



RAG

Retrieval-Augmented *Generation*

Para Leigos

Sandeco Macedo

*Tudo o que você
precisa saber para
ampliar o conhecimento
da sua LLM*

Os meus alunos do Instituto Federal de Goiás, amo vocês!

Copyright © 2025

Prefácio

É uma honra escrever estas primeiras palavras para o mais recente trabalho do meu amigo Sandeco. Nossos bate-papos sempre transbordam ideias e possibilidades, com aquela energia contagiante que só verdadeiros entusiastas de tecnologia conseguem gerar.

Sandeco se estabeleceu como um dos maiores evangelistas de agentes inteligentes e CrewAI do Brasil. Sua didática excepcional não vem do acaso – é fruto de décadas dedicadas à pesquisa, ensino e aplicação prática de inteligência artificial. O que mais admiro em seu trabalho é a capacidade de transformar conceitos complexos em conhecimento acessível, sem jamais sacrificar o rigor técnico.

Tenho o privilégio de participar da incrível e vibrante comunidade que ele construiu, um ecossistema de makers e entusiastas que não se contentam apenas com teoria – eles implementam, testam, quebram e reconstruem. Um testemunho do impacto transformador que o conhecimento bem compartilhado pode ter.

Neste livro, Sandeco aborda dois protocolos que prometem revolucionar como as IAs se comunicam e colaboram: o Model Context Protocol (MCP) e o Agent-to-Agent Protocol (A2A). Não é exagero dizer que estamos diante de algo tão fundamental quanto foram as APIs para o desenvolvimento de software. Se as APIs padronizaram como aplicações se comunicam, MCP e A2A padronizam como inteligências artificiais interagem com o mundo e entre si.

Na minha pesquisa sobre Organizações Cognitivas, tenho visto como estes protocolos serão fundamentais para construir redes de agentes verdadeiramente eficazes. O MCP fornece às IAs uma forma padronizada de acessar ferramentas e dados externos – são as "mãos" que permitem que modelos toquem o mundo real. Já o A2A proporciona a "linguagem social" que permite que múltiplos agentes colaborem de forma orquestrada.

Gigantes como Anthropic e Google não apenas desenvolveram estes protocolos, mas já vemos adoção significativa pela Microsoft, Amazon e diversas startups integrando ambos em suas soluções. Ao mergulhar nas páginas que seguem, você está se colocando na vanguarda de uma revolução. Não se trata apenas de aprender uma nova tecnologia, mas de compreender as fundações sobre as quais o futuro da IA colaborativa será construído.

Parabenizo você, leitor, por escolher investir seu tempo em um tema tão fundamental. E parabenizo meu amigo Sandeco por, mais uma vez, iluminar o caminho com clareza e maestria.

Boa leitura!



Kenneth Corrêa

*Autor de 'Organizações Cognitivas' e
Especialista em Tecnologias
Emergentes*

RAG - RETRIEVAL-AUGMENTED GENERATION

Licenciado para - Pedro Henrique de Magalhães Casimiro - 11871259703 - Protegido por Eduzz.com

Meus Livros sobre Inteligência Artificial

The image shows three book covers from Sandeco Macedo's portfolio:

- Agentes Inteligentes vol. 1**: Features a team of white humanoid robots rowing a boat, with one robot at the helm shouting into a megaphone. The cover includes the "crewAI Para iniciantes" logo.
- Python para Inteligência Artificial**: Shows a stylized blue and yellow circuit board forming a question mark shape. The subtitle "Para iniciantes com IA" is at the top.
- Prompts em Ação**: Displays a man working at a computer, with a speech bubble icon above him. The subtitle "Engenharia de Prompts para leigos" is present.

Below the books, a portrait of Sandeco Macedo is shown from the chest up, wearing glasses and a yellow t-shirt, pointing his right index finger upwards. In the bottom right corner of the image area, there is a green circular icon containing a white telephone receiver symbol.

Resultado das minhas pesquisas em IA

Sumário

Prefácio	2
1 Gerando Revoluções	6
1.1 Quando usar RAG	7
1.2 Decodificando a Arquitetura: Os Bastidores do RAG	8
1.3 Um RAG Simples	11
1.4 Ele pode lembrar, RAG com Memória	11
1.5 O RAG Autônomo: Agent RAG	12
1.6 O RAG Corretivo: CRAG	13
1.7 O RAG Adaptativo: Adaptive RAG	14
1.8 O RAG em Grafos: GraphRAG	15
1.9 O RAG Híbrido: Hybrid RAG	16
1.10 O RAG-Fusion: Reciprocal Rank Fusion (RRF)	17
1.11 Hypothetical Document Embedding	18
2 O RAG Clássico	19
2.1 O que é um Corpus de Textos	20
2.2 O Fluxo do RAG	21
2.3 A Fase do Indexador	23
2.4 Lendo e convertendo	24
2.5 Chunking	26
2.6 Criação de Embeddings	28
2.7 Banco de dados Vetorial	30
2.8 Nossa Classe de Encoder	31
2.9 Recuperando conhecimento	35
2.10 Aumento de informação	37
2.11 Gerando a Resposta	39
2.12 Rodando com Streamlit	40
2.13 Exercícios	43
3 Rag com Memória	44
3.1 Criando a Memória com o Redis	46

RAG - RETRIEVAL-AUGMENTED GENERATION

3.2	Redis no Docker	47
3.3	Instalando o Redis	50
3.4	Criando a Mémória	51
3.5	Adicionando a Memória ao RAG	54
3.6	Exercícios	58
4	RAG Autônomo com Agentic RAG	59
4.1	AgenticRAG	60
4.2	Classe de Registro de Datasets	63
4.3	Agente Abstrato	64
4.4	Agente com API da LLM	66
4.5	Agentic RAG com CrewAI	69
4.6	Executando o Agentic RAG	72
4.7	Exercícios	74

CAPÍTULO 1

Gerando Revoluções

Imaginem o seguinte cenário: vocês têm um colega de classe, o Léo, que é simplesmente um gênio. O cara leu todos os livros da biblioteca, assistiu a todas as aulas e tem uma capacidade absurda de conversar sobre qualquer assunto. Ele manja muito de história, física, arte e até daquela matéria de cálculo que todo mundo sofre. A escrita dele é fluida, convincente e ele consegue conectar ideias como ninguém. Só que o Léo tem um pequeno problema: a memória dele, apesar de vasta, às vezes prega peças. Ele pode confundir uma data, atribuir uma citação à pessoa errada ou, na pior das hipóteses, inventar um detalhe que soa verdadeiro só pra manter a conversa fluindo. Esse é o LLM puro: brilhante, mas não totalmente confiável.

Agora, vamos botar esse 'gênio' à prova. Entreguem a ele uma prova final sobre a 'Revolução Francesa', mas com uma pergunta bem específica: 'Qual foi o preço exato do pão em Paris na semana anterior à queda da Bastilha e como isso influenciou os discursos de Camille Desmoulins?'. O Léo, usando apenas sua memória, provavelmente vai desenrolar uma resposta incrível, bem escrita e contextualizada. Ele vai falar sobre a fome, a crise econômica e o papel dos oradores. Mas o preço exato do pão? Ele talvez erre por alguns centavos ou invente um valor que pareça plausível. A essência da resposta estará lá, mas o dado crucial, o fato bruto, pode estar incorreto.

É aqui que a mágica do RAG começa a brilhar. Agora, imaginem que, ao lado do Léo, senta a Ana. A Ana não tem a memória encyclopédica do Léo, mas ela é a rainha da organização e da pesquisa. Ela tem um fichário perfeitamente indexado com resumos de todos os livros da biblioteca. Quando o professor faz a pergunta, antes de o Léo começar a escrever, a Ana entra em ação. Ela não lê o livro inteiro sobre a Revolução Francesa. Ela vai direto na ficha certa, encontra o parágrafo exato sobre a economia pré-revolução e entrega ao Léo um pequeno cartão com a informação: 'Preço do pão: 4 sous. Discursos de Desmoulins mencionaram o "preço absurdo" como estopim para a revolta popular'. A Ana, nesse caso, é o nosso 'Retriever'.

Com esse cartão em mãos, o Léo se transforma. Ele pega aquela informação precisa e a integra em sua genialidade narrativa. A resposta dele agora não é apenas bem escrita e contextualizada, ela é factualmente irrefutável. Ele começa o texto com o dado exato, conecta o preço do pão à inflação da época e tece uma análise brilhante sobre como Desmoulins usou essa informação para inflamar a

RAG - RETRIEVAL-AUGMENTED GENERATION

Retrieval-Augmented Generation



Figura 1.1: Leo e Ana. Gerador e Retriever

população. Ele não apenas 'colou' a informação da Ana; ele a 'aumentou' com seu poder de geração de texto. Essa colaboração perfeita entre o pesquisador focado (Ana) e o gerador eloquente (Léo) é a essência do 'Retrieval-Augmented Generation'.

Pensem nessa dinâmica em escala. A 'biblioteca' não é apenas sobre a Revolução Francesa, mas sim sobre todos os documentos da sua empresa, todos os artigos médicos publicados nos últimos 20 anos, toda a base de conhecimento de um produto ou todos os livros do Sandeco. O fichário da Ana é o nosso banco de vetores, e a habilidade dela de encontrar a ficha certa é o algoritmo de busca. Parâmetros como **chunk_size** definem o tamanho de cada 'resumo' no fichário dela, enquanto **top_k** determina quantos 'cartões' de informação ela entrega ao Léo para cada pergunta. Ajustar esses detalhes é o que transforma uma boa resposta em uma resposta perfeita.

1.1 QUANDO USAR RAG

A decisão de usar RAG em vez de outras técnicas, como o Fine-Tuning, não é apenas uma escolha técnica, mas uma decisão estratégica. Ambas as abordagens buscam especializar um LLM em um domínio de conhecimento, mas o fazem de maneiras fundamentalmente diferentes e com implicações radicalmente distintas em custo, agilidade e manutenção. Entender quando o RAG brilha é o primeiro passo para construir sistemas de IA robustos e eficientes.

O cenário ideal para o RAG é aquele onde a **verdade é volátil**. Pensem em qualquer base de conhecimento que não seja estática. A legislação de um país, por exemplo, está em constante fluxo: novas leis são sancionadas, decretos são publicados e artigos são alterados. Um sistema de IA para advogados que foi treinado ou mesmo 'fine-tuned' em um Vade Mecum de 2023 se tornaria obsoleto e perigosamente impreciso em 2024. O custo para retreinar ou refinar o modelo a cada nova portaria seria proibitivo. O RAG resolve isso com uma elegância impressionante. A 'inteligência' do modelo (sua

RAG - RETRIEVAL-AUGMENTED GENERATION

capacidade de ler e interpretar) é separada do 'conhecimento' (os documentos). Quando uma lei muda, você não toca no cérebro do LLM; você simplesmente atualiza o arquivo de texto correspondente no seu Vector Store. A verdade é atualizada em segundos, a um custo marginal.

Isso nos leva ao segundo ponto crucial: **custo e complexidade**. O Fine-Tuning, por mais poderoso que seja para alterar o 'comportamento' ou o 'estilo' de um modelo, é um processo computacionalmente intensivo. Ele exige a preparação de datasets massivos de exemplos, um poder de processamento considerável (geralmente múltiplas GPUs de ponta rodando por horas ou dias) e um conhecimento técnico aprofundado para ajustar os hiperparâmetros e evitar problemas como o 'catastrophic forgetting'. É como reformar a fundação de um prédio. O RAG, em comparação, é como mobiliar o prédio. A implementação consiste em 'plugar' componentes: um carregador de documentos, um modelo de embedding e um banco de vetores. A atualização do conhecimento é uma simples operação de escrita em um banco de dados, algo trivial em termos de custo computacional.

