentas que já foram executadas.
* Informações relevantes recuperadas de uma base de conhecimento (memória de longo prazo, ou RAG).
* Uma “memória compartilhada” ou um “quadro branco” onde outros agentes deixaram notas.

A engenharia de contexto é a arte de garantir que o agente certo receba a informação certa no momento certo, sem sobrecarregá-lo com dados irrelevantes. É um desafio de design crucial para a eficiência de sistemas multiagentes.

### RAG (Retrieval-Augmented Generation): Ampliando o Conhecimento do LLM

Até agora, nossos LLMs operam com o conhecimento que foi “treinado” neles. Mas e se precisarmos que eles respondam a perguntas sobre dados muito específicos, privados ou que mudam constantemente? É aqui que entra o **RAG (Retrieval-Augmented Generation)**, um padrão arquitetural que permite aos LLMs acessar e utilizar informações externas e atualizadas.

Pense no RAG como dar ao seu LLM a capacidade de “consultar livros” antes de responder. O processo segue um fluxo lógico de quatro etapas:

1. **Consulta (Query):** O usuário faz uma pergunta ou fornece uma entrada ao sistema.
2. **Recuperação (Retrieval):** Em vez de o LLM tentar responder apenas com seu conhecimento interno, o sistema primeiro “recupera” informações relevantes de uma base de dados externa (como documentos, artigos, bancos de dados, etc.). Essa recuperação é feita buscando por similaridade semântica entre a consulta do usuário e os documentos na base de conhecimento.
3. **Aumento (Augmentation):** As informações recuperadas são então “aumentadas” (adicionadas) ao prompt original do usuário. Isso cria um prompt mais rico e contextualizado, que é então enviado ao LLM.
4. **Geração (Generation):** O LLM recebe o prompt aumentado e gera uma resposta que não apenas utiliza seu conhecimento interno, mas também incorpora e se baseia nas informações recuperadas externamente.

O RAG é crucial para construir aplicações de IA que precisam ser factualmente precisas, atualizadas e capazes de operar sobre grandes volumes de dados específicos de um domínio. Ele minimiza as “alucinações” (respostas inventadas) dos LLMs e os torna muito mais úteis em cenários corporativos e de dados sensíveis.

### Componentes de um Agente: Ferramentas e o Executor

Para construir um agente, precisamos de dois componentes principais:

1. **Ferramentas (Tools):** São as ações que o agente pode executar. Uma ferramenta é, essencialmente, uma função Python com uma descrição muito bem escrita. A descrição é crucial, pois é isso que o LLM lê para decidir se e quando deve usar aquela ferramenta. Exemplos de ferramentas:
   * Uma busca na web.
   * Uma calculadora.
   * Uma função que lê um arquivo.
   * Uma função que consulta uma API.
   * Uma função que interage com um banco de dados.
2. **Executor do Agente (Agent Executor):** É o runtime que orquestra o loop de raciocínio do agente. O padrão mais comum, conhecido como **ReAct (Thought + Action + Observation)**, funciona da seguinte maneira:
   * **Thought (Pensamento):** O LLM recebe o objetivo e a lista de ferramentas disponíveis. Ele então “pensa em voz alta” (*Chain of Thought*) para decidir qual ferramenta usar e com quais argumentos.
   * **Action (Ação):** O executor invoca a ferramenta escolhida com os argumentos definidos pelo LLM.
   * **Observation (Observação):** O resultado da ferramenta é retornado ao LLM como uma “observação”.
   * **Repeat (Repetição):** O LLM analisa a observação e decide se a tarefa está concluída ou se precisa de mais um ciclo de Pensamento, Ação e Observação.

Ferramentas são funções Python decoradas ou instâncias de classes Tool, e descrições detalhadas são essenciais para o LLM decidir seu uso. O executor do agente pode ser criado com create\_react\_agent do LangGraph, integrando LLM, ferramentas, prompt e memória.

### Hands-on: Exercício — Agente de Pesquisa Simples

#### O que é a API da Tavily? Como obter e custos

* **Objetivo:** Construir um agente simples que pode usar uma ferramenta de busca para responder a perguntas factuais.
* **Nome do Arquivo:** exercicios/capitulo\_05/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, langchain-tavily, python-dotenv
* **Comando de Instalação:**

uv add langchain langchain-google-genai langchain-tavily python-dotenv

* **Configuração Adicional:** Adicione sua chave de API da Tavily ao arquivo .env como TAVILY\_API\_KEY.

A **API da Tavily** é uma ferramenta de busca que permite ao seu agente LangChain pesquisar informações atualizadas na internet, especialmente sobre eventos recentes, pessoas, empresas e fatos que mudam rapidamente. Ela é usada como uma “ferramenta externa” para ampliar o conhecimento do agente além do que está disponível no modelo de linguagem.

* **Para que serve:** Permite que o agente realize buscas em tempo real, trazendo respostas baseadas em fontes confiáveis e atualizadas, reduzindo o risco de respostas desatualizadas ou inventadas.
* **Como obter:**
  1. Acesse <https://app.tavily.com/> e crie uma conta gratuita.
  2. No painel da Tavily, vá em “API Keys” e gere uma nova chave.
  3. Copie a chave e adicione ao seu arquivo .env como TAVILY\_API\_KEY=suachaveaqui.
* **Tem custo?**
  + A Tavily oferece um plano gratuito com limite de buscas mensais (veja detalhes atualizados no site).
  + Para uso mais intenso ou comercial, existem planos pagos com limites maiores e recursos adicionais.

**Dica:** Sempre proteja sua chave de API e nunca a compartilhe publicamente.

# exercicios/capitulo\_05/exercicio\_1/main.py  
  
import os   
from dotenv import load\_dotenv   
from langchain\_google\_genai import ChatGoogleGenerativeAI   
from langchain\_tavily import TavilySearch  
from langchain.agents import AgentExecutor, create\_tool\_calling\_agent   
from langchain\_core.prompts import ChatPromptTemplate  
  
# Carregar variáveis de ambiente   
load\_dotenv()  
  
# 1. Escolher o LLM que será o cérebro do nosso agente   
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
  
# 2\. Definir as ferramentas que o agente pode usar   
search\_tool = TavilySearch(  
 max\_results=2,  
 description="Uma ferramenta de busca para encontrar informações na internet sobre eventos atuais, pessoas, lugares ou empresas."  
)   
tools = [search\_tool]  
  
# 3. Criar o Prompt do Agente   
# Este prompt é um template especial que guia o agente, permitindo que ele "pense" e registre suas ações.   
# Os placeholders {chat\_history} e {agent\_scratchpad} são cruciais e gerenciados automaticamente pelo AgentExecutor.   
# O {agent\_scratchpad} é onde o agente registra seu processo de raciocínio (Thought), as ferramentas que decide usar (Action)   
# e os resultados dessas ações (Observation), formando o ciclo ReAct que vimos anteriormente.   
prompt = ChatPromptTemplate.from\_messages([   
 ("system", "Você é um assistente prestativo."),   
 ("placeholder", "{chat\_history}"),   
 ("human", "{input}"),   
 ("placeholder", "{agent\_scratchpad}"),   
])  
  
# 4\. Criar o Agente   
# Esta função conecta o LLM, as ferramentas e o prompt   
agent = create\_tool\_calling\_agent(llm, tools, prompt)  
  
# 5\. Criar o Executor do Agente   
# O executor é o loop que roda o agente até a resposta final   
agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)  
  
# \--- Execução \---  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("Agente de Pesquisa pronto! Faça sua pergunta.")  
  
 pergunta1 = "Qual foi o filme vencedor do Oscar de Melhor Filme em 2024?"  
 print(f"\n> Pergunta: {pergunta1}")  
 response1 = agent\_executor.invoke({"input": pergunta1})  
 print(f"\n< Resposta Final: {response1['output']}")  
  
 pergunta2 = "Qual é a cor do céu?"  
 print(f"\n> Pergunta: {pergunta2}")  
 response2 = agent\_executor.invoke({"input": pergunta2})  
 print(f"\n< Resposta Final: {response2['output']}")

**Comando de Execução (Linux/macOS):**

chmod +x exercicios/capitulo\_05/exercicio\_1/run.sh  
./exercicios/capitulo\_05/exercicio\_1/run.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
exercicios\capitulo\_05\exercicio\_1\run.bat

**Saída Esperada (pode variar):**

Agente de Pesquisa pronto! Faça sua pergunta.  
> Pergunta: Qual foi o filme vencedor do Oscar de Melhor Filme em 2024?  
< Resposta Final: [Resposta baseada na busca Tavily]  
> Pergunta: Qual é a cor do céu?  
< Resposta Final: Azul

* **Dica:** O exercício pode ser facilmente adaptado para LangGraph usando create\_react\_agent, tornando o agente mais robusto e escalável. Exemplo:

from langgraph.prebuilt import create\_react\_agent  
agent\_executor = create\_react\_agent(llm, tools)  
response = agent\_executor.invoke({"messages": [("human", pergunta1)]})  
print(response["messages"][-1].content)

* **Dica:** Use streaming para inspecionar o raciocínio do agente passo a passo:

for event in agent\_executor.stream({"messages": [("human", pergunta1)]}, stream\_mode="values"):  
 print(event["messages"][-1].content)

* Proteja chaves de API com arquivos .env e configure tracing conforme necessário.

### Sistemas Multiagentes e a Magia do LangGraph

E se um problema for tão complexo que um único agente não é suficiente? Entramos no mundo dos **sistemas multiagentes**. A ideia é criar um “time” de agentes especializados, cada um com suas próprias ferramentas e responsabilidades, que colaboram para resolver um problema maior.

Por exemplo, imagine um agente de pesquisa, um agente escritor e um agente crítico trabalhando juntos para criar um relatório. O orquestrador (ou “roteador”) passaria a tarefa inicial para o pesquisador, o resultado para o escritor, a versão escrita para o crítico, e o feedback de volta para o escritor, em um ciclo.

Gerenciar esses fluxos complexos, que podem ter ciclos e condicionais, é um desafio. É para isso que o **LangGraph** foi criado. Ele é uma biblioteca construída sobre o LangChain que permite definir fluxos de trabalho de agentes como um **grafo**. Cada nó no grafo pode ser um agente ou uma função, e as arestas definem como o estado (as informações) flui entre eles.

O LangGraph é o estado da arte para construir sistemas multiagentes robustos e é um tópico avançado que exploraremos em projetos futuros, mas é fundamental que você saiba que ele existe e qual problema ele resolve.

### Troubleshooting Comum

* Instale o LangGraph com pip install langgraph.
* Configure memória e checkpointers para agentes multiagentes.
* Debug de fluxos multiagentes pode ser feito inspecionando o histórico de mensagens e eventos.
* Proteja chaves de API e configure tracing para inspeção detalhada.

### Pontos Chave

* Agentes usam LLMs como “cérebros” para tomar decisões dinâmicas, diferentemente das Chains com fluxo fixo.
* Engenharia de Contexto e uso de memória são cruciais para gerenciar informações em sistemas agênticos.
* Ferramentas (Tools) e o Executor do Agente (Agent Executor) são componentes essenciais para a funcionalidade do agente.
* O LangGraph é o padrão para orquestrar sistemas multiagentes complexos.
* O uso de ferramentas bem descritas e engenharia de contexto são diferenciais para agentes eficientes.

### Resumo do Capítulo

Neste capítulo, você aprendeu sobre agentes autônomos e multiagentes no LangChain, entendendo como eles diferem das chains tradicionais ao tomar decisões dinâmicas. Explorou o padrão ReAct, engenharia de contexto, o papel do RAG para ampliar o conhecimento dos agentes, e construiu um agente de pesquisa hands-on. Conheceu o LangGraph para orquestração de sistemas multiagentes e revisou os principais conceitos para consolidar o aprendizado.

**Nota do autor:** O tema de sistemas multiagentes e orquestração com LangGraph é vasto e está em rápida evolução. Se você se interessa por arquiteturas avançadas de IA, vale considerar um livro dedicado ao LangGraph. Ele permitiria explorar casos de uso, padrões de design, integração com outras ferramentas e exemplos práticos em profundidade. O interesse por sistemas multiagentes está crescendo, e um material aprofundado pode ser muito útil para a comunidade.

### Teste seu Conhecimento

1. Qual é a característica que define um Agente e o diferencia de uma Chain?
   1. O uso de modelos de linguagem do Google.
   2. A capacidade de tomar decisões dinâmicas sobre qual ação executar a seguir.
   3. A velocidade de processamento.
   4. A capacidade de gerar texto.
2. No padrão ReAct (Reason + Act), qual é o papel do LLM?
   1. Apenas executar a ferramenta (Act).
   2. Apenas raciocinar sobre qual ferramenta usar (Reason).
   3. Raciocinar sobre qual ferramenta usar e, em seguida, formular a resposta final com base na observação.
   4. Armazenar os resultados em um banco de dados.
3. O que é uma “Tool” (Ferramenta) no contexto de um agente LangChain?
   1. Um modelo de linguagem específico para uma tarefa.
   2. Uma função Python com uma boa descrição que o agente pode decidir invocar.
   3. A interface de usuário do agente.
   4. Um tipo especial de prompt.
4. No exercício do agente de pesquisa, por que o agente não usou a ferramenta de busca para responder “Qual é a cor do céu?”
   1. Porque a API da Tavily estava offline.
   2. Porque o LLM “sabia” a resposta e julgou que não precisava de informações externas.
   3. Porque a pergunta estava mal formulada.
   4. Porque a ferramenta de busca não funciona para perguntas sobre cores.
5. Para qual tipo de problema o LangGraph é a ferramenta mais indicada?
   1. Para criar prompts simples.
   2. Para construir um pipeline linear com duas etapas.
   3. Para orquestrar sistemas complexos com múltiplos agentes que podem interagir em ciclos.
   4. Para treinar um novo modelo de linguagem.
6. Como agentes podem manter histórico e contexto?
   1. Usando checkpointers e componentes de memória como MemorySaver.
   2. Apenas com prompts fixos.
   3. Ignorando o histórico de mensagens.
   4. Usando apenas ferramentas externas.