Portanto, a regra geral é clara: se o seu desafio é injetar conhecimento factual, específico e, principalmente, **mutável** em um LLM, o RAG é quase sempre a resposta correta. Ele oferece um caminho mais barato, rápido e sustentável para manter sua IA aterrada nos fatos e sincronizada com a realidade do seu domínio de negócio.

1.2 DECODIFICANDO A ARQUITETURA: OS BASTIDORES DO RAG

A historinha do Léo e da Ana foi legal, né? Ajudou a dar uma clareada nas ideias. Mas agora vamos tirar os apelidos e colocar os nomes técnicos na mesa. O nosso gênio Léo é o que chamamos de **Generator**, geralmente um Modelo de Linguagem (LLM). A nossa pesquisadora supereficiente Ana é o **Retriever**. A colaboração entre eles forma o **sistema RAG**, e esse processo todo tem dois grandes momentos: uma fase de preparação, que acontece antes mesmo de vocês fazerem qualquer pergunta, e uma fase de execução, que rola em tempo real.

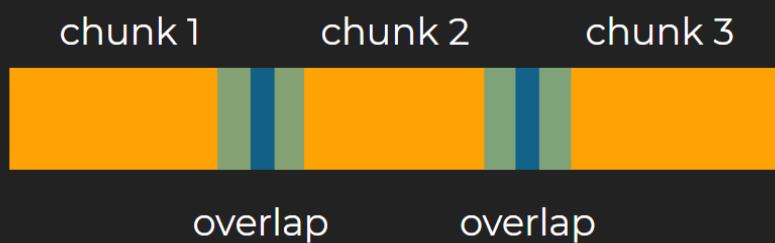
Fase 1: A Preparação (Indexação de Conhecimento)

Pensem na Ana montando o fichário dela. Isso não acontece na hora da prova, no desespero. Ela faz antes, com calma e método. No RAG, esse rolê se chama **Indexação**. É aqui que a gente pega uma montanha de informação desestruturada e a organiza de uma forma que o nosso sistema consiga 'pesquisar' de maneira inteligente. A gente basicamente cria o cérebro externo do nosso LLM.

O primeiro passo é **carregar os documentos**. Esqueçam livros físicos; pensem em arquivos PDF, páginas de um site, documentos de texto, transcrições de vídeos, o que for. Depois de carregar tudo, a gente precisa **quebrar esses documentos em pedaços menores**. A gente chama esses pedaços de 'chunks'. Por que fazemos isso? Simples. Mandar um livro de 500 páginas para o LLM analisar e

RAG - RETRIEVAL-AUGMENTED GENERATION

encontrar uma única frase é ineficiente e caro. É muito mais inteligente entregar só o parágrafo certo. Aqui, vocês já encontram dois parâmetros cruciais: o **chunk_size**, que define o tamanho de cada pedaço de texto, e o **chunk_overlap**, que é uma pequena sobreposição entre os 'chunks' pra garantir que a gente não perca o contexto de uma ideia que começa no final de um pedaço e termina no início do outro. Sacaram?



Criando os mapas

Pensem no universo de todas as ideias como uma galáxia gigante e escura. Cada 'chunk' de texto que vocês criaram, cada conceito, é uma estrela brilhando nessa imensidão. Sozinho, é apenas um ponto de luz. O que o modelo de **Embedding** faz é atuar como um astrônomo superavançado, equipado com um telescópio que enxerga o significado.

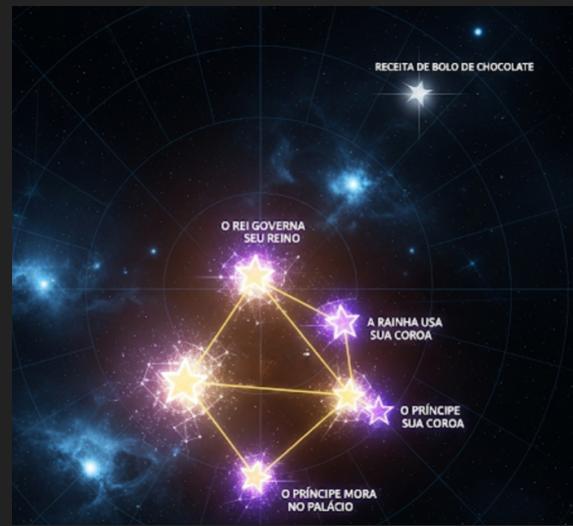
Ele não apenas vê as estrelas; ele cria um mapa tridimensional (na verdade, com centenas de dimensões) dessa galáxia. Para cada estrela, ou seja, para cada 'chunk' de texto, ele atribui um conjunto de coordenadas matemáticas únicas e superprecisas. Essa lista de coordenadas é o **vetor de embedding**.

E aqui vem a parte genial, a mágica do mapa: ele não é aleatório. O nosso astrônomo (o modelo) posiciona as estrelas de forma que aquelas com 'brilho' ou 'energia' conceitual parecida fiquem próximas. A estrela que representa o texto 'O rei governa seu reino' não vai estar perto da estrela 'Receita de bolo de chocolate'. Mas ela vai estar muito próxima das estrelas 'A rainha usa sua coroa' e 'O príncipe mora no palácio'. Elas formam uma '**constelação de significado**'.

Quando vocês fazem uma pergunta, ela também é transformada em uma nova estrela, com suas próprias coordenadas, e colocada nesse mapa. A busca, então, se torna uma tarefa simples: o sistema apenas precisa olhar para a estrela da sua pergunta e identificar quais são as estrelas vizinhas mais próximas. É por isso que a busca é tão poderosa: ela não procura por palavras-chave, ela procura por **vizinhanças de significado** na galáxia das suas informações.

Com os textos todos fatiados, o próximo passo é traduzir esses 'chunks' para uma língua que o computador entende de verdade: números. Esse processo, que é o coração da busca inteligente, se chama **Embedding**. Um modelo de 'embedding' — pensem nele como um tradutor universal de conceitos — pega cada 'chunk' de texto e o converte em um vetor, que é basicamente uma lista gigante

RAG - RETRIEVAL-AUGMENTED GENERATION



de números. A grande mágica é que textos com significados parecidos terão vetores (listas de números) matematicamente próximos. É como dar um CEP superpreciso para cada ideia contida nos seus documentos.

E onde a gente guarda todos esses 'CEPs' de ideias? Num lugar especial chamado **Vector Store**, ou banco de dados de vetores. Pensem nele como o armário de fichas da Ana, mas digital, super-rápido e otimizado para buscar por proximidade. Em vez de procurar por uma palavra-chave exata, a gente entrega o 'CEP' (vetor) da nossa pergunta e ele nos devolve os 'CEPs' mais parecidos que ele tem guardado em questão de milissegundos. É a base da nossa busca semântica.

Fase 2: A Execução (Recuperação e Geração)

Certo, o fichário está pronto e organizado. Agora é hora da prova, o momento em que vocês, usuários, entram em cena. Quando vocês mandam uma pergunta, o sistema inicia a fase de **Recuperação e Geração**. A primeira coisa que ele faz é pegar a pergunta de vocês — 'Qual foi o preço do pão em Paris?' — e usar o **mesmo modelo de embedding** da fase anterior para transformá-la em um vetor. É fundamental que seja o mesmo modelo para que a 'linguagem' dos CEPs seja a mesma.

Com o vetor da pergunta em mãos, o sistema vai até o Vector Store e faz a busca por similaridade. Ele basicamente pergunta: 'Ei, me devolva os **top_k** vetores mais parecidos com este aqui que eu te entreguei'. O parâmetro **top_k** é simplesmente o número de 'chunks' que vocês acham relevante recuperar. Se definirem **top_k** como 5, ele vai pegar os 5 pedaços de texto mais relevantes da sua base de dados para responder àquela pergunta específica.

Esses 'chunks' recuperados são o nosso **contexto**. Agora vem o pulo do gato: o sistema monta um novo prompt, muito mais poderoso, para o LLM. Ele junta a pergunta original de vocês com o contexto que acabou de encontrar. A instrução para o LLM (o nosso Léo) é mais ou menos assim: 'Responda à pergunta 'Qual foi o preço do pão em Paris?' usando **apenas** as informações contidas nos

RAG - RETRIEVAL-AUGMENTED GENERATION

seguintes textos: *[Chunk 1, Chunk 2, Chunk 3...]*'. Ao forçar o LLM a se basear no contexto fornecido, a gente reduz drasticamente a chance de ele inventar coisas, ou como chamamos na área, de 'alucinar'.

E aí está. O LLM, que já é um mestre em interpretar e gerar texto, recebe a pergunta junto com a 'cola' perfeita e factual. A tarefa dele agora não é mais 'lembrar' a resposta, mas sim sintetizar as informações que ele acabou de receber e formular uma resposta coesa, precisa e diretamente baseada nas suas fontes. Entenderam a jogada? O RAG não deixa o LLM mais 'inteligente' no sentido de conhecimento próprio. Ele o torna perfeitamente 'informado' no momento exato da pergunta. E é esse superpoder que vamos aprender a construir e a refinar ao longo deste livro.

1.3 UM RAG SIMPLES

Esta é a forma mais pura e fundamental da nossa arquitetura, o alicerce sobre o qual todas as outras variações são construídas. Pense no RAG Simples, ou 'Vanilla RAG', como o fluxo direto que desenhamos na nossa analogia inicial. É a implementação canônica da colaboração entre o pesquisador focado (o Retriever) e o escritor eloquente (o Generator).

O processo é linear e elegante na sua simplicidade. Tudo começa com a pergunta do usuário. Essa pergunta é transformada em um vetor de embedding e usada para consultar o nosso banco de vetores. O banco, por sua vez, retorna os 'chunks' de texto mais relevantes que ele possui armazenados. Estes 'chunks' são o nosso contexto. A partir daí, a mágica acontece: a pergunta original e o contexto recuperado são combinados em um novo prompt, que é então enviado ao LLM.

A instrução para o LLM é clara e direta: "Responda a esta pergunta usando estritamente as informações fornecidas neste contexto". O LLM não precisa 'lembrar' de nada do seu treinamento massivo; sua única tarefa é sintetizar, extrair e articular uma resposta a partir do material factual que acabou de receber. Não há loops, não há autocorreção, não há memória de conversas passadas. É um sistema de 'input-processamento-output' direto, focado em uma única coisa: responder a uma pergunta de cada vez com a maior fidelidade possível às fontes fornecidas. Dominar este fluxo é a chave, pois cada técnica avançada que exploraremos a seguir nada mais é do que uma adição inteligente ou uma modificação engenhosa neste processo fundamental.

1.4 ELE PODE LEMBRAR, RAG COM MEMÓRIA

O RAG Simples que acabamos de ver é poderoso, mas tem a memória de um peixinho dourado. Ele trata cada pergunta que você faz como se fosse a primeira vez que vocês conversam. Se você perguntar 'Quem foi Santos Dumont?' e depois 'E onde ele nasceu?', um RAG Simples ficaria perdido. Ele não saberia a quem 'ele' se refere. Para construir chatbots e assistentes verdadeiramente úteis,

RAG - RETRIEVAL-AUGMENTED GENERATION

precisamos superar essa amnésia. É aqui que entra o **RAG com Memória**.

A ideia é exatamente o que o nome sugere: dar ao nosso sistema a capacidade de **lembra do que foi dito antes**. Em vez de descartar a conversa após cada resposta, o sistema passa a manter um histórico do diálogo, uma espécie de 'buffer de memória'. Esse histórico se torna uma nova fonte de contexto, tão importante quanto os documentos da nossa base de conhecimento.

Na prática, isso geralmente se manifesta de uma forma muito inteligente. Quando uma nova pergunta chega — como 'E onde ele nasceu?' — o sistema não a envia diretamente para o processo de busca. Primeiro, ele olha para a nova pergunta e para o histórico da conversa. Com essas duas informações, um LLM intermediário reescreve a pergunta para que ela se torne autossuficiente. O sistema transforma a pergunta ambígua 'E onde ele nasceu?' na pergunta clara e completa: 'Onde Santos Dumont nasceu?'.

É essa pergunta reescrita, agora cheia de contexto, que é usada para fazer a busca no Vector Store. O resultado é uma mudança transformadora na experiência do usuário. A interação deixa de ser uma série de perguntas e respostas isoladas e se torna um diálogo fluido e natural. O sistema entende pronomes, resolve ambiguidades e permite que o usuário explore um tópico de forma orgânica. É o primeiro grande passo para transformar nosso sistema de busca em um verdadeiro parceiro conversacional.

1.5 O RAG AUTÔNOMO: AGENT RAG

Se o RAG com Memória deu ao nosso sistema a capacidade de lembrar, o **Agent RAG** o eleva a um novo patamar: o da autonomia e da tomada de decisão. Em vez de seguir uma sequência fixa de passos (buscar e depois gerar), um Agent RAG se comporta como um pequeno 'cérebro' que **decide dinamicamente qual a melhor ação a tomar** para responder a uma pergunta.

Imaginem um investigador. Quando ele recebe uma pergunta complexa, ele não sai buscando em todos os arquivos de uma vez. Ele pensa: "Preciso de mais informações? Qual ferramenta devo usar? Devo consultar o banco de dados interno ou preciso pesquisar na internet? Devo quebrar essa pergunta grande em outras menores?" O Agent RAG simula esse processo de raciocínio.

No coração de um Agent RAG está um LLM que atua como o 'agente' principal, munido de uma lista de **ferramentas**. Essas ferramentas podem ser diversas:

- Uma ferramenta de busca no seu Vector Store (o nosso Retriever padrão).
- Uma ferramenta para fazer buscas na web (como uma API do Google Search).
- Uma ferramenta para executar código ou cálculos.
- Uma ferramenta para consultar uma base de dados estruturada.
- E até mesmo ferramentas para interagir com o usuário e pedir mais esclarecimentos.

RAG - RETRIEVAL-AUGMENTED GENERATION

Quando o agente recebe uma pergunta, ele analisa o seu pedido e o seu histórico de conversa. Com base nisso, ele **decide qual ferramenta usar, se é que precisa usar alguma**. Ele pode decidir que a pergunta é simples e que ele já tem a resposta. Ou, pode concluir que precisa buscar informações em um documento específico, ou que a busca na web é mais apropriada. Depois de usar uma ferramenta, o resultado é enviado de volta para o agente, que avalia se a informação é suficiente ou se ele precisa usar outra ferramenta, ou talvez refinar a busca.

Essa capacidade de planejar, executar e iterar nas suas próprias ações transforma o RAG de um sistema reativo em um sistema proativo. O Agent RAG pode navegar por problemas complexos, consultando diferentes fontes de informação e até mesmo corrigindo seu próprio curso. É um passo crucial em direção a IAs mais inteligentes e capazes de resolver problemas do mundo real de forma mais independente.

1.6 O RAG CORRETIVO: CRAG

No mundo real, nem toda informação é ouro. Às vezes, o nosso 'Retriever' (o buscador de informações) pode trazer documentos irrelevantes, desatualizados ou até mesmo incorretos. Isso pode levar o LLM a gerar respostas ruins ou, o que é pior, a alucinar com base em um contexto falho. É para combater esse problema crítico que surge o **CRAG (Corrective RAG)**.

Pense no CRAG como um rigoroso editor de jornal que, antes de publicar uma matéria, verifica a qualidade das fontes. Ele não confia cegamente no que o repórter (o Retriever) trouxe. Ele adiciona uma camada inteligente de **autoavaliação e correção** à pipeline do RAG, garantindo que o LLM só receba informações de alta qualidade.

A principal inovação do CRAG reside em sua capacidade de **avaliar a pertinência e a qualidade dos documentos recuperados**. Pesquisas recentes na área de RAG, incluindo o artigo original que propôs o CRAG, destacam que a qualidade da recuperação é o gargalo mais comum para a performance. Um CRAG faz o seguinte:

1. Após o Retriever buscar os documentos iniciais, um módulo de **avaliador de qualidade** (geralmente um LLM menor ou um modelo treinado especificamente para essa tarefa) entra em ação.
2. Este avaliador analisa os 'chunks' recuperados e atribui uma 'pontuação de relevância' ou 'confiança'. Ele pode classificar os documentos como "altamente relevantes", "parcialmente relevantes" ou "irrelevantes".
3. Com base nessa avaliação, o CRAG toma uma decisão:
 - Se os documentos são **altamente relevantes**: Eles são enviados diretamente para o LLM gerador.

RAG - RETRIEVAL-AUGMENTED GENERATION

- Se os documentos são **parcialmente relevantes**: O sistema pode decidir que precisa de mais informações. Ele pode, por exemplo, **expandir a busca** para documentos relacionados ou até mesmo fazer uma **nova busca na web** para encontrar informações complementares.
- Se os documentos são **irrelevantes**: O sistema pode optar por **descartá-los** e, talvez, tentar uma abordagem diferente de busca, ou até mesmo indicar que não conseguiu encontrar informações confiáveis.

Essa capacidade de 'corrigir o curso' da recuperação é o que torna o CRAG tão poderoso. Ele não apenas busca, ele **verifica a validade da busca** e age proativamente para melhorar a qualidade do contexto antes que a geração aconteça. Isso resulta em respostas significativamente mais precisas, com menor taxa de alucinação e uma confiança muito maior na informação final apresentada ao usuário. O CRAG é um passo fundamental para tornar os sistemas RAG mais robustos e confiáveis em ambientes de dados dinâmicos e heterogêneos.

1.7 O RAG ADAPTATIVO: ADAPTIVE RAG

A realidade é que nem toda pergunta é criada da mesma forma. Algumas são simples e diretas ("Qual a capital da França?"), enquanto outras são complexas e exigem uma investigação profunda ("Quais foram as implicações econômicas do Tratado de Versalhes para a indústria têxtil alemã?"). Um RAG Simples trata ambas da mesma maneira: busca, concatena e gera. Isso pode ser ineficiente e até prejudicial. Para perguntas simples, a busca pode ser um desperdício de tempo e recursos. Para perguntas complexas, uma única busca pode ser insuficiente. O **Adaptive RAG** resolve isso com uma dose de bom senso computacional.

O Adaptive RAG, como o próprio nome diz, **adapta sua estratégia com base na complexidade da pergunta**. Em vez de seguir um caminho único e rígido, ele primeiro analisa a pergunta e decide qual a rota mais eficiente para chegar à melhor resposta. Pense nele como um triagista experiente em um hospital: ele avalia o paciente (a pergunta) e o direciona para o tratamento correto, seja um curativo rápido ou uma cirurgia complexa.

O fluxo de trabalho geralmente envolve um componente inicial, um 'classificador' ou 'roteador', que examina a pergunta do usuário. Com base nessa análise, o sistema pode decidir por um de vários caminhos:

- **Sem Busca (No Retrieval)**: Para perguntas de conhecimento geral ou conversas informais ("Como você está hoje?"), o sistema pode concluir que o LLM já sabe a resposta e não precisa de busca externa. Isso economiza tempo e poder computacional.
- **Busca Simples (Single-step Retrieval)**: Para a maioria das perguntas factuais, o sistema realiza o nosso processo RAG padrão: uma única busca no banco de vetores para encontrar o

RAG - RETRIEVAL-AUGMENTED GENERATION

contexto relevante.

- **Busca Iterativa ou em Múltiplos Passos (Multi-step Retrieval):** Para perguntas complexas, o sistema pode iniciar um ciclo de busca e raciocínio. Ele pode fazer uma busca inicial, analisar os resultados e decidir que precisa refinar a pergunta e buscar novamente, ou talvez decompor a pergunta original em sub-perguntas e buscar as respostas para cada uma delas antes de sintetizar a resposta final.

Essa capacidade de ajustar o esforço à complexidade da tarefa é o que torna o Adaptive RAG tão eficiente. Ele não desperdiça recursos em problemas fáceis e, ao mesmo tempo, tem a capacidade de aprofundar a investigação quando o desafio exige. É um sistema que não apenas responde, mas primeiro 'pensa' sobre a melhor forma de responder, trazendo um nível de inteligência e otimização muito mais próximo do raciocínio humano.

1.8 O RAG EM GRAFOS: GRAPHRAG

Até agora, nossa visão do RAG tem sido centrada em documentos: artigos, relatórios, páginas da web. Nós quebramos esses textos em pedaços e buscamos os mais relevantes. Mas e se a informação mais valiosa não estiver contida em um parágrafo, mas sim na **relação** entre diferentes pedaços de informação? É para desvendar esse conhecimento conectado que surge o **GraphRAG**.

O GraphRAG troca a nossa tradicional base de dados de vetores por uma estrutura muito mais rica: um **Grafo de Conhecimento (Knowledge Graph)**. Pense em um grafo como um mapa de relacionamentos. Em vez de 'chunks' de texto isolados, temos:

- **Nós (Nodes):** Que representam entidades como pessoas, empresas, lugares ou conceitos (ex: "Santos Dumont", "Paris", "Avião 14-Bis").
- **Arestas (Edges):** Que representam a relação entre essas entidades (ex: "Santos Dumont" *nasceu em* "Palmira", "Santos Dumont" *inventou o* "Avião 14-Bis").

Quando uma pergunta chega a um sistema GraphRAG, a abordagem é completamente diferente. Em vez de simplesmente buscar por similaridade semântica em textos, o sistema primeiro tenta entender as entidades e as relações na pergunta. Ele então traduz essa pergunta em uma consulta que **navega pelo grafo**.

Imagine a pergunta: "Quais inventores brasileiros moraram na mesma cidade que o criador do 14-Bis?". Um RAG tradicional teria muita dificuldade. Ele poderia encontrar documentos sobre inventores e sobre o 14-Bis, mas conectar os pontos seria um desafio. O GraphRAG, por outro lado, executaria uma sequência de passos lógicos:

1. Identifica que o "criador do 14-Bis" é o nó "Santos Dumont".

RAG - RETRIEVAL-AUGMENTED GENERATION

2. Segue a aresta *morou em* a partir do nó "Santos Dumont" para encontrar o nó "Paris".
3. Busca por outros nós com a propriedade *é um* "Inventor Brasileiro" que também tenham uma aresta *morou em* apontando para "Paris".

O resultado é uma resposta precisa, inferida a partir das conexões explícitas no conhecimento. O GraphRAG é uma técnica poderosa para domínios onde as relações são a chave, como em investigações financeiras (seguindo o dinheiro), descobertas científicas (conectando pesquisadores a artigos e a descobertas) e sistemas de recomendação inteligentes. Ele nos permite fazer um tipo de pergunta totalmente novo: não apenas "o quê?", mas principalmente "como se conecta?".

1.9 O RAG HÍBRIDO: HYBRID RAG

O **Hybrid RAG** surge como uma evolução natural quando percebemos que nenhum único mecanismo de recuperação é suficiente para lidar com a diversidade das perguntas do mundo real. Até agora, exploramos diferentes sabores de RAG: o baseado em vetores, o baseado em memória, RAG por Agentes e o em grafos. Mas a pergunta é direta: por que escolher apenas um, se podemos combinar vários e tirar o melhor de cada abordagem?

Pensem no Hybrid RAG como um time multidisciplinar. Em vez de depender só da Ana (nossa Retriever clássica de vetores), trazemos também o Pedro, que é craque em grafos, e a Júlia, especialista em busca lexical. Cada um tem uma lente diferente para enxergar a informação. O segredo do Hybrid RAG está em orquestrar essas lentes para que o sistema selecione, combine ou até mesmo faça um **reranking** inteligente dos resultados.

Na prática, o Hybrid RAG combina múltiplas estratégias de recuperação, como:

- **Busca Vetorial:** Localiza os **chunks** semanticamente mais próximos da pergunta, usando embeddings.
- **Busca Lexical (BM25, TF-IDF):** Garante que palavras-chave exatas não passem despercebidas, algo crucial em domínios jurídicos e médicos.
- **Grafos de Conhecimento:** Permite navegar em relações explícitas, conectando conceitos que não estão no mesmo parágrafo, mas fazem parte da mesma rede de significado.