**Respostas:** 1. b 2. c 3. b 4. b 5. c 6. a

### Projeto Hands-on: Orquestrando Agentes Inteligentes

Coloque em prática os conceitos do capítulo criando um projeto Python que utiliza LangChain para construir um sistema multiagente. Implemente pelo menos dois agentes especializados (ex: pesquisador e escritor), cada um com suas próprias ferramentas. Use LangGraph para orquestrar o fluxo entre eles, proteja suas chaves de API com .env e documente todos os comandos usados. Siga a estrutura sugerida:

* Estrutura de diretórios para sistemas multiagentes.
* Checklist de boas práticas: modularização, logging, versionamento, testes.
* Fluxograma do grafo de agentes.
* Dicas para integração de múltiplas ferramentas e fontes de dados.
* Passos para ativar tracing e compartilhar logs via GitHub.
* Experimente rodar o projeto em diferentes versões do Python usando pyenv.

## Capítulo 6: Ferramentas e Agentes: Capacitando LLMs com Ações

**Neste capítulo, você vai aprender:**

* Como equipar LLMs com ferramentas externas para criar agentes autônomos e interativos.
* Diferenças entre ferramentas (Tools) e agentes (Agents) no LangChain.
* Implementação prática de agentes com ferramentas customizadas, APIs externas e integração via LangGraph.
* Exercícios práticos de análise de vendas e cotação de moedas, com instruções de execução, troubleshooting e dicas de modularização.
* Pontos chave, checklist e projeto hands-on para consolidar o aprendizado sobre agentes e ferramentas.

Neste capítulo, exploraremos um dos conceitos mais poderosos do LangChain: a capacidade de equipar Large Language Models (LLMs) com ferramentas externas, transformando-os em agentes autônomos capazes de interagir com o mundo real. Veremos como os agentes podem raciocinar sobre qual ferramenta usar, executar ações e observar os resultados para alcançar seus objetivos. O padrão ReAct é recomendado para raciocínio iterativo e uso eficiente de ferramentas.

### **6.1. O que são Ferramentas (Tools)?**

No contexto do LangChain, uma **Ferramenta (Tool)** é uma função que um agente pode invocar para interagir com o mundo exterior. Isso pode incluir:

* **APIs externas:** Buscar informações na web, enviar e-mails, acessar bancos de dados, etc.
* **Funções customizadas:** Realizar cálculos complexos, manipular dados, interagir com sistemas internos.
* **Outros modelos de IA:** Chamar modelos de visão computacional, processamento de fala, etc.

Ferramentas podem ser funções decoradas com @tool, instâncias de classes Tool, ou carregadas via load\_tools para integrações externas. Descrições detalhadas são essenciais para o LLM decidir o uso correto da ferramenta. Recomenda-se testar ferramentas isoladamente antes de integrá-las ao agente.

### **6.2. Agentes (Agents): O Cérebro por Trás das Ações**

Um **Agente (Agent)** é um sistema que utiliza um LLM como seu “cérebro” para decidir qual ação tomar, observar o resultado dessa ação e repetir o processo até que a tarefa seja concluída. O processo geralmente segue o padrão **ReAct (Reason + Act)**:

1. **Reason (Raciocinar):** O LLM analisa a entrada do usuário e o estado atual para determinar qual ferramenta (se houver) deve ser usada e com quais argumentos.
2. **Act (Agir):** A ferramenta selecionada é executada, e o resultado é observado.
3. **Observe (Observar):** O LLM recebe o resultado da execução da ferramenta e o utiliza para refinar seu raciocínio ou gerar a resposta final.

O LangChain oferece diversas implementações de agentes, mas a abordagem moderna e recomendada é o create\_tool\_calling\_agent (LangChain) ou create\_react\_agent (LangGraph), que se beneficiam das capacidades de *tool calling* nativas dos LLMs mais recentes. Prompts podem ser customizados para orientar o agente. Recomenda-se uso de streaming (agent\_executor.stream) para inspecionar o raciocínio do agente passo a passo.

### Hands-on: Exercício 1 — Agente de Análise de Vendas com Ferramenta SQL

* **Objetivo:** Criar um agente que utilize uma ferramenta SQL para consultar dados de vendas e responder a perguntas complexas.
* **Nome do Arquivo:** exercicios/capitulo\_06/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv
* **Comando de Instalação:**

uv add langchain langchain-google-genai python-dotenv

* **Configuração Adicional:** Certifique-se de que sua chave de API do Google Gemini (GOOGLE\_API\_KEY) está configurada corretamente no seu arquivo .env.

# exercicios/capitulo\_06/exercicio\_1/main.py  
import sqlite3  
from dotenv import load\_dotenv  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain.agents import AgentExecutor, create\_tool\_calling\_agent  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain.tools import tool  
  
# Carregar variáveis de ambiente  
load\_dotenv()  
  
# 1. Configurar o banco de dados SQLite em memória  
def setup\_database():  
 conn = sqlite3.connect(':memory:')  
 cursor = conn.cursor()  
 cursor.execute('''  
 CREATE TABLE IF NOT EXISTS vendas (  
 id INTEGER PRIMARY KEY,  
 produto TEXT,  
 quantidade INTEGER,  
 preco REAL,  
 data TEXT  
 )  
 ''')  
 vendas\_data = [  
 ('Laptop', 2, 1200.00, '2024-01-15'),  
 ('Mouse', 5, 25.00, '2024-01-15'),  
 ('Teclado', 3, 75.00, '2024-01-16'),  
 ('Monitor', 1, 300.00, '2024-01-17'),  
 ('Laptop', 1, 1200.00, '2024-01-18'),  
 ('Webcam', 4, 50.00, '2024-01-18'),  
 ]  
 cursor.executemany('INSERT INTO vendas (produto, quantidade, preco, data) VALUES (?, ?, ?, ?)', vendas\_data)  
 conn.commit()  
 return conn  
  
# 2. Definir a ferramenta SQL para o agente  
@tool  
def execute\_sql\_query(query: str) -> str:  
 """  
 Executa uma consulta SQL no banco de dados de vendas e retorna o resultado.  
 Use esta ferramenta para responder a perguntas sobre dados de vendas.  
 O banco de dados contém uma tabela 'vendas' com as colunas:  
 'id', 'produto', 'quantidade', 'preco', 'data'.  
 Exemplo de uso: SELECT SUM(quantidade) FROM vendas WHERE produto = 'Laptop';  
 """  
 conn = setup\_database()  
 cursor = conn.cursor()  
 try:  
 cursor.execute(query)  
 results = cursor.fetchall()  
 conn.close()  
 return str(results)  
 except sqlite3.Error as e:  
 conn.close()  
 return f"Erro ao executar a consulta SQL: {e}"  
  
# 3. Escolher o LLM que será o cérebro do nosso agente  
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0)  
  
# 4. Definir as ferramentas que o agente pode usar  
tools = [execute\_sql\_query]  
  
# 5. Criar o Prompt do Agente  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "Você é um assistente especializado em analisar dados de vendas. Use a ferramenta 'execute\_sql\_query' para responder a perguntas sobre vendas. Se a pergunta não puder ser respondida com os dados de vendas, diga que não tem informações."),  
 ("human", "{input}"),  
 ("placeholder", "{agent\_scratchpad}"),  
])  
  
# 6. Criar o Agente  
agent = create\_tool\_calling\_agent(llm, tools, prompt)  
  
# 7. Criar o Executor do Agente  
agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)  
  
# --- Execução ---  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("Agente de Análise de Vendas pronto! Faça sua pergunta.")  
  
 perguntas = [  
 "Qual a quantidade total de Laptops vendidos?",  
 "Qual o produto mais vendido em termos de quantidade?",  
 "Qual o valor total de vendas?",  
 "Quantos itens foram vendidos em 2024-01-18?",  
 "Qual o nome do meu cachorro?"  
 ]  
  
 for pergunta in perguntas:  
 print(f"\n> Pergunta: {pergunta}")  
 response = agent\_executor.invoke({"input": pergunta})  
 print(f"\n< Resposta Final: {response['output']}")

**Comando de Execução (Linux/macOS):**

cd exercicios/capitulo\_06/exercicio\_1  
chmod +x run.sh  
./run.sh

**Comando de Execução (Windows):**

cd exercicios\capitulo\_06\exercicio\_1  
call run.bat

**Saída Esperada (pode variar):**

Agente de Análise de Vendas pronto! Faça sua pergunta.  
> Pergunta: Qual a quantidade total de Laptops vendidos?  
< Resposta Final: ...  
> Pergunta: Qual o produto mais vendido em termos de quantidade?  
< Resposta Final: ...  
> Pergunta: Qual o valor total de vendas?  
< Resposta Final: ...  
> Pergunta: Quantos itens foram vendidos em 2024-01-18?  
< Resposta Final: ...  
> Pergunta: Qual o nome do meu cachorro?  
< Resposta Final: Não tenho informações sobre isso.

**Troubleshooting Comum:** \* sqlite3.OperationalError: no such table: vendas: Verifique se a função setup\_database() está sendo chamada corretamente. \* AuthenticationError: Verifique a chave de API do Gemini no .env. \* ModuleNotFoundError: Instale todas as dependências com uv add.

### Hands-on: Exercício 2 — Agente de Cotação de Moedas com Ferramenta de API Externa

* **Objetivo:** Criar um agente que utilize uma ferramenta para consultar taxas de câmbio de moedas via API externa.
* **Nome do Arquivo:** exercicios/capitulo\_06/exercicio\_2/main.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv, requests
* **Comando de Instalação:**

uv add langchain langchain-google-genai python-dotenv requests

* **Configuração Adicional:** Se usar uma API que exija chave, adicione ao .env (ex: EXCHANGE\_RATE\_API\_KEY).

# exercicios/capitulo\_06/exercicio\_2/main.py  
from dotenv import load\_dotenv  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain.agents import AgentExecutor, create\_tool\_calling\_agent  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain.tools import tool  
import requests  
import os  
  
# Carregar variáveis de ambiente  
load\_dotenv()  
  
# 1. Definir a ferramenta de cotação de moedas  
@tool  
def get\_exchange\_rate(base\_currency: str, target\_currency: str) -> str:  
 """  
 Obtém a taxa de câmbio atual entre duas moedas.  
 Use esta ferramenta para converter valores entre moedas.  
 Exemplo: get\_exchange\_rate("USD", "BRL")  
 """  
 try:  
 # Usando uma API de cotação de moedas gratuita (ex: ExchangeRate-API)  
 # Você pode precisar se registrar para obter uma chave de API real  
 # Para fins de demonstração, usaremos um endpoint público ou simulado  
 api\_key = os.getenv("EXCHANGE\_RATE\_API\_KEY") # Se necessário  
 url = f"https://api.exchangerate-api.com/v4/latest/{base\_currency}"  
 response = requests.get(url)  
 data = response.json()  
   
 if data and data["rates"] and target\_currency in data["rates"]:  
 rate = data["rates"][target\_currency]  
 return f"A taxa de câmbio de {base\_currency} para {target\_currency} é {rate}"  
 else:  
 return f"Não foi possível obter a taxa de câmbio para {base\_currency}/{target\_currency}"  
 except Exception as e:  
 return f"Erro ao obter a taxa de câmbio: {e}"  
  
# 2. Escolher o LLM que será o cérebro do nosso agente  
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0)  
  
# 3. Definir as ferramentas que o agente pode usar  
tools = [get\_exchange\_rate]  
  
# 4. Criar o Prompt do Agente  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "Você é um assistente especializado em cotação de moedas. Use a ferramenta 'get\_exchange\_rate' para responder a perguntas sobre taxas de câmbio."),  
 ("human", "{input}"),  
 ("placeholder", "{agent\_scratchpad}"),  
])  
  
# 5. Criar o Agente  
agent = create\_tool\_calling\_agent(llm, tools, prompt)  
  
# 6. Criar o Executor do Agente  
agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)  
  
# --- Execução ---  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("Agente de Cotação de Moedas pronto! Faça sua pergunta.")  
  
 perguntas = [  
 "Qual a taxa de câmbio de USD para BRL?",  
 "Quanto é 100 EUR em JPY?",  
 "Qual o valor do dólar hoje?",  
 "Qual o nome do meu gato?"  
 ]  
  
 for pergunta in perguntas:  
 print(f"\n> Pergunta: {pergunta}")  
 response = agent\_executor.invoke({"input": pergunta})  
 print(f"\n< Resposta Final: {response['output']}")

**Comando de Execução (Linux/macOS):**

cd exercicios/capitulo\_06/exercicio\_2  
chmod +x run.sh  
./run.sh

**Comando de Execução (Windows):**

cd exercicios\capitulo\_06\exercicio\_2  
call run.bat

**Saída Esperada (pode variar):**

Agente de Cotação de Moedas pronto! Faça sua pergunta.  
> Pergunta: Qual a taxa de câmbio de USD para BRL?  
< Resposta Final: ...  
> Pergunta: Quanto é 100 EUR em JPY?  
< Resposta Final: ...  
> Pergunta: Qual o valor do dólar hoje?  
< Resposta Final: ...  
> Pergunta: Qual o nome do meu gato?  
< Resposta Final: Não tenho informações sobre isso.