O desafio, claro, é decidir como combinar esses resultados. Alguns pipelines adotam a estratégia de **união**: trazem todos os resultados de todas as buscas e deixam o LLM fazer a síntese. Outros preferem a **interseção**: só os documentos que aparecem em mais de um método são considerados confiáveis. A abordagem mais avançada é o **reranking neural**, onde um modelo adicional atribui pesos de relevância e reorganiza os documentos, privilegiando os mais consistentes.

Um detalhe importante é o papel dos hiperparâmetros. Ajustar **top_k** em um cenário híbrido não é trivial. Pode-se definir um **top_k** específico para cada recuperador (por exemplo, 10 vetoriais, 5 lexicais

RAG - RETRIEVAL-AUGMENTED GENERATION

e 3 de grafo) ou estabelecer um limite global após o reranking. É nessa engenharia fina que o Hybrid RAG mostra sua força: a flexibilidade de moldar a recuperação ao contexto da pergunta.

Em resumo, o Hybrid RAG é como montar um supertime em vez de apostar todas as fichas em um único jogador. Ele reduz a chance de documentos cruciais ficarem de fora, equilibra precisão e abrangência, e garante maior robustez em domínios onde os dados são heterogêneos, ambíguos ou fortemente conectados. É um passo essencial para levar o RAG de uma solução pontual para uma arquitetura verdadeiramente universal e resiliente.

1.10 O RAG-FUSION: RECIPROCAL RANK FUSION (RRF)

O **RAG-Fusion**, também chamado de **Reciprocal Rank Fusion (RRF)**, é uma técnica refinada de combinação de resultados em buscas híbridas. Até agora, vimos que o Hybrid RAG mistura diferentes recuperadores (vetorial, lexical, grafos). Mas a grande pergunta é: como juntar essas listas de documentos de forma justa e eficiente? É aqui que o RRF brilha.

O truque do RRF é simples, mas genial. Imaginem duas filas de classificação: uma saída da busca vetorial e outra saída da busca lexical. Cada documento aparece em uma posição específica em cada fila (primeiro, segundo, quinto, etc.). O algoritmo RRF pega essas posições e calcula uma pontuação de relevância combinada. Quanto mais alto o documento estiver em qualquer uma das filas, maior será a sua chance de aparecer no resultado final. O efeito é equilibrar os dois mundos: semântica e palavras-chave.

O resultado é que documentos bem classificados em **qualquer um** dos métodos de busca recebem destaque. Isso é fundamental porque, em certos casos, a busca lexical pode capturar um detalhe exato (como o nome de uma lei ou artigo), enquanto a busca vetorial entende melhor o contexto semântico. O RRF garante que nenhum desses sinais seja perdido.

Na prática, o RAG-Fusion melhora muito a robustez do Hybrid RAG. Ele reduz a dependência em um único tipo de busca e cria uma fusão matemática mais estável, garantindo que documentos relevantes tenham chance real de aparecer no contexto entregue ao LLM. Além disso, como é uma técnica leve e independente do modelo, pode ser aplicada em escala, sem precisar de re-treinamento complexo ou modelos adicionais de reranking.

Então é isso: o RAG-Fusion é como aquele juiz justo que pega notas de diferentes jurados e monta um ranking final equilibrado. Ele não deixa que a opinião de um único jurado domine o resultado, mas também não ignora quando alguém dá uma nota muito alta. Para sistemas híbridos de recuperação, o RRF é hoje uma das formas mais práticas e eficazes de combinar evidências.

1.11 HYPOTHETICAL DOCUMENT EMBEDDING

Um dos desafios mais sutis do RAG é o que os pesquisadores chamam de "desalinhamento" entre a pergunta e o documento. Uma pergunta do usuário costuma ser curta, direta e usa palavras-chave específicas ("sintomas de dengue hemorrágica"). Um bom documento que responde a essa pergunta, no entanto, é geralmente longo, detalhado, usa uma linguagem mais formal e pode nem mesmo conter a frase exata da pergunta (ele pode falar de "manifestações clínicas da febre hemorrágica por dengue"). Essa diferença de estilo e conteúdo pode confundir o nosso buscador. É para resolver esse problema que existe uma técnica brilhante e um tanto contraintuitiva: o **Hypothetical Document Embedding (HyDE)**.

O HyDE parte de uma premissa genial: em vez de usar a pergunta para encontrar uma resposta, que tal se a gente usasse uma **resposta ideal (mesmo que falsa)** para encontrar uma resposta real? É um truque mental que funciona surpreendentemente bem. O processo é o seguinte:

1. O sistema recebe a pergunta do usuário ("sintomas de dengue hemorrágica").
2. Em vez de enviar essa pergunta direto para o banco de vetores, ele primeiro a envia para um LLM com uma instrução simples: "Escreva um documento que responda a esta pergunta".
3. O LLM, então, gera uma **resposta hipotética**. Ele pode 'alucinar' detalhes, mas o documento gerado será semanticamente rico, estruturado como uma boa resposta e conterá o vocabulário e os conceitos relevantes (ex: "O paciente pode apresentar febre alta, dor de cabeça intensa, dor retro-orbital, além de manifestações hemorrágicas como petéquias e sangramento gengival...").
4. É este documento hipotético, e não a pergunta original, que é transformado em um vetor de embedding.
5. Finalmente, este vetor, que representa a 'essência' de uma resposta perfeita, é usado para buscar no nosso banco de vetores. A busca agora não é mais "pergunta-documento", mas sim "documento-documento", o que tende a produzir resultados muito mais relevantes.

O HyDE funciona como uma "ponte semântica". Ele traduz a intenção concisa do usuário em um exemplo detalhado do que ele espera encontrar. Mesmo que o documento hipotético contenha imprecisões, seu embedding captura o padrão geral de uma resposta relevante, tornando-o uma "isca" muito mais eficaz para "pescar" os documentos corretos na nossa base de conhecimento. É uma técnica poderosa para melhorar a precisão da recuperação, especialmente para perguntas complexas ou de nicho.

CAPÍTULO 2

O RAG Clássico

A história do RAG Clássico começa em um momento em que os Modelos de Linguagem já mostravam seu brilho, mas também expunham suas fragilidades. Por volta de 2019 e 2020, quando os LLMs como GPT-2 e GPT-3 começaram a impressionar o mundo com textos cada vez mais coerentes e criativos, um problema ficou evidente: eles escreviam muito bem, mas não necessariamente diziam a verdade. As alucinações — respostas convincentes, porém incorretas — se tornaram um ponto crítico que limitava seu uso em cenários profissionais. Era como ter um aluno genial, mas que inventava dados quando não sabia a resposta.

Nesse cenário, surgiu a ideia de conectar esses modelos não apenas à sua memória interna, mas também a fontes externas de informação. Se o modelo é ótimo em linguagem, mas fraco em lembrança factual, por que não deixá-lo buscar os fatos em bases confiáveis antes de escrever a resposta? Esse raciocínio deu origem ao que hoje chamamos de **Retrieval-Augmented Generation**. Em vez de treinar o modelo continuamente com novos dados, que é caro e lento, a solução foi ensiná-lo a consultar documentos já disponíveis. É aqui que nasce o RAG Clássico, a primeira arquitetura que formalizou essa fusão entre busca e geração.

O RAG Clássico foi apresentado em 2020 em um artigo da Meta AI (na época, Facebook AI Research). A proposta era direta, mas poderosa: quando o usuário fazia uma pergunta, o sistema primeiro a convertia em uma representação vetorial e consultava uma base de documentos para recuperar trechos relevantes. Esses trechos eram, então, combinados com a pergunta original e enviados ao LLM, que gerava a resposta fundamentada. Essa simples mudança de fluxo reduziu drasticamente as alucinações e deu origem a uma nova geração de sistemas de IA mais confiáveis.

O link do artigo da Meta está aqui:

<https://arxiv.org/abs/2005.11401>

A importância do RAG Clássico não está apenas em resolver o problema das alucinações, mas em inaugurar um novo paradigma de arquitetura. Ele mostrou que os modelos de linguagem não precisavam carregar todo o conhecimento do mundo dentro de si. Em vez disso, poderiam ser

RAG - RETRIEVAL-AUGMENTED GENERATION

especialistas em interpretação e escrita, enquanto delegavam a tarefa de lembrança a um mecanismo externo de recuperação. Essa separação de funções foi revolucionária porque abriu caminho para IAs atualizáveis em tempo real: bastava adicionar novos documentos ao repositório, sem retreinar o modelo inteiro.

Outro impacto decisivo do RAG Clássico foi econômico. Treinar ou refinar LLMs gigantes sempre exigiu poder computacional massivo e datasets cuidadosamente preparados. O RAG Clássico quebrou essa barreira ao demonstrar que era possível manter um modelo base estático e barato, adicionando conhecimento factual de forma incremental e sob demanda. Isso reduziu custos, acelerou implementações e permitiu que empresas menores também explorassem o poder dos LLMs sem precisar de uma infraestrutura bilionária.

Por fim, o RAG Clássico se consolidou como uma peça fundamental porque trouxe confiabilidade ao centro da conversa sobre IA. Ele mostrou que a genialidade linguística dos LLMs, combinada com a precisão de bases externas, poderia produzir respostas úteis, auditáveis e atualizadas. Em outras palavras, transformou os modelos de linguagem de curiosidades acadêmicas em ferramentas práticas para negócios, saúde, direito, jornalismo e muito mais. Esse é o legado do RAG Clássico: o ponto de partida de uma revolução que ainda está se desdobrando diante de nós.

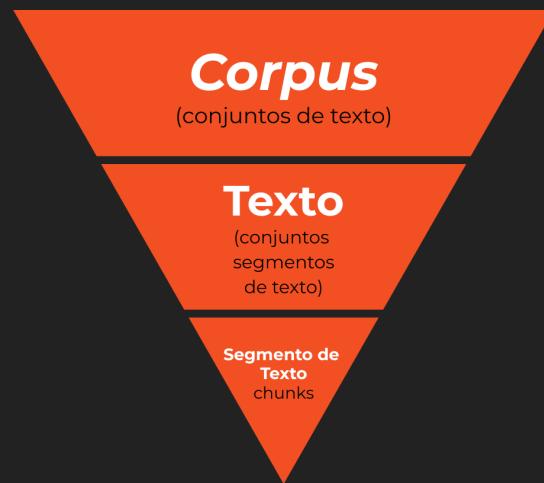
2.1 O QUE É UM CORPUS DE TEXTOS

Quando falamos de RAG, uma palavra aparece o tempo todo: **corpus**. Mas afinal, o que isso significa? Um corpus de textos nada mais é do que uma coleção organizada de documentos que serve como base de conhecimento para o sistema. Esses documentos podem assumir muitas formas: artigos científicos, decisões jurídicas, páginas da web, transcrições de aulas, manuais técnicos ou até mesmo postagens em redes sociais.

O ponto central é que o corpus funciona como a **biblioteca externa** do nosso modelo. Em vez de depender apenas da memória interna do LLM, o sistema consulta essa biblioteca sempre que precisa de informações atualizadas ou especializadas. Quanto mais bem estruturado for o corpus, melhor será a qualidade das respostas.

Vale lembrar que um corpus não precisa ser estático. Ele pode crescer com o tempo, receber novas versões de documentos e até ser segmentado em áreas temáticas. Essa flexibilidade é o que torna o RAG tão poderoso: basta atualizar o corpus e, automaticamente, o modelo passa a trabalhar com conhecimento mais fresco e confiável.

A Divisão do *Corpus*



2.2 O FLUXO DO RAG

O fluxo do RAG pode ser entendido como uma sequência de etapas bem definidas que conectam a preparação dos documentos à geração final da resposta. A Figura 2.1 mostra de maneira clara como esse processo se organiza em duas grandes fases: a fase de **Indexing**, que ocorre de forma offline, e a fase online do próprio RAG, acionada a cada consulta do usuário.