**Troubleshooting Comum:** \* requests.exceptions.ConnectionError: Verifique sua conexão com a internet. \* AuthenticationError: Verifique a chave de API do Gemini no .env. \* ModuleNotFoundError: Instale todas as dependências com uv add.

### Troubleshooting Comum

* Debug de ferramentas: teste funções isoladamente e verifique logs detalhados com verbose=True.
* Use tracing para inspeção de execuções e identificar gargalos.
* Instale todas as dependências e configure corretamente o .env.
* Proteja chaves de API e modularize ferramentas para facilitar manutenção.

### Pontos Chave

* Ferramentas permitem que LLMs interajam com sistemas externos e superem limitações.
* Agentes usam LLMs para raciocinar, decidir e executar ações com ferramentas.
* O padrão ReAct (Reason + Act) e prompts customizados são diferenciais para agentes autônomos eficientes.
* Exercícios práticos mostram integração com banco de dados, APIs externas e modularização.

### Resumo do Capítulo

Neste capítulo, você aprendeu como equipar LLMs com ferramentas externas, criando agentes autônomos e interativos no LangChain. Viu a diferença entre ferramentas e agentes, implementou agentes com ferramentas customizadas e APIs externas, e consolidou o aprendizado com exercícios práticos, troubleshooting e dicas de boas práticas.

### Teste seu Conhecimento

1. No LangChain, o que é uma Ferramenta (Tool)?
   1. Um modelo de linguagem.
   2. Uma função que o agente pode invocar para interagir com sistemas externos.
   3. Um arquivo de configuração.
   4. Um tipo de prompt especial.
2. Qual é o papel do agente em um sistema LangChain?
   1. Executar comandos de sistema.
   2. Raciocinar, decidir qual ferramenta usar e observar resultados.
   3. Gerar apenas texto sem interação externa.
   4. Armazenar dados em banco de dados.
3. O padrão ReAct (Reason + Act) envolve:
   1. Apenas raciocinar sobre a entrada do usuário.
   2. Raciocinar, agir usando ferramentas e observar resultados.
   3. Executar comandos sem raciocínio.
   4. Gerar prompts estáticos.
4. Qual erro pode ocorrer se a tabela ‘vendas’ não for criada corretamente?
   1. requests.exceptions.ConnectionError
   2. sqlite3.OperationalError: no such table: vendas
   3. ModuleNotFoundError
   4. AuthenticationError
5. Para consultar taxas de câmbio em tempo real, o agente precisa:
   1. Apenas um LLM.
   2. Uma ferramenta que acesse uma API externa.
   3. Um arquivo .env vazio.
   4. Um banco de dados local.
6. Como inspecionar o raciocínio do agente passo a passo?
   1. Usando agent\_executor.stream para visualizar cada etapa.
   2. Executando apenas o LLM sem ferramentas.
   3. Ignorando logs.
   4. Usando prompts estáticos.

**Respostas:** 1. b 2. b 3. b 4. b 5. b 6. a

### Projeto Hands-on: Orquestrando Ferramentas e Agentes

Coloque em prática os conceitos do capítulo criando um projeto Python que utiliza LangChain para construir um agente capaz de integrar múltiplas ferramentas: uma para análise de dados locais (ex: vendas) e outra para consulta de APIs externas (ex: cotação de moedas). Siga a estrutura sugerida:

* Estrutura de diretórios para agentes e ferramentas.
* Checklist de boas práticas: modularização, logging, versionamento, testes automatizados.
* Fluxograma do fluxo de decisão do agente.
* Dicas para integração de múltiplas ferramentas e fontes de dados.
* Passos para ativar tracing e compartilhar logs via GitHub.
* Experimente rodar o projeto em diferentes versões do Python usando pyenv.

## Capítulo 7: RAG - Geração Aumentada por Recuperação com LangChain

**Neste capítulo, você vai aprender:**

* O que é RAG (Retrieval-Augmented Generation) e por que essa técnica é fundamental para criar IAs com conhecimento atualizado e específico.
* Os conceitos essenciais por trás do RAG: Embeddings, Bancos de Dados Vetoriais (ChromaDB) e Chunking.
* Como usar a LangChain Expression Language (LCEL) para construir pipelines de RAG de forma declarativa e intuitiva.
* A implementação de um projeto prático: um chatbot que responde a perguntas sobre um documento PDF usando RAG, LangChain e o modelo Gemini do Google.

### O Problema: A Memória Curta da IA

Modelos de linguagem (LLMs) como o Gemini do Google são extremamente inteligentes, mas seu conhecimento é limitado ao que viram durante o treinamento. Eles não sabem sobre seus documentos privados ou eventos recentes. Para resolver isso, usamos uma técnica chamada RAG (Retrieval-Augmented Generation), que dá ao LLM uma “memória externa” para consultar.

Esse problema é conhecido como “memória curta” dos LLMs. Mesmo os modelos mais avançados, como o Gemini, GPT-4 ou Claude, só conseguem responder com base nas informações presentes em seus dados de treinamento, que geralmente têm um corte temporal (por exemplo, até 2023) e não incluem documentos privados, dados corporativos ou informações específicas do seu contexto. Isso significa que, se você perguntar sobre um relatório interno da sua empresa, um artigo científico recém-publicado ou até mesmo sobre um documento PDF que só você possui, o LLM não conseguirá responder de forma precisa — simplesmente porque ele nunca viu esse conteúdo.

Além disso, LLMs não têm acesso direto à internet ou a bancos de dados externos durante a inferência. Eles não podem buscar informações em tempo real ou consultar arquivos locais por conta própria. Isso limita bastante sua utilidade em aplicações que exigem respostas atualizadas, personalizadas ou baseadas em dados privados.

Outro desafio é o tamanho da janela de contexto: mesmo que você tente “colar” um documento inteiro no prompt, há um limite de tokens que o modelo pode processar de uma vez. Documentos longos, bases de conhecimento extensas ou múltiplos arquivos rapidamente ultrapassam esse limite, tornando inviável enviar tudo para o LLM analisar de uma só vez.

Por isso, surge a necessidade de uma abordagem que permita ao LLM acessar informações externas de forma eficiente, segura e contextualizada. É aí que entra o RAG: ele atua como um “ponteiro inteligente” para a memória externa, permitindo que o modelo recupere apenas os trechos mais relevantes de grandes volumes de dados e use esse contexto para gerar respostas fundamentadas, atualizadas e personalizadas.

Em resumo, a limitação dos LLMs em acessar conhecimento fora do seu treinamento é um obstáculo para aplicações práticas em ambientes corporativos, acadêmicos ou pessoais. O RAG resolve esse problema ao conectar o modelo a fontes de dados externas, ampliando sua utilidade e tornando-o realmente capaz de responder perguntas sobre qualquer conteúdo que você desejar.

### A Solução: RAG (Geração Aumentada por Recuperação)

RAG (Retrieval-Augmented Generation) é uma abordagem que conecta modelos de linguagem (LLMs) a fontes externas de conhecimento, permitindo que eles respondam perguntas com base em informações que não estavam presentes em seus dados de treinamento. O processo de RAG pode ser dividido em três etapas principais:

1. **Recuperar (Retrieve):** Quando uma pergunta é feita, o sistema busca os trechos mais relevantes de uma base de conhecimento externa (como documentos, wikis, PDFs, bancos de dados, etc.). Essa busca é feita por similaridade semântica, usando embeddings para encontrar os textos mais próximos do significado da pergunta.
2. **Aumentar (Augment):** Os trechos recuperados são inseridos em um prompt junto com a pergunta original. Esse contexto adicional fornece ao LLM informações específicas e atualizadas, ampliando sua capacidade de resposta.
3. **Gerar (Generate):** O prompt enriquecido é enviado ao LLM, que utiliza tanto o contexto recuperado quanto seu conhecimento prévio para gerar uma resposta precisa, fundamentada e contextualizada.

Essa arquitetura reduz significativamente o risco de “alucinações” (respostas inventadas) e permite que a IA utilize dados privados, corporativos ou atualizados, tornando-a muito mais útil em cenários do mundo real.

### As Ferramentas Essenciais

Para implementar um pipeline RAG eficiente, alguns componentes são fundamentais:

* **Embeddings:** São representações numéricas de textos, criadas por modelos especializados. Embeddings permitem que o significado de frases, sentenças ou documentos seja comparado matematicamente. Textos com significados semelhantes terão embeddings próximos no espaço vetorial. Isso é essencial para buscar informações relevantes por similaridade de significado, e não apenas por palavras-chave.
* **Banco de Dados Vetorial (ChromaDB):** Um banco de dados vetorial armazena embeddings e permite buscas rápidas por similaridade. O ChromaDB, por exemplo, é uma solução open source leve e fácil de usar, ideal para prototipagem e produção. Ele permite indexar grandes volumes de dados e recuperar rapidamente os chunks mais relevantes para cada pergunta.
* **Chunking:** Como LLMs têm um limite de tokens (quantidade de texto que conseguem processar de uma vez), é necessário dividir documentos longos em pedaços menores chamados chunks. O chunking garante que cada parte do texto seja gerenciável e que informações importantes não sejam perdidas. Parâmetros como chunk\_size e chunk\_overlap ajudam a equilibrar contexto e granularidade.
* **LangChain Expression Language (LCEL):** LCEL é uma linguagem declarativa baseada em operadores de pipeline (|) que facilita a composição de fluxos de processamento em IA. Com LCEL, você pode conectar loaders, splitters, retrievers, prompts, LLMs e parsers de forma enxuta e intuitiva, criando pipelines robustos e fáceis de manter.

Essas ferramentas, quando combinadas, permitem criar aplicações de IA capazes de responder perguntas sobre qualquer conteúdo, seja ele um documento PDF, um arquivo Markdown, uma planilha CSV ou uma base de dados inteira. O RAG transforma o LLM em um verdadeiro “especialista” sobre os dados que você fornecer, tornando-o útil para aplicações corporativas, acadêmicas e pessoais.

### Projeto Prático: Chatbot de Documentos com LCEL e Gemini

Vamos construir uma aplicação de linha de comando em Python que lê um documento PDF e responde a perguntas sobre seu conteúdo.

Use o comando abaixo para iniciar o projeto:

uv init

Assim, você terá a seguinte estrutura de pastas:

exercicios/  
└── capitulo\_07/  
 └── exercicio\_1/  
 ├── app.py  
 ├── seu\_documento.pdf  
 ├── pyproject.toml  
 └── .env

#### Instalação de Dependências

Crie um ambiente virtual e instale as bibliotecas com uv:

# Navegue até o diretório do exercício  
cd exercicios/capitulo\_07/exercicio\_1  
  
# Crie o ambiente virtual com uv  
uv venv  
  
# Ative o ambiente virtual  
# No Linux/macOS:  
source .venv/bin/activate  
# No Windows:  
.venv\Scripts\activate  
  
# Instale as dependências  
uv add langchain langchain-google-genai langchain-chroma pypdf python-dotenv

#### Configuração da Chave de API

Você precisará de uma chave de API do Google AI Studio. Crie um arquivo chamado .env na pasta do projeto e adicione sua chave:

GOOGLE\_API\_KEY="SUA\_CHAVE\_API\_AQUI"

#### O Código Completo (main.py)

Este script único contém toda a lógica. Ele carrega o PDF, o divide em chunks, armazena no ChromaDB e cria a cadeia RAG com LCEL para interagir com o modelo Gemini.

# exercicios/capitulo\_07/exercicio\_1/main.py  
import os  
from dotenv import load\_dotenv  
  
# LangChain Imports  
from langchain\_google\_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings  
from langchain\_community.document\_loaders import PyPDFLoader  
from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
from langchain\_chroma import Chroma  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.runnables import RunnablePassthrough  
from langchain\_core.output\_parsers import StrOutputParser  
  
# Carrega as variáveis de ambiente do arquivo .env  
load\_dotenv()  
  
# --- 1. FASE DE INDEXAÇÃO (INGESTÃO DE DADOS) ---  
  
# Carregar o documento PDF  
print("Carregando PDF...")  
loader = PyPDFLoader("seu\_documento.pdf")  
docs = loader.load()  
  
# Dividir o documento em chunks  
print("Dividindo documento em chunks...")  
text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  
splits = text\_splitter.split\_documents(docs)  
  
# Inicializar o modelo de embeddings do Google  
print("Inicializando modelo de embeddings...")  
embedding\_model = GoogleGenerativeAIEmbeddings(model="models/embedding-001")  
  
# Criar o banco de dados vetorial ChromaDB com os chunks  
print("Criando e armazenando embeddings no ChromaDB...")  
vectorstore = Chroma.from\_documents(documents=splits, embedding=embedding\_model)  
  
# --- 2. FASE DE RECUPERAÇÃO E GERAÇÃO (RAG) ---  
  
# Criar o retriever a partir do vectorstore  
retriever = vectorstore.as\_retriever()  
  
# Inicializar o LLM Gemini  
print("Inicializando LLM Gemini...")  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)  
  
# Template do Prompt  
prompt\_template = """  
Use o seguinte contexto para responder à pergunta.  
Se você não sabe a resposta, apenas diga que não sabe. Não tente inventar uma resposta.  
  