Na parte superior da imagem, vemos a fase de **Indexing**. É aqui que o **corpus**, isto é, o conjunto de documentos brutos, passa por um **Preprocess**, onde o texto é limpo e normalizado. Em seguida, o material é segmentado em pedaços menores por meio do **Chunking**, o que facilita a manipulação e garante granularidade na hora da busca. Esses segmentos, então, são convertidos em vetores pelo módulo de **Embedding**, e finalmente armazenados em um **Vector Database**. Esse banco é a estrutura que permite que, no futuro, as consultas sejam respondidas de forma rápida e precisa.

Na parte inferior da Figura 2.1, está a fase online, isto é, o momento em que o usuário faz sua pergunta. A **Query** é recebida e imediatamente transformada em um embedding, permitindo que fale a mesma língua matemática dos documentos armazenados. Esse vetor da consulta é enviado ao **Retriever**, que consulta o Vector Database e devolve os **chunks** mais relevantes. Na sequência, entra em cena o **Augmented**, que combina a pergunta original com os trechos recuperados, criando um prompt enriquecido. Esse prompt é passado ao módulo de **Generation**, onde o LLM produz a resposta final, agora fundamentada em dados concretos.

Não se preocupe se algum desses termos ainda parece abstrato. Ao longo das próximas seções, vamos desbrinchar cada parte do fluxo separadamente — explicando com calma o que é o **Retriever**, o que significa **Augmented**, como funciona a **Generation** — e mais: você vai implementar em Python cada uma dessas etapas para ver o RAG ganhar vida em código.

RAG - RETRIEVAL-AUGMENTED GENERATION

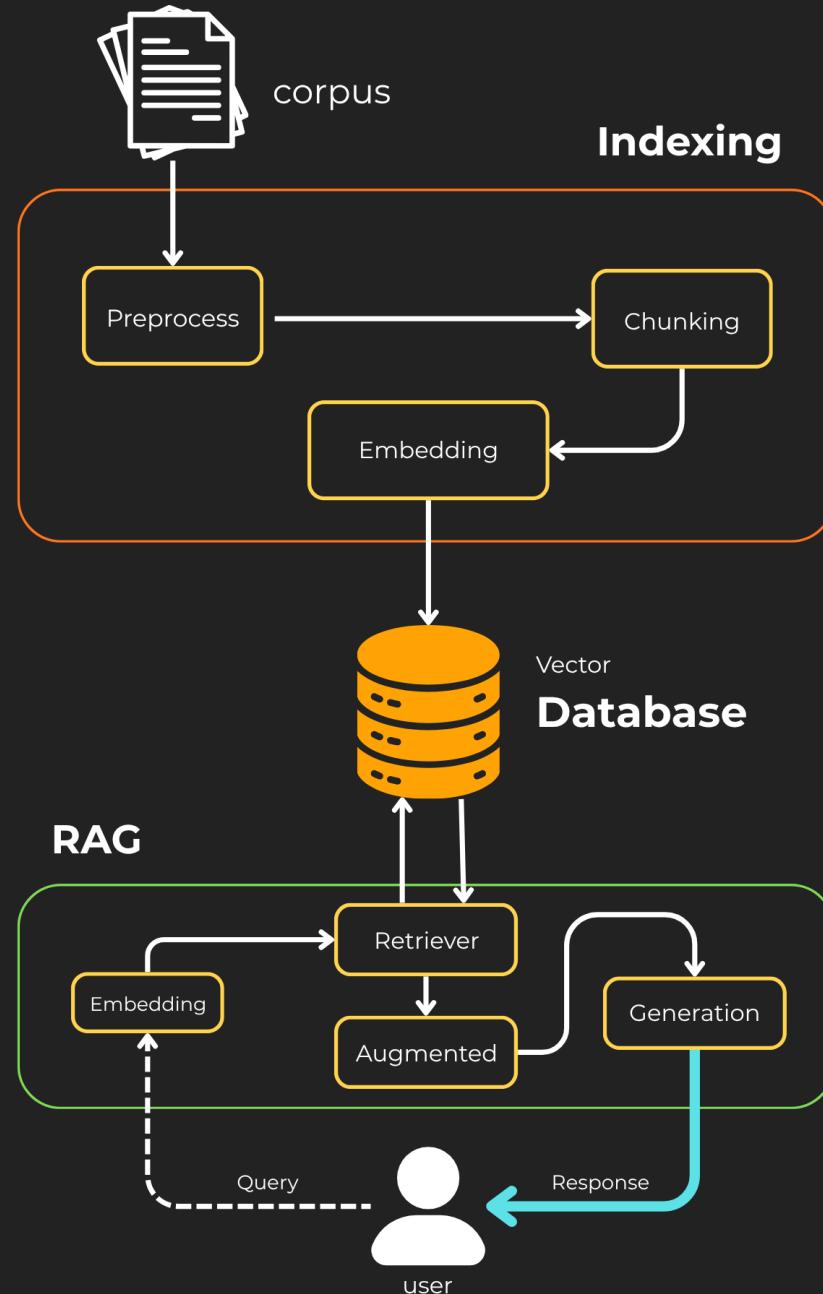


Figura 2.1: O fluxo do RAG Clássico dividido em duas fases: Indexing (offline) e execução do RAG (online).

2.3 A FASE DO INDEXADOR

Imagine uma grande biblioteca que acabou de receber milhares de novos livros. Antes que os leitores possam consultar esse acervo, é necessário que bibliotecários os organizem: cataloguem títulos, autores, assuntos e os coloquem nas prateleiras corretas. Sem esse trabalho de bastidores, encontrar qualquer informação se tornaria uma tarefa caótica e lenta. No RAG, essa função de organização e preparação é realizada pelo **Indexador**.

O Indexador é a etapa responsável por transformar o **corpus** bruto em uma estrutura que possa ser consultada de forma eficiente pelo sistema. Ele começa recebendo os documentos originais, que podem ser artigos, decisões jurídicas, relatórios técnicos ou até postagens em redes sociais.

Em seguida, o Indexador divide os documentos em pedaços menores, chamados de **chunks**. Essa segmentação é essencial porque evita que o sistema tenha que lidar com textos enormes e dispersos. Ao trabalhar com trechos curtos, a busca se torna mais precisa e o risco de perder detalhes importantes diminui. Cada chunk funciona como uma ficha de catálogo, pronta para ser indexada.

Finalmente, cada chunk é convertido em um vetor de **embedding**, que representa matematicamente o seu significado. Esses vetores são armazenados em um **Vector Database**, que é o equivalente digital de um catálogo bem organizado. Quando o usuário fizer uma pergunta, o sistema não precisará ler tudo de novo: bastará consultar o índice construído pelo Indexador. É por isso que essa fase é considerada o alicerce de todo o RAG, garantindo velocidade, precisão e confiabilidade na etapa de recuperação.

Instalando tudo

Para começar, faça o download do nosso projeto no link abaixo:

Link do projeto:

https://bit.ly/sandeco_rag_classico

Certifique-se de que o seu ambiente já esteja preparado com a versão mais estável do python, eu recomendo o **Python 3.12**. Além disso, instale o gerenciador de pacotes **UV**, que será utilizado para configurar as dependências do projeto. Para instalar o UV, execute o seguinte comando no terminal:

```
pip install uv
```

Com o UV instalado, crie um ambiente virtual já apontando para a versão correta do Python e depois sincronize todas as dependências do projeto com os comandos abaixo:

RAG - RETRIEVAL-AUGMENTED GENERATION

```
uv venv --python=3.12  
uv sync
```

2.4 LENDO E CONVERTENDO

A primeira tarefa da indexação é a abertura, leitura e transformação dos documentos em um formato que possa ser manipulado pelo sistema. Por isso, criamos a classe **ReadFiles**. Observe que, ao inicializar essa classe, nenhuma configuração especial é feita no método `__init__`. O foco está em disponibilizar um ponto de entrada para os métodos que realmente executam o processamento. O método central se chama **docs_to_markdown**, e é nele que acontece o fluxo completo: receber um diretório, identificar os arquivos dentro dele, converter cada um para markdown e salvar o resultado.

No início do método **docs_to_markdown**, a lista de arquivos é capturada com a função **read_dir**, que retorna todos os nomes existentes no diretório passado. Para cada arquivo, construa o caminho completo com **os.path.join**. Em seguida, determine a extensão com o comando **file.split('.')[−1]**, que pega exatamente o que está após o último ponto no nome. Esse detalhe é importante porque arquivos como `2111.01888v1.pdf` têm mais de um ponto no nome. O split garante que só a última extensão seja considerada para identificar o tipo.

Se o arquivo for de texto ou documento, como pdf, docx, csv ou similares, crie um objeto **MarkItDown** com plugins habilitados. Caso o arquivo seja uma imagem, inicialize o **MarkItDown** com parâmetros diferentes: passe um cliente **OpenAI**, defina o modelo `gpt-5-mini` e um prompt que instrui o sistema a gerar três parágrafos descrevendo a imagem em português. Essa distinção mostra que o tratamento varia de acordo com o tipo de dado: texto estruturado segue direto para a conversão, enquanto imagens exigem interpretação por meio de linguagem natural.

Depois de converter o arquivo com **md.convert**, salve o resultado em formato markdown dentro da pasta `markdown`. Para isso, use **os.path.splitext** para remover a última extensão e criar um nome limpo, adicionando `.md` ao final. Se a pasta não existir, crie-a com **os.makedirs**. Por fim, abra o arquivo resultante em modo de escrita com codificação `utf-8` e grave o conteúdo. Quando todos os arquivos já foram processados, percorra novamente a pasta `markdown`, leia cada documento salvo e concatene em uma única string, que é devolvida pelo método. Esse comportamento garante que o resultado final da indexação esteja pronto para ser usado na próxima etapa do RAG.

```
#read_files.py  
  
import os  
from markitdown import MarkItDown  
  
class ReadFiles:
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
def __init__(self):
    pass

def docs_to_markdown(self, dir_path):

    docs = self.read_dir(dir_path)

    for file in docs:

        file_path = os.path.join(dir_path, file)

        extension = file.split('.')[ -1]

        if extension == 'pdf' or \
            extension == 'doc' or \
            extension == 'docx' or \
            extension == "xls" or \
            extension == "xlsx" or \
            extension == "ppt" or \
            extension == "pptx" or \
            extension == "csv" or \
            extension == "txt" or \
            extension == "json" or \
            extension == "xml" or \
            extension == "html" or \
            extension == "htm" or \
            extension == "yaml":

            md = MarkItDown(enable_plugins=True)

        elif extension == 'jpg' or \
            extension == 'png' or \
            extension == 'jpeg' or \
            extension == 'gif' or \
            extension == 'bmp' or \
            extension == 'webp' or \
            extension == 'svg' or \
            extension == 'tiff' or \
            extension == 'ico':

            client = OpenAI()

            md = MarkItDown(llm_client=client,
                            llm_model="gpt-5-mini",
                            llm_prompt="""Em 3 parágrafos,
                            descreva a imagem detalhadamente em
                            pt-br""",
                            enable_plugins=True)
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
result = md.convert(file_path)

filename_without_ext = os.path.splitext(file)[0]
md_path = os.path.join("markdown", filename_without_ext + ".md")

if not os.path.exists("markdown"):
    os.makedirs("markdown")

with open(md_path, "w", encoding="utf-8") as f:
    f.write(result.text_content)

md_content = ""

for file in os.listdir("markdown"):

    with open(os.path.join("markdown", file), "r", encoding="utf-8") as f:
        md_content += f.read()

return md_content

def read_dir(self, dir_path):

    files = os.listdir(dir_path)

    return files
```

2.5 CHUNKING

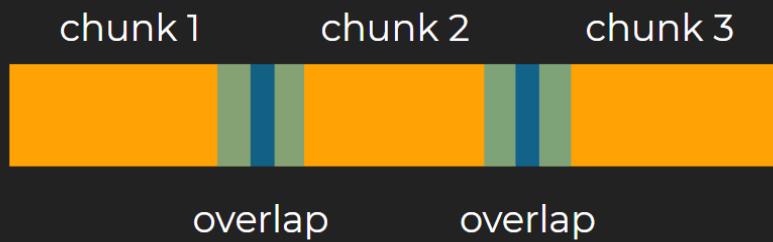
Imagine que você precisa estudar para uma prova de história, mas o livro tem 800 páginas. Se você tentar ler tudo de uma vez, ficará sobrecarregado e não vai conseguir lembrar dos detalhes importantes. A solução natural é dividir o livro em capítulos, depois em seções e, às vezes, até em resumos menores com pontos-chave. Essa é exatamente a lógica do **Chunking** no RAG: pegar documentos grandes e fragmentá-los em pedaços menores e mais fáceis de manipular.