Contexto:  
{context}  
  
Pergunta:  
{question}  
  
Resposta útil:  
"""  
prompt = ChatPromptTemplate.from\_template(prompt\_template)  
  
# Função para formatar os documentos recuperados  
def format\_docs(docs):  
 return "\n\n".join(doc.page\_content for doc in docs)  
  
# Construir a cadeia RAG usando LangChain Expression Language (LCEL)  
rag\_chain = (  
 {"context": retriever | format\_docs, "question": RunnablePassthrough()}  
 | prompt  
 | llm  
 | StrOutputParser()  
)  
  
# --- 3. INTERAÇÃO COM O USUÁRIO ---  
  
print("\nSetup completo! O chatbot está pronto.")  
print("Digite 'sair' para encerrar.")  
  
while True:  
 user\_question = input("\nFaça sua pergunta: ")  
 if user\_question.lower() == 'sair':  
 break  
   
 print("\nGerando resposta...")  
 response = rag\_chain.invoke(user\_question)  
 print("--- Resposta ---")  
 print(response)  
 print("----------------")  
  
# Limpeza (opcional): remove o banco de dados vetorial ao sair  
vectorstore.delete\_collection()  
print("\nChatbot encerrado.")

### Executando e Testando

Para executar sua aplicação, simplesmente rode o script no seu terminal:

uv run main.py

A aplicação irá processar seu PDF e, em seguida, solicitará que você faça perguntas.

**Exemplo de Interação:**

1. Carregue um PDF sobre a história do Brasil.
2. Execute uv run main.py.
3. Quando solicitado, pergunte: “Quem descobriu o Brasil?”
4. O chatbot usará o conteúdo do seu PDF para responder de forma precisa.

### Exemplos Adicionais

Para fixar ainda mais os conceitos, preparei outros exemplos de como usar RAG com diferentes tipos de documentos e formatos.

#### Lendo Arquivos Markdown (.md)

Por que vetorizar arquivos em formato markdown? O processo para ler arquivos Markdown é muito semelhante ao de PDFs. A principal diferença é o uso do UnstructuredMarkdownLoader.

A vantagem prática é que markdowns são compreensíveis por humanos e LLMs ao mesmo tempo. Frequentemente, usamos markdowns em uma wiki, por exemplo, para documentar projetos, o que facilita a consulta e a atualização de informações.

Outra vantagem é o caracter delimitador, o que permite que o LLM entenda melhor a estrutura do documento, como títulos, subtítulos e listas.

**Sobre o caracter delimitador, chunk size e overlap:**

O caracter delimitador é um elemento do texto (como títulos #, ##, listas ou tabelas) usado para dividir o documento em partes menores (chunks) de forma lógica. No caso de arquivos Markdown, delimitadores como títulos ajudam o splitter a identificar onde começa e termina cada seção, permitindo que cada chunk corresponda a uma parte coesa do conteúdo. Isso mantém o contexto e a estrutura do documento, facilitando a recuperação de informações relevantes durante o processo de RAG.

Ao configurar o splitter, você pode definir:

* **chunk\_size**: o número máximo de caracteres em cada chunk. Por exemplo, chunk\_size=1000 cria pedaços de até 1000 caracteres. Ajuste esse valor conforme o tamanho médio das seções do seu documento e a janela de contexto do LLM.
* **chunk\_overlap**: quantos caracteres do final de um chunk serão repetidos no início do próximo. Por exemplo, chunk\_overlap=200 garante que informações importantes na fronteira entre dois chunks não sejam perdidas.

Exemplo de uso com splitter de texto:

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  
splits = text\_splitter.split\_documents(docs)

No caso de Markdown, você pode usar delimitadores de cabeçalho para dividir por seções:

headers\_to\_split\_on = [  
 ("#", "Título"),  
 ("##", "Subtítulo"),  
 ("###", "Subsubtítulo"),  
]  
text\_splitter = MarkdownHeaderTextSplitter(  
 headers\_to\_split\_on=headers\_to\_split\_on,  
 strip\_headers=True  
)  
splits = text\_splitter.split\_documents(docs)

O ajuste desses parâmetros depende do equilíbrio entre contexto suficiente para o LLM e o limite de tokens do modelo. Teste diferentes valores para encontrar o melhor resultado para seu caso de uso.

O processo para ler e estruturar dados a partir de arquivos Markdown é bastante flexível, pois o formato markdown permite organizar informações em listas, tabelas e seções bem definidas. Ao usar o UnstructuredMarkdownLoader, cada seção, lista ou bloco de texto pode ser tratado como um documento separado, facilitando a recuperação de informações específicas durante o processo de RAG.

Por exemplo, se você possui um arquivo markdown com uma tabela de dados ou listas de tópicos, o loader irá dividir automaticamente esses elementos em chunks lógicos. Isso permite que perguntas direcionadas sobre itens, tópicos ou valores específicos sejam respondidas com precisão, aproveitando a estrutura do markdown para melhorar a segmentação e a relevância das respostas.

Além disso, você pode personalizar o pré-processamento do markdown para extrair apenas certas seções ou tipos de conteúdo, tornando o pipeline ainda mais adaptado ao seu caso de uso. Dessa forma, o markdown não serve apenas como um formato de documentação, mas também como uma fonte estruturada de dados para aplicações de IA.

**Estrutura de Pastas:**

exercicios/  
└── capitulo\_07/  
 └── exercicio\_2/  
 ├── main.py  
 ├── seu\_documento.md  
 ├── pyproject.toml  
 └── .env

**Código Completo (main.py):**

# exercicios/capitulo\_07/exercicio\_2/main.py  
import os  
from dotenv import load\_dotenv  
  
# LangChain Imports  
from langchain\_google\_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings  
from langchain\_community.document\_loaders import UnstructuredMarkdownLoader  
from langchain\_text\_splitters import MarkdownHeaderTextSplitter  
from langchain\_chroma import Chroma  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.runnables import RunnablePassthrough  
from langchain\_core.output\_parsers import StrOutputParser  
  
# Carrega as variáveis de ambiente do arquivo .env  
load\_dotenv()  
  
# --- 1. FASE DE INDEXAÇÃO (INGESTÃO DE DADOS) ---  
  
  
# Carregar o conteúdo do arquivo Markdown  
print("Carregando Markdown...")  
with open("universo.md", "r", encoding="utf-8") as f:  
 markdown\_text = f.read()  
  
# Definir os delimitadores de chunk para capturar seções inteiras por subtítulo  
headers\_to\_split\_on = [  
 ("#", "Título"),  
 ("##", "Subtítulo"),  
 ("###", "Subsubtítulo"),  
]  
  
print("Dividindo documento em chunks por subtítulo...")  
text\_splitter = MarkdownHeaderTextSplitter(  
 headers\_to\_split\_on=headers\_to\_split\_on,  
 strip\_headers=True  
)  
  
# O método correto é split\_text, que retorna uma lista de Document  
splits = text\_splitter.split\_text(markdown\_text)  
  
# Inicializar o modelo de embeddings do Google  
print("Inicializando modelo de embeddings...")  
embedding\_model = GoogleGenerativeAIEmbeddings(model="models/embedding-001")  
  
# Criar o banco de dados vetorial ChromaDB com os chunks  
print("Criando e armazenando embeddings no ChromaDB...")  
vectorstore = Chroma.from\_documents(documents=splits, embedding=embedding\_model)  
  
# --- 2. FASE DE RECUPERAÇÃO E GERAÇÃO (RAG) ---  
  
# Criar o retriever a partir do vectorstore  
retriever = vectorstore.as\_retriever()  
  
# Inicializar o LLM Gemini  
print("Inicializando LLM Gemini...")  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)  
  
# Template do Prompt  
prompt\_template = """  
Use o seguinte contexto para responder à pergunta.  
Se você não sabe a resposta, apenas diga que não sabe. Não tente inventar uma resposta.  
  
Contexto:  
{context}  
  
Pergunta:  
{question}  
  
Resposta útil:  
"""  
prompt = ChatPromptTemplate.from\_template(prompt\_template)  
  
# Função para formatar os documentos recuperados  
def format\_docs(docs):  
 return "\n\n".join(doc.page\_content for doc in docs)  
  
# Construir a cadeia RAG usando LangChain Expression Language (LCEL)  
rag\_chain = (  
 {"context": retriever | format\_docs, "question": RunnablePassthrough()}  
 | prompt  
 | llm  
 | StrOutputParser()  
)  
  
# --- 3. INTERAÇÃO COM O USUÁRIO ---  
  
print("\nSetup completo! O chatbot está pronto.")  
print("Digite 'sair' para encerrar.")  
  
while True:  
 user\_question = input("\nFaça sua pergunta: ")  
 if user\_question.lower() == 'sair':  
 break  
   
 print("\nGerando resposta...")  
 response = rag\_chain.invoke(user\_question)  
 print("--- Resposta ---")  
 print(response)  
 print("----------------")  
  
# Limpeza (opcional): remove o banco de dados vetorial ao sair  
vectorstore.delete\_collection()  
print("\nChatbot encerrado.")

**Como funciona a divisão por subtítulo:**  
O MarkdownHeaderTextSplitter permite que cada chunk corresponda exatamente a uma seção do markdown, delimitada por títulos como ## ou ###. Assim, quando você faz uma pergunta, o sistema pode recuperar o chunk inteiro de um subtítulo relevante, garantindo que a resposta esteja contextualizada dentro daquela seção do documento.

**Dependências Adicionais:**

Adicione unstructured e markdown ao seu pyproject.toml:

[project]  
# ...  
dependencies = [  
 # ...  
 "unstructured",  
 "markdown",  
]

#### Lendo Arquivos CSV (.csv)

Para arquivos CSV, usamos o CSVLoader. Ele trata cada linha do CSV como um documento separado.

**Como funciona a leitura de CSV:**

O formato CSV (Comma-Separated Values) é amplamente utilizado para armazenar dados tabulares, como planilhas ou exportações de bancos de dados. Cada linha do arquivo representa um registro (ou entrada), e cada coluna representa um campo desse registro.

Ao usar o CSVLoader da LangChain, cada linha do CSV é carregada como um documento independente. Isso significa que, para cada registro, você terá um chunk separado no seu banco vetorial. Esse comportamento é útil para buscas precisas em dados estruturados, como perguntas sobre clientes, produtos, transações, etc.

**Exemplo de uso:**

from langchain\_community.document\_loaders.csv\_loader import CSVLoader  
  
# Carregar o documento CSV  
loader = CSVLoader(file\_path="seus\_dados.csv")  
docs = loader.load()  
  
# docs será uma lista de Document, cada um representando uma linha do CSV

Se o seu CSV for muito grande, você pode aplicar chunking adicional em campos de texto longos, mas normalmente cada linha já é um chunk lógico.

**Dica:** Os objetos Document criados pelo loader incluem os dados de cada linha e também podem conter metadados, como o número da linha ou o nome das colunas. Isso facilita a recuperação de informações específicas e a exibição de resultados contextualizados.

**Quando usar CSVLoader:** - Quando você tem dados tabulares e quer permitir buscas por registros específicos. - Quando precisa de granularidade por linha (ex: perguntas sobre um cliente ou transação específica).

Se quiser personalizar o pré-processamento, pode manipular os dados antes de criar os embeddings, por exemplo, filtrando colunas ou combinando campos em um texto único para cada registro.

**Estrutura de Pastas:**

exercicios/  
└── capitulo\_07/  
 └── exercicio\_3/  
 ├── app\_csv.py  
 ├── seus\_dados.csv  
 ├── pyproject.toml  
 └── .env

**Código Completo (main.py):**

# exercicios/capitulo\_07/exercicio\_3/main.py  
import os  
from dotenv import load\_dotenv  
  
# LangChain Imports  
from langchain\_google\_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings  
from langchain\_community.document\_loaders.csv\_loader import CSVLoader  
from langchain\_text\_splitters import RecursiveCharacterTextSplitter  
from langchain\_chroma import Chroma  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.runnables import RunnablePassthrough  
from langchain\_core.output\_parsers import StrOutputParser  
  
# Carrega as variáveis de ambiente do arquivo .env  
load\_dotenv()  
  
# --- 1. FASE DE INDEXAÇÃO (INGESTÃO DE DADOS) ---  
  
# Carregar o documento CSV  
print("Carregando CSV...")  
loader = CSVLoader(file\_path="seus\_dados.csv")  
docs = loader.load()  
  
# Se necessário, aplicar chunking em campos de texto longos  
# Para a maioria dos casos, cada linha já é um chunk lógico  
# Exemplo de uso do splitter (opcional):  
# text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  
# splits = text\_splitter.split\_documents(docs)  
splits = docs # normalmente, cada linha do CSV já é um chunk  
  
# Inicializar o modelo de embeddings do Google  
print("Inicializando modelo de embeddings...")  
embedding\_model = GoogleGenerativeAIEmbeddings(model="models/embedding-001")  
  
# Criar o banco de dados vetorial ChromaDB com os chunks  
print("Criando e armazenando embeddings no ChromaDB...")  
vectorstore = Chroma.from\_documents(documents=splits, embedding=embedding\_model)  
  
# --- 2. FASE DE RECUPERAÇÃO E GERAÇÃO (RAG) ---  
  
# Criar o retriever a partir do vectorstore  
retriever = vectorstore.as\_retriever()  
  
# Inicializar o LLM Gemini  
print("Inicializando LLM Gemini...")  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)  
  
# Template do Prompt  
prompt\_template = """  
Use o seguinte contexto para responder à pergunta.  
Se você não sabe a resposta, apenas diga que não sabe. Não tente inventar uma resposta.  
  