O **Chunking** acontece logo no início do processo de indexação. Cada documento é cortado em blocos de texto — os **chunks**. Esses blocos têm um tamanho pré-definido, controlado pelo parâmetro **chunk_size**. Definir bem esse tamanho é crucial: se o chunk for muito grande, o modelo pode receber informações demais e se perder no excesso de contexto; se for muito pequeno, pode faltar informação para dar sentido ao texto recuperado. Encontrar o equilíbrio é como ajustar a lente de uma câmera: nem perto demais, nem distante demais.

Outro ponto importante é o **chunk_overlap**, que adiciona uma pequena sobreposição entre os

RAG - RETRIEVAL-AUGMENTED GENERATION

chunks. Pense em duas páginas consecutivas de um livro: a frase pode começar no final de uma e terminar no início da outra. Se os blocos não se sobreponerem, esse detalhe pode ser perdido. O overlap garante que o contexto não se quebre de forma brusca, permitindo que a busca recupere informações mais completas e coerentes.



No fim das contas, o **Chunking** é o que transforma documentos extensos em uma base granular e eficiente para o RAG. Ele prepara o material de forma que cada pergunta feita pelo usuário encontre pedaços de informação relevantes, já prontos para serem buscados e usados pelo Retriever. Sem essa etapa, o sistema teria de lidar com blocos desorganizados e desproporcionais, comprometendo tanto a performance quanto a precisão da resposta.

A Classe Chunks

A classe **Chunks** do nosso projeto, possui métodos públicos que permitem controlar todo o processo de fragmentação de textos em pedaços menores e reutilizáveis. O método `create_chunks(text)` recebe um texto em formato de string e o divide em blocos de acordo com o `chunk_size` e o `overlap_size`. Durante essa divisão, ele tenta preservar a coerência natural do conteúdo, procurando pontos adequados de corte, como quebras de parágrafo, final de frases ou espaços entre palavras. O resultado é uma lista de chunks prontos para serem usados em indexação ou busca.

O método `create_chunks_with_metadata(text, source_info)` vai além: além de gerar os chunks, ele também adiciona informações extras a cada um deles. Essas informações incluem um identificador único para cada chunk, o texto em si, o tamanho do chunk, a quantidade total de chunks criados, a posição inicial estimada no texto original e ainda dados opcionais de `source_info`, que podem descrever a origem ou outras características do documento processado. O retorno é uma lista de dicionários, cada um representando um chunk enriquecido com metadados.

O método `get_chunk_info()` retorna um resumo das configurações ativas da classe. Ele mostra o `chunk_size`, o `overlap_size` e o passo efetivo da divisão, calculado pela diferença entre os dois. Essa consulta é útil para validar se os parâmetros atuais estão de acordo com o esperado antes de iniciar a criação de novos chunks.

Por fim, o método `update_settings(chunk_size, overlap_size)` permite atualizar dinamicamente as configurações de chunking. O usuário pode modificar tanto o tamanho dos chunks quanto a

RAG - RETRIEVAL-AUGMENTED GENERATION

sobreposição entre eles, e o método se encarrega de validar os novos valores, garantindo que não haja inconsistências, como uma sobreposição maior que o chunk em si. Esse recurso torna a classe flexível, permitindo ajustes finos para diferentes tipos de textos ou cenários de aplicação.

2.6 CRIAÇÃO DE EMBEDDINGS

Ainda em organização de livros, em vez de guardar os exemplares apenas pelo título, você decide criar um mapa em que cada livro recebe coordenadas de acordo com o seu tema, estilo e vocabulário. Assim, romances parecidos ficam próximos uns dos outros, livros de física ocupam outra região e poesias formam seu próprio agrupamento. Essa é a essência da criação de **embeddings**: transformar textos em pontos de um espaço multidimensional onde a proximidade reflete a semelhança de significado.

Livros Separados Semanticamente

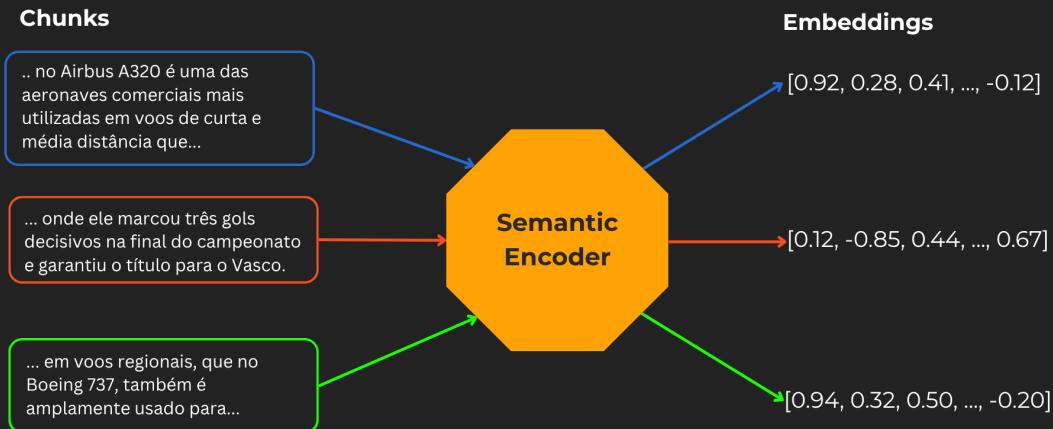


No contexto do RAG, a criação de embeddings é o passo seguinte após o **chunking**. Cada **chunk** é convertido em um vetor numérico, ou seja, uma lista de números que representa o conteúdo semântico daquele pedaço de texto. Essa tradução é feita por um modelo especializado de embedding, que captura padrões de linguagem e os projeta em um espaço vetorial de alta dimensionalidade. O grande benefício é que textos com sentidos parecidos acabam ficando matematicamente próximos uns dos outros. Esse processo de vetorização cria uma espécie de mapa conceitual que torna possível a busca semântica. Em vez de procurar apenas por palavras exatas, o sistema pode encontrar conteúdos relacionados por significado.

Por exemplo: observe as duas Figuras abaixo. Veja como os **chunks** de texto são enviados ao **Semantic Encoder**, que atua como um tradutor semântico. Faça a leitura dessa conversão como uma mudança de idioma: o que antes era linguagem natural passa a ser representado em coordenadas matemáticas chamadas de **embeddings**. Escreva mentalmente essa associação, pois ela será a base para todo o processo de busca semântica no RAG.

RAG - RETRIEVAL-AUGMENTED GENERATION

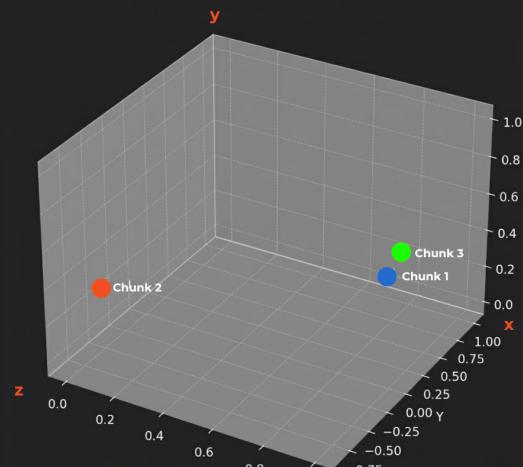
Esses embeddings não são palavras, mas sim posições em um espaço de significados. Quanto mais próximos estiverem dois vetores, mais parecidos são seus conteúdos originais. Isso significa que textos diferentes, mas semanticamente relacionados, ficam próximos nesse mapa multidimensional. Fixe essa noção, porque é exatamente dessa forma que o RAG será capaz de encontrar informações relevantes, não apenas pelo termo usado, mas pelo sentido carregado no contexto.



É aqui que a inteligência do RAG começa a se destacar em relação à simples busca lexical.

Assuntos

Chunk 1 e 3 representam o assunto sobre **Aeronave** e o Chunk 2, **Futebol**.



SentenceTransformer

Para gerar os **embeddings** de texto no nosso pipeline, vamos utilizar a classe `SentenceTransformer`, disponível na biblioteca `sentence_transformers`. Essa classe encapsula modelos de linguagem treinados para converter sentenças, parágrafos ou documentos inteiros em representações vetoriais de alta dimensionalidade. Diferente de vetorizadores simples baseados em frequência de palavras, como TF-IDF, o **SentenceTransformer** captura nuances semânticas, garantindo que textos com significados próximos sejam mapeados para vetores igualmente próximos no espaço vetorial.

RAG - RETRIEVAL-AUGMENTED GENERATION

No nosso caso, adotamos o modelo `paraphrase-multilingual-MiniLM-L12-v2`. Esse modelo é multilíngue, isto é, foi treinado para lidar com diferentes idiomas, incluindo o português, e retorna embeddings com tamanho fixo de **384 dimensões**. Esse valor significa que cada sentença ou documento é convertido em um vetor de 384 números, posicionados em um espaço semântico multidimensional. Nesse espaço, textos com significados semelhantes ficam próximos entre si, enquanto textos de temas diferentes ficam mais distantes. Essa característica é o que permite ao RAG realizar a busca semântica com precisão. Nos próximos capítulos, exploraremos outros modelos de **SentenceTransformer**, comparando seus tamanhos de embeddings, desempenho e adequação a diferentes contextos de uso.

Antes de utilizar a classe, instale a biblioteca no ambiente do projeto com o seguinte comando:

```
uv add sentence_transformers
```

Depois da instalação, vamos testar o modelo em um código simples em Python direto:

```
from sentence_transformers import SentenceTransformer

# Inicializa o modelo multilíngue
model = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')

# Exemplo de textos
sentencas = [
    'O RAG combina busca e geração.',
    'Os embeddings representam o significado de frases.',
    'ChromaDB é um banco de dados vetorial.'
]

# Geração dos embeddings
vetores = model.encode(sentencas)

print(vetores.shape) # (3, 384)
```

2.7 BANCO DE DADOS VETORIAL

Depois de criados, os embeddings são armazenados no **banco de vetores**, que funciona como um catálogo digital extremamente eficiente. Esse banco permite consultas rápidas e precisas: quando uma pergunta chega, ela também é convertida em um vetor e comparada com os vetores já guardados. Os mais próximos são retornados como contexto relevante. Sem a criação de embeddings, não haveria como o sistema localizar informações de forma semântica; seria como tentar navegar na feira de livros sem o mapa, apenas andando às cegas.

ChromaDB

O **ChromaDB** é uma das implementações mais populares de banco de dados vetorial no ecossistema de RAG. Ele foi projetado para armazenar, indexar e consultar **embeddings** de forma eficiente, oferecendo suporte nativo para operações de similaridade semântica. Diferente de bancos relacionais tradicionais, o ChromaDB trabalha em um espaço de alta dimensionalidade, onde cada vetor representa um pedaço de conhecimento. Sua arquitetura otimizada permite consultas extremamente rápidas, mesmo em coleções com milhões de vetores, garantindo que o **Retriever** consiga responder em tempo hábil sem comprometer a precisão.

Outro ponto de destaque do **ChromaDB** é sua simplicidade de uso e integração com frameworks modernos. Ele suporta tanto armazenamento em memória quanto persistência em disco, o que facilita experimentos locais e também projetos em produção. Além disso, disponibiliza recursos como filtros condicionais, atualizações incrementais de coleções e compatibilidade com diferentes formatos de embeddings. Isso torna o ChromaDB uma escolha versátil para aplicações práticas de RAG, equilibrando desempenho, escalabilidade e facilidade de adoção no fluxo de desenvolvimento.

2.8 NOSSA CLASSE DE ENCODER

Agora que entendemos todas as partes da indexação, vamos criar uma classe chamada 'SentenceEncoder' que vai ler os arquivos PDF de uma pasta, transformar os textos em chunks, transformar em embeddings e salvar no ChromaDB.

Comece analisando a primeira parte do código. Escreva as importações que trazem as dependências externas e internas que serão usadas pelo restante da classe. Veja que são importados os módulos Chunks, ReadFiles, o modelo SentenceTransformer, além do chromadb e do uuid. Esses módulos permitem organizar os textos em chunks, transformar em embeddings e salvar no banco vetorial.

```
# arquivo semantic_encoder.py

from chunks import Chunks
from read_files import ReadFiles
from sentence_transformers import SentenceTransformer
import chromadb
import uuid
```

Na sequência, observe a definição da classe SemanticEncoder. Escreva o construtor `__init__` recebendo os parâmetros principais: o diretório dos documentos, o tamanho dos chunks, a sobreposição, o caminho do banco ChromaDB e o nome da coleção. Note que no corpo do `__init__` são inicializadas

RAG - RETRIEVAL-AUGMENTED GENERATION

as dependências essenciais: a leitura de arquivos, o chunker, o modelo de embeddings multilíngue, o cliente persistente do ChromaDB e o atributo collection.

```
class SemanticEncoder:  
    """  
        Constrói a base vetorial e popula o ChromaDB a partir de documentos em um diretório.  
    """  
  
    def __init__(  
        self,  
        docs_dir: str,  
        chunk_size: int,  
        overlap_size: int,  
        db_path: str = "./chroma_db",  
        collection_name: str = "documentos_rag",  
    ) -> None:  
        self.docs_dir = docs_dir  
        self.chunk_size = chunk_size  
        self.overlap_size = overlap_size  
        self.db_path = db_path  
        self.collection_name = collection_name  
  
        # Dependências  
        self.rf = ReadFiles()  
        self.chunker = Chunks(chunk_size=self.chunk_size, overlap_size=self.  
overlap_size)  
        self.modelo = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')  
        self.client = chromadb.PersistentClient(path=self.db_path)  
        self.collection = None
```

Agora concentre-se no método build. Execute cada etapa com clareza: primeiro, leia os documentos e converta-os em markdown. Depois, crie os chunks de texto e aplique o modelo para gerar os embeddings. Observe que os embeddings são transformados em listas porque o ChromaDB exige esse formato. Em seguida, recrie ou obtenha a coleção no banco vetorial. Se reset_collection for True, apague a coleção existente antes de prosseguir.

```
def build(self, reset_collection: bool = True, collection_name: str = None) -> dict:  
    # 1) Ler documentos  
    mds = self.rf.docs_to_markdown(self.docs_dir)  
  
    # 2) Criar chunks  
    text_chunks = self.chunker.create_chunks(mds)  
  
    # 3) Gerar embeddings  
    base_vetorial_documentos = self.modelo.encode(text_chunks)  
    embeddings = base_vetorial_documentos.tolist()
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
# 4) (Re)criar/obter coleção
if reset_collection:
    try:
        self.client.delete_collection(name=collection_name)
        print(f"Coleção '{collection_name}' existente foi deletada.")
    except Exception:
        pass

    try:
        self.collection = self.client.get_collection(name=collection_name)
    except Exception:
        self.collection = self.client.create_collection(
            name=collection_name,
            metadata={"description": "Coleção de chunks de documentos com embeddings"
        },
    )
```

Finalize entendendo a inserção dos dados no ChromaDB. Crie identificadores únicos com uuid, construa metadados com informações do chunk e use o método add para inserir embeddings, documentos e metadados na coleção. Por fim, imprima as estatísticas do processo e retorne um dicionário com os resultados.

```
# 5) Inserir dados
ids = [str(uuid.uuid4()) for _ in range(len(text_chunks))]
metadatas = [
    {
        "chunk_id": i,
        "chunk_size": len(chunk),
        "source": self.docs_dir,
    }
    for i, chunk in enumerate(text_chunks)
]

self.collection.add(
    ids=ids,
    embeddings=embeddings,
    documents=text_chunks,
    metadatas=metadatas,
)

print(f" Salvos {len(text_chunks)} chunks no ChromaDB!")
print(
    f" Coleção '{self.collection_name}' agora possui {self.collection.count()} documentos"
)
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
        return {
            "chunks_salvos": len(text_chunks),
            "colecao": self.collection_name,
            "total_documentos": self.collection.count(),
        }
```

Criando uma main

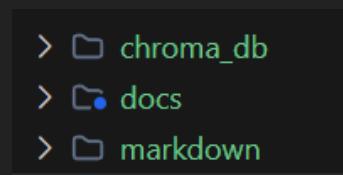
Por último, veja o bloco principal que é executado quando o arquivo é rodado diretamente. Observe a criação do objeto SemanticEncoder com os parâmetros de diretório, chunk e overlap. Em seguida, execute o método build para construir a base vetorial e imprimir estatísticas.

```
if __name__ == "__main__":
    encoder = SemanticEncoder(
        docs_dir="docs", #diretório dos documentos
        chunk_size=2000, #tamanho do chunk
        overlap_size=500, #tamanho da sobreposição
    )

    # Construir base vetorial
    stats = encoder.build(collection_name="synthetic_dataset_papers")

    # Imprimir estatísticas
    print(stats)
```

Depois da execução do arquivo `main.py`, surgem automaticamente duas pastas fundamentais para o funcionamento do projeto. A pasta `chroma_db` é responsável por armazenar o banco vetorial utilizado nas buscas semânticas do RAG, enquanto a pasta `markdown` concentra os arquivos convertidos a partir dos documentos originais presentes em `docs`. Essa organização garante que os dados brutos fiquem separados dos conteúdos processados e que o índice vetorial possa ser consultado de forma rápida e eficiente.



2.9 RECUPERANDO CONHECIMENTO

Imagine que vai alguém que vai entrar na sua biblioteca gigantesca com milhares de livros e, no balcão, encontrar um bibliotecário extremamente eficiente. Ele faz uma pergunta e, em vez de ler todos os volumes, esse bibliotecário sabe exatamente em quais prateleiras e páginas procurar. Ele não te entrega o livro inteiro, mas destaca apenas os trechos mais relevantes que respondem à sua dúvida. Esse bibliotecário é a perfeita analogia para o **Retriever** dentro do RAG.

O papel do Retriever é localizar no **corpus** os pedaços de texto mais próximos daquilo que foi perguntado. Para isso, ele transforma a pergunta em uma representação numérica chamada **embedding**, um vetor de números que captura o significado semântico do enunciado. Em seguida, o Retriever consulta o banco de vetores, comparando essa representação com as representações já armazenadas dos documentos. O resultado é uma lista de **chunks** que estão semanticamente mais próximos da pergunta.

É importante perceber que o Retriever não inventa nada, ele apenas busca. A sua função é trazer para o modelo gerador a parte da informação que realmente importa. Se pensarmos novamente na analogia da biblioteca, ele não dá opiniões, não interpreta, apenas recupera. A qualidade do trabalho do Retriever depende de fatores como a escolha do modelo de embedding, o **chunk_size** usado para dividir os documentos e o parâmetro **top_k**, que define quantos trechos devem ser retornados.

Sem o Retriever, o RAG não teria como se apoiar em informações externas. Ele é a ponte entre o modelo de linguagem e a base de conhecimento. É por meio dele que a resposta deixa de ser apenas uma construção da memória do LLM e passa a ser fundamentada em dados concretos. Em resumo, o Retriever é quem garante que a conversa com a máquina não seja apenas bonita, mas também informada.

A classe de Recuperação

Agora vamos construir o **retriever**, que será o responsável por realizar buscas semânticas na base vetorial. Observe a primeira parte do código. Escreva o cabeçalho com o nome do arquivo e faça as importações necessárias: o módulo chromadb, a classe SentenceTransformer para geração de embeddings e o módulo sys para permitir encerrar a aplicação em caso de erro.

```
# arquivo retriever.py

import chromadb
from sentence_transformers import SentenceTransformer
import sys
```

RAG - RETRIEVAL-AUGMENTED GENERATION

Na sequência, defina a classe Retriever. Execute o método `__init__` passando o caminho para o banco ChromaDB e o nome da coleção. Dentro do construtor, inicialize os atributos que armazenam o cliente, a coleção e o modelo de embeddings. Em seguida, chame o método interno `_initialize()` para configurar a conexão com o banco vetorial e carregar o modelo.

```
class Retriever:  
    def __init__(self, db_path=".chroma_db", collection_name=""):  
        """  
        Inicializa o sistema de query RAG.  
        """  
        self.db_path = db_path  
        self.collection_name = collection_name  
        self.client = None  
        self.collection = None  
        self.modelo = None  
  
        self._initialize()
```

Agora concentre-se no método `_initialize`. Execute a conexão persistente com o ChromaDB utilizando o caminho configurado e recupere a coleção pelo nome fornecido. Em seguida, carregue o modelo de embeddings `paraphrase-multilingual-MiniLM-L12-v2`. Repare que há mensagens de saída informando a conexão bem-sucedida e o número de documentos na coleção. Se ocorrer algum erro, capture a exceção, informe o usuário e encerre o programa com `sys.exit(1)`.

```
def _initialize(self):  
    """Inicializa o cliente ChromaDB e carrega o modelo."""  
    try:  
        # Conectar ao ChromaDB  
        self.client = chromadb.PersistentClient(path=self.db_path)  
        self.collection = self.client.get_collection(name=self.collection_name)  
  
        # Carregar modelo de embeddings  
        print("Carregando modelo de embeddings...")  
        self.modelo = SentenceTransformer('paraphrase-multilingual-MiniLM-L12-v2')  
    )  
  
    print(f"Conectado à coleção '{self.collection_name}'")  
    print(f"Total de documentos: {self.collection.count()}")  
  
except Exception as e:  
    print(f"Erro ao inicializar: {e}")  
    print("Certifique-se de que o banco ChromaDB foi criado executando  
rag_classic.py primeiro.")  
    sys.exit(1)
```

RAG - RETRIEVAL-AUGMENTED GENERATION

Por fim, veja o método `search`. Escreva a query em linguagem natural e transforme-a em embedding usando o modelo carregado. Depois, utilize o método `query` do ChromaDB para buscar documentos semelhantes, especificando o número de resultados e pedindo que sejam retornados documentos, distâncias e metadados. Note que o código extrai a primeira lista de documentos recuperados e a retorna como resultado da busca. Caso aconteça algum erro, capture a exceção e retorne `None`.

```
def search(self, query_text, n_results=5, show_metadata=False):
    """
    Busca documentos similares à query.
    """
    try:
        # Gerar embedding da query
        query_embedding = self.modelo.encode([query_text])

        # Buscar no ChromaDB
        results = self.collection.query(
            query_embeddings=query_embedding.tolist(),
            n_results=n_results,
            include=['documents', 'distances', 'metadatas']
        )

        res = results['documents'][0]

        return res
    except Exception as e:
        print(f"Erro na busca: {e}")
        return None
```

2.10 AUMENTO DE INFORMAÇÃO

O **Augmented** é a etapa do RAG em que a consulta original do usuário deixa de ser tratada de forma isolada e passa a ser enriquecida com informações adicionais provenientes do **Retriever**. Em outras palavras, é aqui que o sistema pega a pergunta feita pelo usuário e a amplia com os trechos mais relevantes recuperados da base vetorial. Esse processo é chamado de **aumento de informação** porque transforma um simples enunciado em um contexto muito mais rico, capaz de guiar o modelo de linguagem a produzir respostas fundamentadas.