Contexto:  
{context}  
  
Pergunta:  
{question}  
  
Resposta útil:  
"""  
prompt = ChatPromptTemplate.from\_template(prompt\_template)  
  
# Função para formatar os documentos recuperados  
def format\_docs(docs):  
 return "\n\n".join(doc.page\_content for doc in docs)  
  
# Construir a cadeia RAG usando LangChain Expression Language (LCEL)  
rag\_chain = (  
 {"context": retriever | format\_docs, "question": RunnablePassthrough()}  
 | prompt  
 | llm  
 | StrOutputParser()  
)  
  
# --- 3. INTERAÇÃO COM O USUÁRIO ---  
  
print("\nSetup completo! O chatbot está pronto.")  
print("Digite 'sair' para encerrar.")  
  
while True:  
 user\_question = input("\nFaça sua pergunta: ")  
 if user\_question.lower() == 'sair':  
 break  
   
 print("\nGerando resposta...")  
 response = rag\_chain.invoke(user\_question)  
 print("--- Resposta ---")  
 print(response)  
 print("----------------")  
  
# Limpeza (opcional): remove o banco de dados vetorial ao sair  
vectorstore.delete\_collection()  
print("\nChatbot encerrado.")

**Sobre o delimitador e chunking em CSV:**  
No caso de arquivos CSV, cada linha já funciona como um chunk lógico, pois representa um registro completo da tabela. Se algum campo do CSV contiver textos longos (por exemplo, descrições ou observações), você pode aplicar um splitter adicional nesses campos usando o RecursiveCharacterTextSplitter para garantir que o conteúdo não ultrapasse o limite de contexto do LLM. No entanto, para a maioria dos usos, cada linha do CSV é suficiente como unidade de recuperação, facilitando buscas precisas e contextualizadas.

### Pontos Chave

* RAG é uma técnica poderosa para estender o conhecimento de LLMs com dados privados ou atualizados.
* O processo de RAG consiste em Recuperar, Aumentar e Gerar.
* Ferramentas como ChromaDB para armazenamento vetorial e LCEL para construção de pipelines são essenciais para implementar RAG com LangChain.
* A combinação de LangChain com modelos como o Gemini permite criar aplicações de IA sofisticadas com relativa simplicidade.

### Resumo do Capítulo

Neste capítulo, você mergulhou no mundo da Geração Aumentada por Recuperação (RAG). Aprendeu os conceitos fundamentais que tornam essa técnica possível e construiu um chatbot funcional do zero, capaz de conversar sobre o conteúdo de um documento PDF. Este projeto prático demonstrou como LangChain, ChromaDB e os modelos Gemini do Google se unem para criar soluções de IA inteligentes e com conhecimento específico.

Além do exemplo com PDF, você também aprendeu como aplicar RAG em outros formatos de documentos, como Markdown e CSV. Viu como o uso de delimitadores (como títulos em Markdown ou linhas em CSV) permite dividir o conteúdo em chunks lógicos, facilitando a recuperação de informações relevantes e mantendo o contexto necessário para respostas precisas.

O capítulo detalhou a importância de ajustar o tamanho do chunk e o overlap para equilibrar contexto e eficiência, além de mostrar como cada tipo de documento pode exigir estratégias diferentes de chunking. Você viu exemplos práticos de como configurar o splitter para cada caso, garantindo que tanto textos longos quanto dados tabulares possam ser usados em pipelines de RAG.

Também foram apresentadas dicas para produção, como persistir o banco vetorial em disco, trabalhar com múltiplos documentos e enriquecer as respostas do chatbot com metadados (por exemplo, indicando a página ou linha de origem da informação). Essas práticas tornam o sistema mais robusto, eficiente e pronto para aplicações reais.

Com esse conhecimento, você está preparado para criar aplicações de IA que vão além do trivial, integrando dados próprios, estruturados ou não, e oferecendo respostas fundamentadas, seguras e contextualizadas para os usuários.

### Teste seu Conhecimento

1. O que significa a sigla RAG?
   1. Random Access Generation
   2. Retrieval-Augmented Generation
   3. Real-time Augmented Grammar
   4. Recursive Agent Generation
2. Qual é a principal função de um banco de dados vetorial como o ChromaDB em um pipeline RAG?
   1. Armazenar o histórico da conversa.
   2. Executar o modelo de linguagem.
   3. Armazenar e buscar embeddings de texto por similaridade.
   4. Dividir documentos em chunks.
3. O que é “chunking” e por que é importante?
   1. O processo de converter texto em embeddings.
   2. O processo de quebrar textos longos em pedaços menores para facilitar a recuperação e respeitar o limite de contexto do LLM.
   3. O processo de enviar um prompt para o LLM.
   4. O processo de avaliar a qualidade da resposta do LLM.
4. Qual o papel dos delimitadores em arquivos Markdown e CSV no contexto de RAG?
   1. Servem apenas para estilizar o texto.
   2. Permitem dividir o documento em chunks lógicos, facilitando a segmentação e a recuperação de informações.
   3. São ignorados pelo pipeline.
   4. Servem para criptografar o conteúdo.
5. Para que serve a LangChain Expression Language (LCEL)?
   1. Para treinar modelos de linguagem.
   2. Para criar interfaces de usuário para chatbots.
   3. Para compor cadeias (chains) de processamento de forma declarativa e enxuta.
   4. Para definir o estilo de escrita do LLM.
6. Qual das opções abaixo é uma boa prática para produção de pipelines RAG?
   1. Manter o banco vetorial apenas em memória.
   2. Persistir o banco vetorial em disco, trabalhar com múltiplos documentos e enriquecer as respostas com metadados.
   3. Não usar chunking.
   4. Ignorar o contexto do usuário.

**Respostas:** 1. b 2. c 3. b 4. b 5. c 6. b

### Projeto Hands-on: Expandindo o Chatbot RAG

Agora é sua vez de aprimorar o projeto que construímos. Tente implementar as seguintes melhorias:

* **Múltiplos Documentos:** Modifique o código para carregar e processar múltiplos arquivos de uma pasta. (Dica: use um loop e a função os.listdir()).
* **Persistência:** O ChromaDB criado neste exemplo é “efêmero” (em memória). Pesquise na documentação do LangChain e do ChromaDB como torná-lo persistente, salvando os dados em disco para não precisar reprocessar o PDF toda vez. (Dica: use persist\_directory ao criar o Chroma).
* **Adicionar Fontes:** Modifique a cadeia RAG para também retornar de qual página do documento a informação foi extraída. (Dica: o retriever retorna objetos Document que contêm metadados, incluindo a página).

## Capítulo 8: Testes, Debugging e Otimização de Aplicações LangChain

**Neste capítulo, você vai aprender:**

* Como testar aplicações LangChain de forma eficiente, cobrindo unitários, integração e funcionais.
* Técnicas de debugging, tracing e monitoramento com LangSmith e ferramentas nativas do LangChain.
* Estratégias de otimização de desempenho, custo e confiabilidade em pipelines de LLM.
* Estudo de caso prático integrando testes, debugging e tracing.
* Exercícios práticos, tabelas comparativas e fluxogramas para fixação dos conceitos.

### 1. Por que Testar e Depurar Aplicações LangChain?

Construir aplicações com LLMs é diferente do desenvolvimento tradicional: as respostas são probabilísticas, o contexto é dinâmico e bugs podem ser sutis. Testes e debugging são essenciais para garantir qualidade, segurança e previsibilidade.

Imagine um chatbot que responde de forma inesperada após uma atualização de prompt. Sem testes automatizados e tracing, encontrar a causa pode ser como procurar uma agulha no palheiro. O LangChain oferece recursos para tornar esse processo transparente e eficiente.

### 2. Testes em LangChain: Unitários, Integração e Funcionais

**Ferramentas recomendadas:** - pytest (Python) - unittest (Python) - LangSmith (para avaliação e tracing)

**Tabela: Tipos de Teste em Aplicações LangChain**

| Tipo | O que cobre? | Ferramenta principal | Exemplo prático |
| --- | --- | --- | --- |
| Unitário | Funções isoladas, prompts | pytest/unittest | Testar parser |
| Integração | Chains, pipelines | pytest + LangSmith | Testar chain |
| Funcional | Comportamento do app | LangSmith | Teste end-to-end |

**Exemplo de Teste Unitário:**

# capitulo\_08/exercicio\_1/main.py  
import pytest  
from langchain\_core.output\_parsers import StrOutputParser  
  
def test\_str\_output\_parser():  
 parser = StrOutputParser()  
 output = parser.invoke({"content": "Olá, mundo!"})  
 assert output == "Olá, mundo!"

**Como executar:**

uv add pytest  
pytest capitulo\_08/exercicio\_1/main.py

**Resultado esperado:**

.  
1 passed in 0.01s

### 3. Debugging: Logs, Verbose e Inspeção de Chains

O LangChain permite ativar logs detalhados para inspecionar o fluxo de execução:

# capitulo\_08/exercicio\_2/main.py  
from langchain.globals import set\_debug, set\_verbose  
  
set\_debug(True) # Ativa logs detalhados  
set\_verbose(True) # Mostra entradas e saídas legíveis  
# ... código da chain ...

**Fluxograma textual do ciclo de debugging:**

1. Ativar logs (set\_debug, set\_verbose)
2. Executar chain/agent
3. Inspecionar entradas, saídas e erros
4. Corrigir e repetir

**Exemplo de saída de log:**

[chain/start] [chain:AgentExecutor] Entering Chain run with input:  
{"input": "Quem dirigiu Oppenheimer?"}  
[llm/start] [chain:AgentExecutor > llm:ChatOpenAI] Entering LLM run...  
[llm/end] ...

### 4. Tracing e Monitoramento com LangSmith

O LangSmith é a plataforma oficial para tracing, avaliação e observabilidade de aplicações LangChain. Ele permite inspecionar cada passo de uma chain, visualizar erros, latência e até associar feedback do usuário.

**Como habilitar tracing com LangSmith:**

# capitulo\_08/exercicio\_3/main.py  
import os, getpass  
os.environ["LANGSMITH\_TRACING"] = "true"  
os.environ["LANGSMITH\_API\_KEY"] = getpass.getpass("Digite sua LangSmith API key: ")

**No terminal:**

export LANGSMITH\_TRACING="true"  
export LANGSMITH\_API\_KEY="<sua\_api\_key>"

**Exemplo prático:**

# capitulo\_08/exercicio\_4/main.py  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_core.prompts import ChatPromptTemplate  
  
prompt = ChatPromptTemplate.from\_template("Resuma: {texto}")  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
chain = prompt | llm  
  
response = chain.invoke({"texto": "LangChain facilita o desenvolvimento de apps com LLMs."})  
print(response.content)

**Após executar, acesse o painel do LangSmith para visualizar o trace completo.**

### 5. Otimização: Desempenho, Custo e Confiabilidade

**Principais estratégias:** - Reduzir número de chamadas ao LLM (cache, batch) - Ajustar temperatura e parâmetros do modelo - Usar memory e retrievers para evitar repetições - Monitorar latência e custo via tracing

**Tabela: Estratégias de Otimização**

| Estratégia | Benefício | Trade-off |
| --- | --- | --- |
| Cache de respostas | Menos custo | Pode perder contexto |
| Batch de queries | Mais rápido | Respostas menos personalizadas |
| Ajuste de temp. | Mais previsível | Menos criatividade |

### 6. Estudo de Caso: Chatbot com Bugs, Testes e Tracing

**Cenário:** Um chatbot que responde perguntas sobre filmes, mas apresenta bugs de contexto e respostas erradas.

**Passos:** 1. Escrever chain com bug proposital 2. Criar teste unitário para identificar erro 3. Ativar logs e tracing para localizar bug 4. Corrigir e otimizar

**Exemplo de código:**

# capitulo\_08/exercicio\_5/main.py  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
  
prompt = ChatPromptTemplate.from\_template("Quem dirigiu {filme}?")  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
chain = prompt | llm  
  
# Bug: não trata filmes desconhecidos  
resposta = chain.invoke({"filme": "Oppenheimer"})  
print(resposta.content)

**Teste unitário para o bug:**

# capitulo\_08/exercicio\_6/main.py  
import pytest  
from capitulo\_08.exercicio\_5.main import chain  
  
def test\_filme\_desconhecido():  
 resposta = chain.invoke({"filme": "FilmeInexistente"})  
 assert "Desculpe" in resposta.content or resposta.content == ""

**Como executar:**

pytest capitulo\_08/exercicio\_6/main.py

**Resultado esperado:** Teste falha, indicando o bug. Após correção, o teste deve passar.

### Hands-on: Exercício 1 — Teste Unitário de Parser

* **Objetivo:** Testar o parser de saída de LLM.
* **Nome do Arquivo:** capitulo\_08/exercicio\_1/main.py
* **Dependências:** pytest, langchain-core
* **Comando de Instalação:**

uv add pytest langchain-core

**Comando de Execução (Linux/macOS):**

pytest capitulo\_08/exercicio\_1/main.py

**Comando de Execução (Windows):**

pytest capitulo\_08\exercicio\_1\main.py

**Saída Esperada (pode variar):**

.  
1 passed in 0.01s

**Troubleshooting Comum:** \* ModuleNotFoundError: Verifique se todas as dependências foram instaladas corretamente. \* Erros de sintaxe: Confira se o código está igual ao exemplo.

### Hands-on: Exercício 2 — Debugging de Chains

* **Objetivo:** Ativar logs detalhados e inspecionar execução de chains.
* **Nome do Arquivo:** capitulo\_08/exercicio\_2/main.py
* **Dependências:** langchain
* **Comando de Instalação:**

uv add langchain

**Comando de Execução (Linux/macOS):**

python capitulo\_08/exercicio\_2/main.py

**Comando de Execução (Windows):**

python capitulo\_08\exercicio\_2\main.py

**Saída Esperada (pode variar):**

[chain/start] ...  
[llm/start] ...  
[llm/end] ...

**Troubleshooting Comum:** \* ModuleNotFoundError: Verifique se todas as dependências foram instaladas corretamente. \* Logs não aparecem: Certifique-se de que set\_debug(True) e set\_verbose(True) estão ativados.

### Hands-on: Exercício 3 — Tracing com LangSmith

* **Objetivo:** Habilitar tracing e monitorar execução de chains.
* **Nome do Arquivo:** capitulo\_08/exercicio\_3/main.py
* **Dependências:** langchain, langchain-google-genai
* **Comando de Instalação:**

uv add langchain langchain-google-genai

**Comando de Execução (Linux/macOS):**

export LANGSMITH\_TRACING="true"  
export LANGSMITH\_API\_KEY="<sua\_api\_key>"  
python capitulo\_08/exercicio\_3/main.py

**Comando de Execução (Windows):**

set LANGSMITH\_TRACING=true  
set LANGSMITH\_API\_KEY=<sua\_api\_key>  
python capitulo\_08\exercicio\_3\main.py

**Saída Esperada (pode variar):**

Resuma: ...  
[Trace disponível no painel LangSmith]

**Troubleshooting Comum:** \* AuthenticationError: Verifique se a chave de API está correta. \* Trace não aparece: Certifique-se de que as variáveis de ambiente estão setadas.

### Hands-on: Exercício 4 — Teste de Bug em Chatbot

* **Objetivo:** Identificar e corrigir bug em chatbot.
* **Nome do Arquivo:** capitulo\_08/exercicio\_5/main.py e capitulo\_08/exercicio\_6/main.py
* **Dependências:** pytest, langchain-core, langchain-google-genai
* **Comando de Instalação:**

uv add pytest langchain-core langchain-google-genai

**Comando de Execução (Linux/macOS):**

pytest capitulo\_08/exercicio\_6/main.py

**Comando de Execução (Windows):**

pytest capitulo\_08\exercicio\_6\main.py

**Saída Esperada (pode variar):**

F  
1 failed in 0.01s

**Troubleshooting Comum:** \* Teste falha: Corrija o bug no chatbot e rode novamente. \* ModuleNotFoundError: Verifique se todas as dependências foram instaladas corretamente.

### Pontos Chave

* Testes unitários, integração e funcionais garantem qualidade em aplicações LangChain.
* Debugging e tracing são essenciais para identificar e corrigir bugs em pipelines de LLM.
* Otimização de desempenho e custo pode ser feita com cache, batch e ajuste de parâmetros.
* Ferramentas como pytest, LangSmith e logs detalhados são fundamentais para o ciclo de desenvolvimento.

### Resumo do Capítulo

Neste capítulo, você aprendeu a testar, debugar, traçar e otimizar aplicações LangChain, usando ferramentas modernas e práticas recomendadas. O domínio desses recursos é essencial para construir aplicações robustas, seguras e eficientes com LLMs.

### Teste seu Conhecimento

1. Qual comando ativa logs detalhados no LangChain?
   1. set\_verbose(False)
   2. set\_debug(True)
   3. set\_trace(True)
   4. enable\_logs()
2. Como habilitar tracing com LangSmith?
   1. Instalar pytest
   2. Setar LANGSMITH\_TRACING e LANGSMITH\_API\_KEY
   3. Ativar cache de respostas
   4. Usar ConversationBufferMemory
3. Qual estratégia reduz custo em aplicações LLM?
   1. Aumentar temperatura
   2. Cache de respostas
   3. Ignorar logs
   4. Usar apenas testes unitários
4. O que diferencia um teste unitário de um teste funcional?
   1. Teste unitário cobre funções isoladas, funcional cobre comportamento do app
   2. Teste funcional cobre apenas chains
   3. Ambos são iguais
   4. Teste unitário cobre apenas LLMs
5. Como inspecionar a execução de uma chain passo a passo?
   1. Ativar logs/tracing
   2. Ignorar erros
   3. Usar cache
   4. Executar sem testes

**Respostas:** 1. b 2. b 3. b 4. a 5. a

### Projeto Hands-on: Testes, Debugging e Tracing em Chatbot

Coloque em prática os conceitos do capítulo criando um projeto Python com LangChain que implementa um chatbot testável, com testes unitários e funcionais, debugging com logs detalhados e tracing via LangSmith. Documente todos os comandos usados, proteja suas chaves de API com .env e compartilhe o repositório via SSH no GitHub. Experimente rodar o projeto em diferentes versões do Python usando pyenv.

## Capítulo 9: LangChain na Produção: Escalabilidade, Observabilidade e Segurança

**Neste capítulo, você vai aprender:**

* Como preparar, escalar, monitorar e proteger aplicações LangChain em ambientes de produção.
* Exercício Hello World contextualizado para produção, com logging e tracing.
* Estratégias de escalabilidade: rate limiting, cache, paralelismo, filas.
* Observabilidade: logs, tracing com LangSmith, métricas e alertas.
* Segurança: proteção de chaves, validação de entradas/saídas, prevenção de injeção de prompt.
* Otimização de custo e desempenho.
* Deploy, CI/CD, manutenção e atualização contínua.
* Estudo de caso prático e exercícios.
* Tabelas, fluxogramas e teste de conhecimento.

### 1. Introdução: Desafios de Produção com LangChain

Levar uma aplicação de IA para produção é como transformar um protótipo em uma máquina industrial: exige robustez, monitoramento, segurança e capacidade de escalar. O LangChain oferece recursos para cada etapa desse processo, mas é preciso conhecer as melhores práticas para evitar surpresas.

### Hands-on: Exercício 1 — Hello World em Produção com Tracing

* **Objetivo:** Demonstrar logging, tracing e boas práticas de produção.
* **Nome do Arquivo:** capitulo\_09/hello\_world\_prod.py
* **Dependências:** langchain, langchain-google-genai, python-dotenv, langsmith
* **Comando de Instalação:**

uv add langchain langchain-google-genai python-dotenv langsmith

**Comando de Execução (Linux/macOS):**

python capitulo\_09/hello\_world\_prod.py

**Comando de Execução (Windows):**

python capitulo\_09\hello\_world\_prod.py

**Saída Esperada (pode variar):**

Uma frase sobre produção de IA, com trace registrado no painel do LangSmith.

**Troubleshooting Comum:** \* AuthenticationError: Verifique se a chave de API do LangSmith está correta. \* Trace não aparece: Certifique-se de que as variáveis de ambiente estão setadas. \* ModuleNotFoundError: Instale todas as dependências com uv add.

# capitulo\_09/hello\_world\_prod.py  
import os  
from dotenv import load\_dotenv  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_core.prompts import ChatPromptTemplate  
from langsmith import traceable, wrappers  
  
load\_dotenv()  
os.environ["LANGSMITH\_TRACING"] = "true"  
os.environ["LANGSMITH\_API\_KEY"] = os.getenv("LANGSMITH\_API\_KEY") or input("LangSmith API Key: ")  
  
@traceable  
def hello\_chain(topico):  
 prompt = ChatPromptTemplate.from\_template("Escreva uma frase sobre {topico}")  
 llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
 chain = prompt | llm  
 return chain.invoke({"topico": topico}).content  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print(hello\_chain("produção de IA"))

### 2. Escalabilidade: Rate Limiting, Cache, Paralelismo e Filas

**Tabela: Estratégias de Escalabilidade** | Estratégia | Benefício | Trade-off | |——————–|———————|———————| | Rate Limiting | Evita bloqueios | Latência aumentada | | Cache de respostas | Menos custo | Pode perder contexto| | Paralelismo | Mais rápido | Complexidade | | Filas de mensagens | Controle de fluxo | Latência variável |

**Exemplo de Rate Limiting com Retentativa:**

# capitulo\_09/rate\_limit\_retry.py  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
import time  
  
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
  
def call\_llm\_with\_retry(prompt, max\_retries=5, initial\_delay=1):  
 for i in range(max\_retries):  
 try:  
 return llm.invoke(prompt)  
 except Exception as e:  
 delay = initial\_delay \* (2 \*\* i)  
 print(f"Erro: {e}. Retentando em {delay}s...")  
 time.sleep(delay)  
 raise Exception("Falha após múltiplas tentativas.")

**Como executar:**

python capitulo\_09/rate\_limit\_retry.py

### 3. Observabilidade: Logs, Tracing e Métricas

Monitorar uma aplicação em produção é essencial para detectar problemas e otimizar desempenho. O LangSmith permite tracing detalhado de chains, agentes e ferramentas.

**Como habilitar tracing com LangSmith:**

import os, getpass  
os.environ["LANGSMITH\_TRACING"] = "true"  
os.environ["LANGSMITH\_API\_KEY"] = getpass.getpass("LangSmith API Key: ")

**Fluxograma textual do ciclo de observabilidade:** 1. Ativar logs/tracing 2. Executar chain/agent 3. Inspecionar traces e métricas 4. Corrigir e otimizar

**Exemplo de tracing em pipeline RAG:**

# capitulo\_09/rag\_tracing.py  
from openai import OpenAI  
from langsmith import traceable, wrappers  
  
openai\_client = wrappers.wrap\_openai(OpenAI())  
  
@traceable  
def retriever(query):  
 return ["Documento relevante"]  
  
@traceable  
def rag(question):  
 docs = retriever(question)  
 system\_message = f"Responda usando apenas: {docs}"  
 return openai\_client.chat.completions.create(  
 messages=[{"role": "system", "content": system\_message}, {"role": "user", "content": question}],  
 model="gpt-4o-mini"  
 )  
  
print(rag("Onde Harrison trabalhou?"))

### 4. Segurança: Proteção de Chaves, Validação e Injeção de Prompt

**Checklist de Segurança:** - Armazene chaves em variáveis de ambiente ou arquivos .env (nunca no código). - Use getpass para entrada segura de chaves. - Valide entradas e saídas para evitar ataques de injeção de prompt. - Utilize guardrails e ferramentas como Layerup Security para bloquear PII e alucinações.

**Exemplo de proteção de chave:**

import os  
from getpass import getpass  
os.environ["OPENAI\_API\_KEY"] = getpass("OpenAI API Key: ")

**Exemplo de bloqueio de PII:**

# capitulo\_09/pii\_block.py  
from langchain\_community.llms.layerup\_security import LayerupSecurity  
from langchain\_openai import OpenAI  
  
openai = OpenAI(model\_name="gpt-3.5-turbo", openai\_api\_key=os.environ["OPENAI\_API\_KEY"])  
layerup\_security = LayerupSecurity(llm=openai, layerup\_api\_key="LAYERUP\_API\_KEY", response\_guardrails=["layerup.hallucination"])  
response = layerup\_security.invoke("Meu nome é Bob Dylan. Meu SSN é 123-45-6789.")  
print(response)

### 5. Otimização de Custo e Desempenho

**Principais estratégias:** - Cache de respostas - Batch de queries - Quantização de modelos (ex: WeightOnlyQuantConfig) - Monitoramento de custos

**Tabela: Otimização de Custo/Desempenho** | Estratégia | Benefício | Trade-off | |——————–|———————|———————| | Cache | Menos custo | Pode perder contexto| | Batch | Mais rápido | Respostas menos personalizadas| | Quantização | Menos memória | Possível perda de precisão|

### 6. Deploy, CI/CD e Manutenção

**Exemplo de Dockerfile para produção:**

# capitulo\_09/Dockerfile  
FROM python:3.12-slim  
WORKDIR /app  
COPY . .  
RUN pip install --no-cache-dir -r requirements.txt  
CMD ["python", "hello\_world\_prod.py"]

**Como executar:**

docker build -t langchain-prod .  
docker run --env-file .env langchain-prod

**Checklist de CI/CD:** - Testes automatizados (pytest) - Deploy contínuo (GitHub Actions, GitLab CI) - Rollback automático em caso de falha

### 7. Estudo de Caso: Chatbot em Produção

**Cenário:** Um chatbot de atendimento que precisa escalar, ser seguro e monitorado.

**Passos:** 1. Implementar chain com tracing e validação 2. Proteger chaves e bloquear PII 3. Monitorar latência e custo 4. Deploy com Docker e CI/CD 5. Corrigir bugs e otimizar

**Exemplo de código:**

# capitulo\_09/chatbot\_prod.py  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_core.prompts import ChatPromptTemplate  
from langsmith import traceable  
  
@traceable  
def atendimento\_chain(pergunta):  
 prompt = ChatPromptTemplate.from\_template("Responda como atendente: {pergunta}")  
 llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash")  
 chain = prompt | llm  
 return chain.invoke({"pergunta": pergunta}).content  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 print(atendimento\_chain("Quais são os horários de funcionamento?"))

### 8. Exercícios Práticos

1. Implemente um pipeline RAG com tracing e validação de entrada.
2. Crie um Dockerfile para sua aplicação LangChain.
3. Simule um ataque de injeção de prompt e bloqueie usando guardrails.
4. Compare latência e custo antes/depois de aplicar cache.

### 9. Tabelas, Fluxogramas e Recursos Visuais

**Tabela comparativa de estratégias de produção:** | Estratégia | Função | Quando usar? | |——————–|—————|———————-| | Tracing (LangSmith)| Observabilidade| Debugging, auditoria | | Guardrails | Segurança | Bloquear PII/alucinação| | Docker/CI/CD | Deploy | Produção automatizada|

**Fluxograma do ciclo de produção:** 1. Escrever código 2. Escrever testes 3. Ativar logs/tracing 4. Proteger chaves 5. Deploy (Docker/CI/CD) 6. Monitorar e otimizar 7. Manter e atualizar

### 10. Teste seu Conhecimento

1. Como ativar tracing com LangSmith?
   1. Instalar pytest
   2. Setar LANGSMITH\_TRACING e LANGSMITH\_API\_KEY
   3. Ativar cache de respostas
   4. Usar ConversationBufferMemory
2. Qual estratégia protege chaves de API em produção?
   1. Armazenar em .env/getpass
   2. Escrever no código fonte
   3. Compartilhar em repositório público
   4. Ignorar variáveis de ambiente
3. O que é rate limiting?
   1. Limitar requisições para evitar bloqueios
   2. Aumentar temperatura do modelo
   3. Usar cache de respostas
   4. Executar queries em batch
4. Como evitar ataques de injeção de prompt?
   1. Validar entradas/saídas e usar guardrails
   2. Ignorar logs
   3. Compartilhar prompts publicamente
   4. Usar apenas LLMs locais
5. Qual o benefício de usar Docker em produção?
   1. Isolamento e automação de deploy
   2. Menos segurança
   3. Menos escalabilidade
   4. Ignorar dependências

**Respostas:** 1. b 2. a 3. a 4. a 5. a

### Projeto Hands-on: Chatbot Seguro e Escalável em Produção

Coloque em prática os conceitos do capítulo criando um projeto Python com LangChain para produção: implemente um chatbot com tracing, proteção de chaves, validação de entradas, bloqueio de PII, deploy com Docker e CI/CD. Documente todos os comandos usados, proteja suas chaves de API com .env e compartilhe o repositório via SSH no GitHub. Experimente rodar o projeto em diferentes versões do Python usando pyenv.

#### Dicas avançadas para produção:

* **Feedback com LangSmith:** Envie feedback após interações reais para monitorar qualidade e ajustar o modelo:
* from langsmith import Client  
  client = Client()  
  run\_id = "<id\_do\_run>"  
  client.create\_feedback(run\_id, key="user\_feedback", score=1, comment="Chatbot respondeu corretamente em produção.")
* **Validação de entradas:** Use pydantic ou TypedDict para garantir integridade dos dados:
* from pydantic import BaseModel  
  class Pergunta(BaseModel):  
   pergunta: str
* **Monitoramento:** Integre métricas customizadas com Prometheus/Grafana para latência, custo e uso de recursos.
* **Cache e paralelismo:** Use cache nativo do LangChain e métodos assíncronos (ainvoke, abatch) para escalar e economizar:
* from langchain.cache import InMemoryCache  
  import langchain  
  langchain.llm\_cache = InMemoryCache()
* **Segurança:** Teste e monitore ataques de prompt injection e vazamento de dados sensíveis. Simule ataques e monitore logs/traces.
* **Documentação:** Mantenha README atualizado com exemplos, comandos, troubleshooting e instruções de feedback/tracing.

## Capítulo 10: Projeto Final Integrador — Assistente de Viagem Inteligente

**Neste capítulo, você vai aprender:**

* Como integrar múltiplos conceitos do LangChain em um projeto robusto.
* Construir um agente de viagem inteligente que pesquisa destinos, clima, atrações e gerencia orçamento.
* Utilizar ferramentas externas (Tavily) e customizadas no LangChain.
* Orquestrar interações com LLM, ferramentas e gerenciamento de segredos.
* Documentar comandos, troubleshooting, pontos chave e testar conhecimento.

### Hands-on: Exercício — Assistente de Viagem Inteligente

* **Objetivo:** Desenvolver um agente completo que auxilia no planejamento de viagens, combinando pesquisa e gerenciamento de orçamento.
* **Nome do Arquivo:** exercicios/capitulo\_09/exercicio\_1/main.py
* **Dependências:** langchain, langchain-google-genai, langchain-tavily, python-dotenv
* **Comando de Instalação:**

uv add langchain langchain-google-genai langchain-tavily python-dotenv

* **Configuração Adicional:** Adicione sua chave de API da Tavily ao arquivo .env como TAVILY\_API\_KEY.

# capitulo\_09/assistente\_viagem.py  
  
import os  
from dotenv import load\_dotenv  
from langchain\_google\_genai import ChatGoogleGenerativeAI  
from langchain\_tavily import TavilySearchResults  
from langchain.agents import AgentExecutor, create\_tool\_calling\_agent  
from langchain\_core.prompts import ChatPromptTemplate  
from langchain.tools import tool  
  
# Carregar variáveis de ambiente  
load\_dotenv()  
  
# --- Ferramentas ---  
  
# Ferramenta de busca na internet  
search\_tool = TavilySearchResults(  
 max\_results=3,  
 description="Uma ferramenta de busca para encontrar informações na internet sobre destinos de viagem, clima, atrações, etc."  
)  
  
# Ferramenta customizada para gerenciar orçamento de viagem (simulado em memória)  
class OrcamentoViagem:  
 def \_\_init\_\_(self):  
 self.orcamento = {}  
  
 @tool  
 def adicionar\_item\_orcamento(self, item: str, custo: float) -> str:  
 """  
 Adiciona um item e seu custo ao orçamento de viagem.  
 Útil para planejar gastos.  
 Exemplo: adicionar\_item\_orcamento("Passagem Aérea", 1500.00)  
 """  
 self.orcamento[item] = self.orcamento.get(item, 0.0) + custo  
 return f"Item '{item}' com custo de R${custo:.2f} adicionado ao orçamento. Orçamento atual: {self.orcamento}"  
  
 @tool  
 def ver\_orcamento\_total(self) -> str:  
 """  
 Retorna o total do orçamento de viagem e a lista de itens.  
 """  
 total = sum(self.orcamento.values())  
 if not self.orcamento:  
 return "O orçamento de viagem está vazio."  
 return f"Orçamento total: R${total:.2f}. Itens: {self.orcamento}"  
  
orcamento\_viagem = OrcamentoViagem()  
  
tools = [search\_tool, orcamento\_viagem.adicionar\_item\_orcamento, orcamento\_viagem.ver\_orcamento\_total]  
  
# --- LLM ---  
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash", temperature=0)  
  
# --- Prompt do Agente ---  
prompt = ChatPromptTemplate.from\_messages([  
 ("system", "Você é um assistente de planejamento de viagens. Use as ferramentas disponíveis para ajudar o usuário a planejar sua viagem, pesquisar informações e gerenciar o orçamento. Seja prestativo e informativo."),  
 ("human", "{input}"),  
 ("placeholder", "{agent\_scratchpad}"),  
])  
  
# --- Agente e Executor ---  
agent = create\_tool\_calling\_agent(llm, tools, prompt)  
agent\_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)  
  
# --- Execução ---  
if \_\_name\_\_ == "\_\_main\_\_":  
 print("Assistente de Viagem pronto! Como posso ajudar a planejar sua próxima aventura?")  
  
 perguntas = [  
 "Qual o clima em Paris em agosto?",  
 "Quais são as principais atrações turísticas em Roma?",  
 "Adicione 'Passagem Aérea' com custo de 2500.00 ao orçamento.",  
 "Adicione 'Hospedagem' com custo de 1200.00 ao orçamento.",  
 "Qual o meu orçamento total até agora?",  
 "Preciso de ideias para uma viagem de aventura na América do Sul.",  
 "Qual o custo de um jantar em um restaurante médio em Tóquio?",  
 "Qual o meu orçamento total?",  
 ]  
  
 for pergunta in perguntas:  
 print(f"\n> Pergunta: {pergunta}")  
 response = agent\_executor.invoke({"input": pergunta})  
 print(f"\n< Resposta Final: {response['output']}")

**Comando de Execução (Linux/macOS):**

chmod +x capitulo\_09/execucao\_exercicio\_9\_1.sh  
./capitulo\_09/execucao\_exercicio\_9\_1.sh

**Comando de Execução (Windows):**

REM Execute o exercício no Windows  
capitulo\_09\execucao\_exercicio\_9\_1.bat

**Saída Esperada (pode variar):**

Assistente de Viagem pronto! Como posso ajudar a planejar sua próxima aventura?  
> Pergunta: Qual o clima em Paris em agosto?  
< Resposta Final: ...  
> Pergunta: Quais são as principais atrações turísticas em Roma?  
< Resposta Final: ...  
> Pergunta: Adicione 'Passagem Aérea' com custo de 2500.00 ao orçamento.  
< Resposta Final: ...  
> Pergunta: Adicione 'Hospedagem' com custo de 1200.00 ao orçamento.  
< Resposta Final: ...  
> Pergunta: Qual o meu orçamento total até agora?  
< Resposta Final: ...  
> Pergunta: Preciso de ideias para uma viagem de aventura na América do Sul.  
< Resposta Final: ...  
> Pergunta: Qual o custo de um jantar em um restaurante médio em Tóquio?  
< Resposta Final: ...  
> Pergunta: Qual o meu orçamento total?  
< Resposta Final: ...

**Troubleshooting Comum:** \* AuthenticationError ou TAVILY\_API\_KEY não configurada: Verifique se a chave está correta no .env. \* GOOGLE\_API\_KEY não configurada: Verifique se está no .env. \* ModuleNotFoundError: Instale todas as dependências com uv add. \* Erros de conexão: Verifique sua internet e limites de API. \* Agente não usando a ferramenta: Revise as descrições das ferramentas. \* Saída inesperada: Ative verbose=True para inspecionar o raciocínio do agente.

### Pontos Chave

* Projetos integradores consolidam múltiplos conceitos do LangChain.
* Agentes com múltiplas ferramentas resolvem tarefas complexas e reais.
* LCEL facilita a composição de pipelines robustos.
* Segurança no gerenciamento de chaves de API é fundamental.

### Resumo do Capítulo

Neste capítulo, você aplicou todos os conceitos aprendidos para construir um Assistente de Viagem Inteligente. O projeto final demonstrou como orquestrar agentes, integrar ferramentas externas e customizadas, aplicar LCEL e gerenciar segredos de forma segura.

### Teste seu Conhecimento

1. Qual o principal benefício de usar múltiplas ferramentas em um agente?
   1. Reduz o custo das chamadas de API.
   2. Permite que o agente execute tarefas mais complexas e diversificadas.
   3. Aumenta a velocidade de resposta do LLM.
   4. Elimina a necessidade de um LLM.
2. No Assistente de Viagem, qual ferramenta seria usada para descobrir “Qual o clima em Paris em agosto?”
   1. adicionar\_item\_orcamento
   2. ver\_orcamento\_total
   3. search\_tool
   4. Nenhuma das anteriores.
3. Por que é importante usar load\_dotenv() e um arquivo .env para as chaves de API?
   1. Para tornar o código mais curto.
   2. Para melhorar a performance do agente.
   3. Para evitar que as chaves de API sejam expostas no código-fonte e versionadas.
   4. Para que o agente possa se conectar à internet.
4. Se o Assistente de Viagem não estivesse usando a ferramenta adicionar\_item\_orcamento quando solicitado, qual seria a primeira coisa a verificar?
   1. A versão do Python.
   2. A descrição da ferramenta adicionar\_item\_orcamento.
   3. A conexão com a internet.
   4. O modelo do LLM.
5. Qual o conceito central do LangChain que permite ao agente decidir dinamicamente qual ferramenta usar?
   1. Prompt Templates.
   2. Output Parsers.
   3. O loop de raciocínio (ReAct) do agente.
   4. Streaming de respostas.

**Respostas:** 1. b 2. c 3. c 4. b 5. c

### Projeto Hands-on: Assistente de Viagem Inteligente

Coloque em prática os conceitos do capítulo criando um projeto Python com LangChain para ser um assistente de viagem inteligente. Implemente pesquisa de destinos, clima, atrações, gerenciamento de orçamento e integração de múltiplas ferramentas. Documente todos os comandos usados, proteja suas chaves de API com .env e compartilhe o repositório via SSH no GitHub. Experimente rodar o projeto em diferentes versões do Python usando pyenv.

## Conclusão

Ao longo deste e-book, você percorreu uma jornada completa pelo universo do LangChain, desde a configuração do ambiente até a construção de agentes inteligentes prontos para produção. Explorou conceitos fundamentais de IA, engenharia de contexto, ferramentas, agentes, memória, feedback, testes, debugging, otimização, segurança e integração de múltiplas tecnologias.

Você aprendeu a: \* Montar um ambiente Python robusto e seguro, com Linux/WSL, pyenv, zsh, Oh My Zsh e gerenciamento moderno de dependências. \* Proteger segredos, chaves de API e garantir boas práticas de segurança. \* Construir chains, agentes autônomos, multiagentes e integrar ferramentas externas e customizadas. \* Implementar memória, feedback e aprendizado contínuo para agentes evolutivos. \* Testar, debugar, monitorar e otimizar aplicações LangChain com LangSmith, tracing, logs e CI/CD. \* Orquestrar projetos práticos, como chatbots, assistentes de viagem e sistemas corporativos, aplicando LCEL e boas práticas de produção.

Mais do que aprender técnicas, você desenvolveu a capacidade de criar soluções reais, seguras, escaláveis e inovadoras com IA generativa. O domínio do LangChain abre portas para aplicações que vão desde protótipos até sistemas robustos em produção, com observabilidade, segurança e integração contínua.

Ao aplicar os conhecimentos adquiridos, você está apto a enfrentar desafios do mundo real, seja desenvolvendo assistentes inteligentes para empresas, automatizando processos, ou criando experiências personalizadas para usuários. O LangChain permite que você aproveite o poder dos LLMs para construir aplicações flexíveis, adaptáveis e preparadas para evoluir conforme novas demandas surgem.

Além disso, você aprendeu a importância da colaboração e do compartilhamento de conhecimento. A comunidade LangChain está em constante crescimento, e contribuir com projetos open source, tutoriais e exemplos práticos é uma excelente forma de se manter atualizado e ajudar outros desenvolvedores a avançar na área de IA generativa.

### Visão de Mercado e Oportunidades

O mercado de IA generativa está em rápida expansão, com demanda crescente por profissionais capazes de criar soluções inovadoras e seguras. Empresas de diversos setores buscam especialistas em LangChain para desenvolver assistentes inteligentes, sistemas de automação, análise de dados avançada e integração de LLMs em processos críticos. Dominar LangChain posiciona você como protagonista em projetos de alto impacto, seja em startups, grandes corporações ou iniciativas acadêmicas.

A capacidade de unir conhecimento técnico, criatividade e visão de negócio é um diferencial competitivo. Projetos bem estruturados com LangChain podem transformar a experiência do usuário, otimizar operações e abrir novas fontes de receita. O ecossistema está em constante evolução, com novas ferramentas, APIs e integrações surgindo a cada mês. Manter-se atualizado e contribuir para a comunidade são atitudes essenciais para o crescimento profissional.

### Protagonismo e Colaboração

O protagonismo do leitor é fundamental para o avanço da área. Compartilhe seus aprendizados, publique tutoriais, participe de fóruns e eventos, e colabore com outros desenvolvedores. O trabalho em equipe, a troca de experiências e o engajamento em projetos open source aceleram o desenvolvimento de soluções cada vez mais sofisticadas e seguras.

A colaboração internacional é uma marca do universo LangChain. Aproveite oportunidades de networking, participe de hackathons, contribua com documentação e exemplos práticos. O futuro da IA generativa depende do envolvimento ativo de profissionais comprometidos com a ética, a inovação e o impacto positivo na sociedade.

### Próximos Passos

* Continue explorando novas ferramentas, APIs e integrações.
* Compartilhe seus projetos, contribua com a comunidade e mantenha-se atualizado.
* Experimente diferentes LLMs, arquiteturas e desafios práticos.
* Aplique os conceitos em problemas reais do seu dia a dia ou empresa.
* Participe de fóruns, eventos e grupos de discussão para expandir sua rede de contatos e aprender com outros profissionais.
* Mantenha-se atento às novidades do ecossistema LangChain e IA generativa, acompanhando lançamentos, atualizações e tendências do mercado.
* Busque certificações, cursos avançados e especializações para aprofundar seu conhecimento.
* Desenvolva projetos autorais e compartilhe resultados em plataformas como GitHub, Medium e LinkedIn.

#### O próximo passo: Design de agentes e workflows com LangChain

À medida que você avança, é fundamental entender os padrões de design para agentes e workflows em aplicações com LLMs. O artigo da Anthropic sobre agentes eficazes destaca que nem sempre a solução mais sofisticada é a melhor: muitas vezes, workflows simples e bem definidos são suficientes para a maioria dos casos. Frameworks como LangChain e LangGraph facilitam a implementação de sistemas agentic, mas é essencial saber quando utilizar agentes autônomos e quando optar por workflows orquestrados.

**Workflows** são ideais para tarefas previsíveis e bem estruturadas, onde o caminho de execução pode ser definido previamente. Já **agentes** são recomendados para problemas abertos, que exigem flexibilidade, tomada de decisão dinâmica e adaptação ao contexto. A orquestração baseada em grafos, como a proposta pelo LangGraph, permite criar sistemas híbridos, combinando workflows e agentes para maximizar eficiência e controle.

O próximo passo para quem domina LangChain é aprofundar-se nesses padrões, estudando casos de uso, experimentando diferentes arquiteturas e avaliando quando cada abordagem é mais adequada. Recomenda-se começar simples, medir resultados e só adicionar complexidade quando necessário. Transparência, simplicidade e documentação clara das ferramentas são princípios-chave para construir agentes confiáveis e escaláveis.

Para saber mais, consulte o artigo “Building Effective Agents” da Anthropic e explore frameworks como LangChain, LangGraph e Model Context Protocol para criar soluções cada vez mais avançadas e alinhadas às melhores práticas do mercado.

## Agradecimento

Parabéns por chegar até aqui! O futuro da IA está em suas mãos. Que este material seja um guia prático e inspirador para suas próximas conquistas com LangChain e IA generativa.

Lembre-se: o aprendizado é contínuo, e cada projeto é uma oportunidade de inovar, aprimorar suas habilidades e impactar positivamente o mundo ao seu redor. Continue explorando, experimentando e compartilhando suas descobertas.

**Bons estudos e ótimos projetos!**

## **Próximos Passos na sua Jornada**

Parabéns! Você chegou ao final deste livro e, mais importante, deu um passo gigantesco na sua jornada com a Inteligência Artificial e o LangChain. Mas lembre-se, o aprendizado é contínuo. Aqui estão algumas sugestões para seus próximos passos:

1. **Explore o LangGraph:** Para sistemas multiagentes ainda mais complexos e com estados, o LangGraph é o próximo nível.
2. **Aprofunde-se em RAG:** Experimente diferentes tipos de Vector Stores, Embeddings e estratégias de recuperação.
3. **Construa seus Próprios Projetos:** A melhor forma de aprender é fazendo. Pense em problemas reais que você enfrenta e tente resolvê-los com agentes de IA.
4. **Contribua para a Comunidade:** Compartilhe seu conhecimento, ajude outros desenvolvedores e contribua para projetos de código aberto.
5. **Mantenha-se Atualizado:** O campo da IA avança rapidamente. Siga blogs, participe de conferências e continue experimentando.

## **Comunidade e Contato**

A jornada do aprendizado não termina aqui. Na verdade, ela está apenas começando.

* Código Fonte: Todo o código deste livro está disponível para você clonar, modificar e experimentar. Acesse o repositório em:  
  https://github.com/igormedeiros/livros/blob/main/langchain-na-pratica/
* Comunidade no Telegram: Junte-se a outros desenvolvedores, tire dúvidas, compartilhe seus projetos e continue a conversa em nossa comunidade:  
  https://t.me/igormedeiros\_comunidade
* Feedback e Contato: Sua opinião é incrivelmente valiosa. Se você gostou deste livro, por favor, considere deixar uma avaliação na Amazon. Isso ajuda outros leitores como você a encontrar este material e me dá o feedback necessário para continuar melhorando. Para outras dúvidas, sugestões ou para saber mais sobre meu trabalho, visite meu site:  
  https://igormedeiros.com.br

Obrigado por me acompanhar nesta jornada. Agora, vá e construa algo incrível!

## **Sobre o Autor**

A jornada de Igor Medeiros é movida por um propósito forjado no amor, na dor e na superação. Filho de um herói que, em suas palavras, “samba todo dia na cara do Alzheimer”, e pai da Melissa, “a luz de sua jornada”, ele carrega em sua história a força desses dois pilares. Sua trajetória profissional de mais de duas décadas foi profundamente ressignificada após um grave desafio de saúde em 2012, que incluiu um tumor gigante na cabeça e uma embolia pulmonar no pós-operatório. **A batalha pela vida deixou marcas permanentes: a paralisia facial e a surdez unilateral total, ambas do lado direito do rosto.**

A experiência de quase morte e a reabilitação que se seguiu não foram uma pausa, mas o ponto de ignição para uma nova missão. Nesse período, sua jornada incluiu um corajoso pivô de carreira, no qual atuou como executivo de marketing e vendas. Essa imersão no lado do negócio ensinou-lhe lições valiosas sobre a importância da qualidade da entrega final e a perspectiva do cliente, visão que hoje integra a todos os seus projetos tecnológicos: usar a tecnologia para honrar a vida e gerar um impacto que realmente importe.

Como Engenheiro de Software Sênior e especialista em Java para smart cards, sua carreira o levou a palestrar por todo o Brasil e até no Japão. Ao longo de mais de 20 anos, liderou a criação de soluções robustas para diversos setores da indústria. Essa base sólida, que vai desde a homologação de sistemas criptográficos para a Casa Civil até a liderança técnica em projetos de inovação, construiu o alicerce para sua atuação atual.

Hoje, como especialista de IA Agents e evangelista de Inteligência Artificial, Igor está na vanguarda da tecnologia, liderando questões técnicas de projetos e definindo padrões para gigantes atuais dos setores financeiro e de telecomunicações, como Santander, Bradesco, Pernambucanas, TIM e o grupo Telefônica. É nesse cenário de ponta que ele desenvolve e coloca em produção sistemas complexos com agentes de IA. Ele direciona essa expertise para sua grande paixão: o setor da saúde, buscando ativamente criar soluções em neurologia que possam, um dia, reescrever histórias como a sua e de seu pai.

Além de seu trabalho corporativo, Igor é um líder de comunidade nato, que acredita que o conhecimento só tem valor quando é compartilhado. Essa vocação para o humanismo tem raízes profundas: por 14 anos, dedicou-se como voluntário em uma ONG com a missão de resgatar pessoas em situação de rua. Em um marcante contraste com seu trabalho de ponta em Inteligência Artificial, essa experiência solidificou sua crença de que a tecnologia mais avançada deve sempre servir à empatia e à conexão humana. Seja em palestras, artigos ou como mentor, ele se dedica a capacitar outros desenvolvedores, fechando o ciclo de uma jornada extraordinária que transforma gratidão em legado. Este livro é mais um passo nesse caminho.

## **Glossário**

* **Agente (Agent):** Sistema que usa um LLM como “cérebro” para decidir ações, utilizando ferramentas e raciocínio dinâmico.
* **Agent Executor:** Componente que orquestra o ciclo ReAct de um agente, invocando ferramentas e processando observações.
* **Chain:** Sequência predefinida de operações ou chamadas a modelos de linguagem para formar um pipeline inteligente.
* **Engenharia de Contexto:** Disciplina de gerenciar e moldar dinamicamente o contexto recebido por agentes para otimizar raciocínio e ações.
* **Engenharia de Prompts:** Arte de projetar e otimizar instruções para LLMs, visando obter respostas precisas e relevantes.
* **Ferramenta (Tool):** Função ou ação que um agente pode executar, geralmente uma função Python com descrição clara, permitindo interação com sistemas externos.
* **LangChain Expression Language (LCEL):** Linguagem declarativa para compor Runnables no LangChain, permitindo criação de pipelines flexíveis, paralelos e eficientes usando o operador pipe (|).
* **LLM (Large Language Model):** Modelo de linguagem de grande escala, como Gemini, GPT-4, Claude, capaz de entender e gerar texto.
* **Memória:** Mecanismo que permite agentes reterem informações, histórico de conversas ou fatos, como ConversationBufferMemory, GenerativeAgentMemory, VectorStore e Knowledge Triples.
* **Multiagente:** Sistema composto por múltiplos agentes especializados que colaboram para resolver problemas complexos, orquestrados por fluxos como LangGraph.
* **Pipeline:** Fluxo de processamento de dados, composto por etapas encadeadas (chains, agentes, ferramentas) para realizar tarefas complexas.
* **Prompt:** Instrução ou entrada de texto fornecida a um LLM para gerar uma resposta.
* **RAG (Retrieval-Augmented Generation):** Padrão arquitetural que permite aos LLMs acessar e utilizar informações externas e atualizadas antes de gerar uma resposta.
* **ReAct (Thought + Action + Observation):** Padrão de raciocínio para agentes, onde o LLM pensa, executa uma ação e observa o resultado em ciclo iterativo.
* **Runnable:** Interface padronizada no LangChain que permite encadear componentes usando o operador pipe (|), incluindo prompts, modelos, parsers e funções Python.
* **uv:** Gerenciador de pacotes e ambientes virtuais escrito em Rust, conhecido por alta performance e compatibilidade com pyproject.toml.
* **VectorStore:** Estrutura para armazenar e buscar memórias semânticas usando embeddings, utilizada em RAG e recuperação de contexto.
* **WSL (Windows Subsystem for Linux):** Camada de compatibilidade que permite executar ambiente Linux diretamente no Windows.
* **LangSmith:** Plataforma para tracing, logging e observabilidade de pipelines LangChain em produção.
* **LangGraph:** Biblioteca para orquestração de sistemas multiagentes via grafos, permitindo fluxos complexos e colaborativos entre agentes.

## **Referências Bibliográficas**

### **Documentação Oficial e Ferramentas**

* **Documentação Oficial do LangChain:** <https://python.langchain.com/>
* **Documentação do LangGraph:** Framework para orquestração de agentes e workflows baseados em grafos. <https://langchain-ai.github.io/langgraph/>
* **Documentação do Model Context Protocol (MCP):** Integração de ferramentas e agentes. <https://modelcontextprotocol.io/>
* **Documentação do Google AI Studio:** <https://aistudio.google.com/>
* **Documentação da Tavily API:** <https://tavily.com/>
* **Documentação do Claude (Anthropic):** <https://www.anthropic.com/claude>

### **Livros Recomendados**

* **“Generative AI with LangChain” de Ben Stokes**
* **“Building LLM Powered Applications” de Josh Star**

### **Artigos e Pesquisas Fundamentais**

* **“Attention Is All You Need” (Vaswani et al., 2017):** Arquitetura Transformer.
* **“Language Models are Few-Shot Learners” (Brown et al., 2020):** Few-shot learning.
* **“Chain-of-Thought Prompting Elicits Reasoning in Large Language Models” (Wei et al., 2022):** Chain-of-Thought.
* **“Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks” (Lewis et al., 2020):** Padrão RAG.
* **“ReAct: Synergizing Reasoning and Acting in Language Models” (Yao et al., 2022):** Padrão ReAct. <https://react-lm.github.io/>
* **“Building Effective Agents” (Schluntz & Zhang, Anthropic, 2024):** Design patterns, agentes vs workflows, orquestração baseada em grafos. <https://www.anthropic.com/engineering/building-effective-agents>