No nosso caso específico, o **Augmented** não envolve nenhuma arquitetura complexa, mas sim a construção de um *prompt* cuidadosamente montado. O que fazemos é pegar a **query** do usuário

RAG - RETRIEVAL-AUGMENTED GENERATION

e combiná-la com os **chunks** selecionados pelo **Retriever**. Dessa forma, o modelo de linguagem não precisa inventar informações do zero: ele já recebe como entrada uma pergunta acompanhada de passagens textuais que têm grande chance de conter a resposta correta ou, pelo menos, partes essenciais dela.

Esse simples mecanismo de concatenação garante que a saída gerada esteja conectada aos documentos originais. Com isso, ao invés de depender apenas da memória estatística do modelo, fornecemos evidências explícitas no prompt. Esse processo fortalece a confiabilidade da resposta e reduz drasticamente o risco de alucinações. Portanto, o **Augmented** é a ponte que une o que foi recuperado na base vetorial com o poder de geração do modelo, dando ao RAG sua característica essencial: responder de forma criativa sem perder a ligação com os dados de origem.

A classe `Augmentation` é responsável por construir o **prompt** que será enviado ao modelo de linguagem. Observe que ela possui um método estático chamado `generate_prompt`, que recebe dois parâmetros: a **query** do usuário e os **chunks** selecionados pelo **Retriever**. Primeiro, defina um separador visual para organizar melhor os blocos de texto. Em seguida, formate os chunks, adicionando um cabeçalho chamado `Conhecimento` para indicar ao modelo que aquele conteúdo representa a base de dados disponível. Por fim, construa uma string estruturada em que o texto da consulta aparece delimitado por `<query>` e os chunks aparecem entre `<chunks>`. Essa formatação garante que o modelo saiba claramente o que é a pergunta e quais são as evidências textuais que deve usar na resposta. Ao retornar esse prompt, a classe entrega uma entrada enriquecida e padronizada, pronta para ser utilizada pelo RAG.

```
class Augmentation:
    def __init__(self):
        pass

    @staticmethod
    def generate_prompt(query_text, chunks):

        separador = "\n\n-----\n\n"

        # Junta os chunks com o separador e adiciona o cabeçalho
        chunks_formatados = f"Conhecimento\n-----\n\n{separador}.
        join(chunks))"

        prompt = f"""Responda em pt-br e em markdown, a query do usuário delimitada
        por <query>
        usando apenas o conhecimento dos chunks delimitados por <chunks>.
        Combine as informações dos chunks para responder a query de forma unificada.
        Se por acaso
        o conhecimento não for suficiente para responder a query, responda apenas
        que não temos conhecimento suficiente para responder a query.

        <chunks>
```

```
{chunks_formatados}  
</chunks>  
  
<query>  
{query_text}  
</query>  
"""  
  
return prompt
```

2.11 GERANDO A RESPOSTA

A classe Generation funciona como um **Adapter** que encapsula a comunicação com o modelo da Google Gemini, simplificando sua utilização no fluxo do RAG. No construtor `__init__`, observe que primeiro é carregado o arquivo `.env` para obter a chave da API de forma segura, evitando expor informações sensíveis no código. Em seguida, a classe inicializa o cliente da API do Gemini com a chave recuperada e define o modelo a ser utilizado, que por padrão é o `gemini-2.5-flash`. Dessa forma, ao instanciar a classe, todo o processo de configuração de credenciais e escolha do modelo já está resolvido, permitindo uma integração direta e transparente.

O método `generate` é o ponto em que a geração da resposta acontece de fato. Ele recebe o `prompt` já preparado na etapa de **Augmented** e o envia ao serviço do Gemini utilizando a função `generate_content`. A resposta retornada pela API é então extraída e devolvida apenas como texto, pronta para ser consumido no pipeline do RAG. Esse design de Adapter é importante porque desacopla a lógica do RAG da implementação específica do provedor de LLM. Assim, se no futuro for necessário trocar o modelo ou até mesmo o provedor, basta modificar essa classe, mantendo o restante do código intacto.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
from google import genai
from dotenv import load_dotenv
import os

class Generation:

    def __init__(self, model="gemini-2.5-flash"):
        load_dotenv()
        self.client = genai.Client(api_key=os.getenv("GEMINI_API_KEY"))
        self.model = model

    def generate(self, prompt):
        client = genai.Client()

        response = client.models.generate_content(
            model=self.model,
            contents=prompt,
        )

        return response.text
```

2.12 RODANDO COM STREAMLIT

O código abaixo representa a junção de todas as etapas do RAG: **Retriever**, **Augmented** e **Generation**. Observe que inicialmente são importadas as três classes que criamos: Retriever, Augmentation e Generation. Em seguida, instanciamos cada uma delas. No caso do Retriever, utilizamos a coleção chamada "synthetic_dataset_papers", que foi construída anteriormente quando indexamos 12 artigos em inglês sobre Banco de Dados Sintéticos. Essa informação é fundamental, pois agora o sistema poderá recuperar trechos desses artigos para fundamentar a resposta.

Na sequência, definimos a variável query com a pergunta "What's a synthetic dataset?". Essa consulta será utilizada para buscar trechos relevantes dentro da coleção indexada. O método search da classe Retriever retorna os chunks mais próximos semanticamente da query, e esses trechos são então passados para o método generate_prompt da classe Augmentation, que monta um prompt enriquecido combinando a pergunta e os pedaços de conhecimento selecionados. Esse prompt é entregue à classe Generation, que, utilizando o modelo gemini-2.5-flash, produz a resposta final em linguagem natural. Por fim, o resultado é impresso na tela, mostrando a saída gerada pelo RAG a partir dos artigos indexados.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
from retriever import Retriever
from augmentation import Augmentation
from generation import Generation

retriever = Retriever(collection_name="synthetic_dataset_papers")
augmentation = Augmentation()
generation = Generation(model="gemini-2.5-flash")

query = "What's a synthetic dataset?"

# Buscar documentos
chunks = retriever.search(query, n_results=10, show_metadata=False)
prompt = augmentation.generate_prompt(query, chunks)

# Gerar resposta
response = generation.generate(prompt)

print(response)
```

agora vou adicionar "Streamlit"ao código e executar o arquivo **app.py** que está no projeto usando:

```
streamlit run app.py
```

A figura abaixo é o resultado da execução do RAG com Streamlit.

RAG - RETRIEVAL-AUGMENTED GENERATION

🕒 RAG Chat Assistant

 what is a synthetic dataset?

 Um *dataset* sintético refere-se a dados gerados artificialmente, em vez de serem coletados no mundo real. O principal objetivo da criação de *datasets* sintéticos é superar o desafio da criação manual e tediosa de grandes *datasets* anotados, necessários para treinar modelos de *deep learning* orientados a dados.

A geração de dados sintéticos pode ser realizada de duas maneiras principais:

1. **Deep Learning:** Utilizando modelos como *Generative Adversarial Networks* (GANs) e *Variational Autoencoders* (VAEs).
2. **Renderização 3D:** Empregando softwares de modelagem 3D como Unity, Unreal ou Blender. Esta abordagem frequentemente visa usar modelos realistas e/ou sensores realistas, mas também pode envolver o uso de modelos realistas de objetos aumentados com texturas irrealistas para enfatizar a forma geométrica durante o processo de aprendizado. Pipelines procedurais são utilizados para controlar a diversidade e cobrir variações do mundo real, criando *datasets*平衡ados.

Os dados sintéticos têm o potencial de gerar *datasets*平衡ados que incluem todas as versões desejadas de dados reais, que podem ser difíceis ou impossíveis de obter na prática. Eles permitem o controle da diversidade em um *dataset*, abrangendo variações como modelos, fundos, iluminação, horários do dia, condições climáticas, posições e orientações, ângulos de

Digite sua pergunta... 

2.13 EXERCÍCIOS

Para consolidar os conceitos apresentados neste capítulo sobre o RAG Clássico, resolva os exercícios abaixo. Eles foram elaborados para estimular tanto a compreensão teórica quanto a prática de implementação em Python.

1. Explique com suas próprias palavras qual foi a principal limitação dos LLMs que levou ao surgimento do RAG Clássico.
2. Diferencie **corpus** e **embedding**. Dê um exemplo prático para cada um deles.
3. Descreva em que consiste a fase de **Indexing** e explique por que ela é considerada offline.
4. Qual a função do parâmetro **chunk_overlap** na classe Chunks? Dê um exemplo de como ele evita a perda de contexto.
5. Utilize a classe ReadFiles para converter ao menos dois arquivos PDF para markdown. Mostre o resultado em um único arquivo concatenado.
6. Gere embeddings para três frases de sua escolha utilizando o modelo `paraphrase-multilingual-MiniLM-L12-v2` e verifique o tamanho do vetor retornado.
7. Crie uma coleção no ChromaDB chamada `meu_corpus_teste` e insira ao menos cinco chunks. Em seguida, utilize o método `count()` para verificar quantos documentos foram armazenados.
8. Explique a diferença entre o papel do **Retriever** e do **Augmented**. Por que não podemos pular a etapa de aumento de informação?
9. Implemente uma pequena aplicação em Python que recebe uma **query** do usuário, recupera três chunks relevantes no ChromaDB e monta o prompt final com a classe `Augmentation`.
10. Explique o conceito de **Adapter** no contexto da classe Generation. Por que ele é importante para desacoplar o código do provedor de LLM utilizado?

CAPÍTULO 3

Rag com Memória

Imagine um professor que não apenas responde perguntas de seus alunos usando livros e artigos, mas que também se lembra de todas as conversas passadas em sala de aula. Esse professor não precisaria repetir explicações já dadas e conseguiria contextualizar cada nova pergunta com base no que já foi discutido. De forma análoga, o RAG com memória amplia as capacidades de um modelo de linguagem ao unir a recuperação de informações externas com a lembrança contínua de interações anteriores.

O RAG tradicional funciona como um consultor que tem acesso imediato a uma vasta biblioteca. Ele consulta as fontes a cada pergunta, mas não necessariamente se recorda do diálogo que já ocorreu. Isso garante precisão factual, mas pode limitar a continuidade da experiência, já que cada resposta é construída como se fosse a primeira. Ao incorporar memória, passamos a ter um consultor que anota, organiza e retoma pontos importantes ao longo da jornada.

Essa evolução traz um aspecto fundamental: o contexto não se esgota na consulta isolada, mas se prolonga em uma linha narrativa. O sistema consegue retomar tópicos discutidos em interações passadas, adaptar suas respostas conforme o histórico e até corrigir mal-entendidos com base em lembranças. Esse tipo de continuidade é particularmente útil em aplicações educacionais, jurídicas, médicas e em qualquer domínio onde a progressão do diálogo importa tanto quanto a resposta imediata.

É nesse cenário que entra o conceito de memória no RAG. Ao unir o mecanismo de **recuperação** com a capacidade de reter conversas passadas, criamos um sistema híbrido mais próximo da forma como os humanos constroem conhecimento. Cada nova pergunta deixa de ser um evento isolado e passa a ser parte de uma trajetória de aprendizado ou investigação. Essa memória pode ser temporária, descartada após certo período, ou persistente, armazenada para uso contínuo.

Por fim, é importante notar que essa integração não se trata apenas de armazenar dados, mas de criar uma dinâmica onde a **geração**, a **memória** e a **recuperação** trabalham em conjunto. A memória fornece continuidade, a recuperação garante precisão e a geração traz fluidez na linguagem. O resultado é um RAG que não apenas responde, mas acompanha, evolui e participa de um diálogo de longo prazo com o usuário.

RAG - RETRIEVAL-AUGMENTED GENERATION

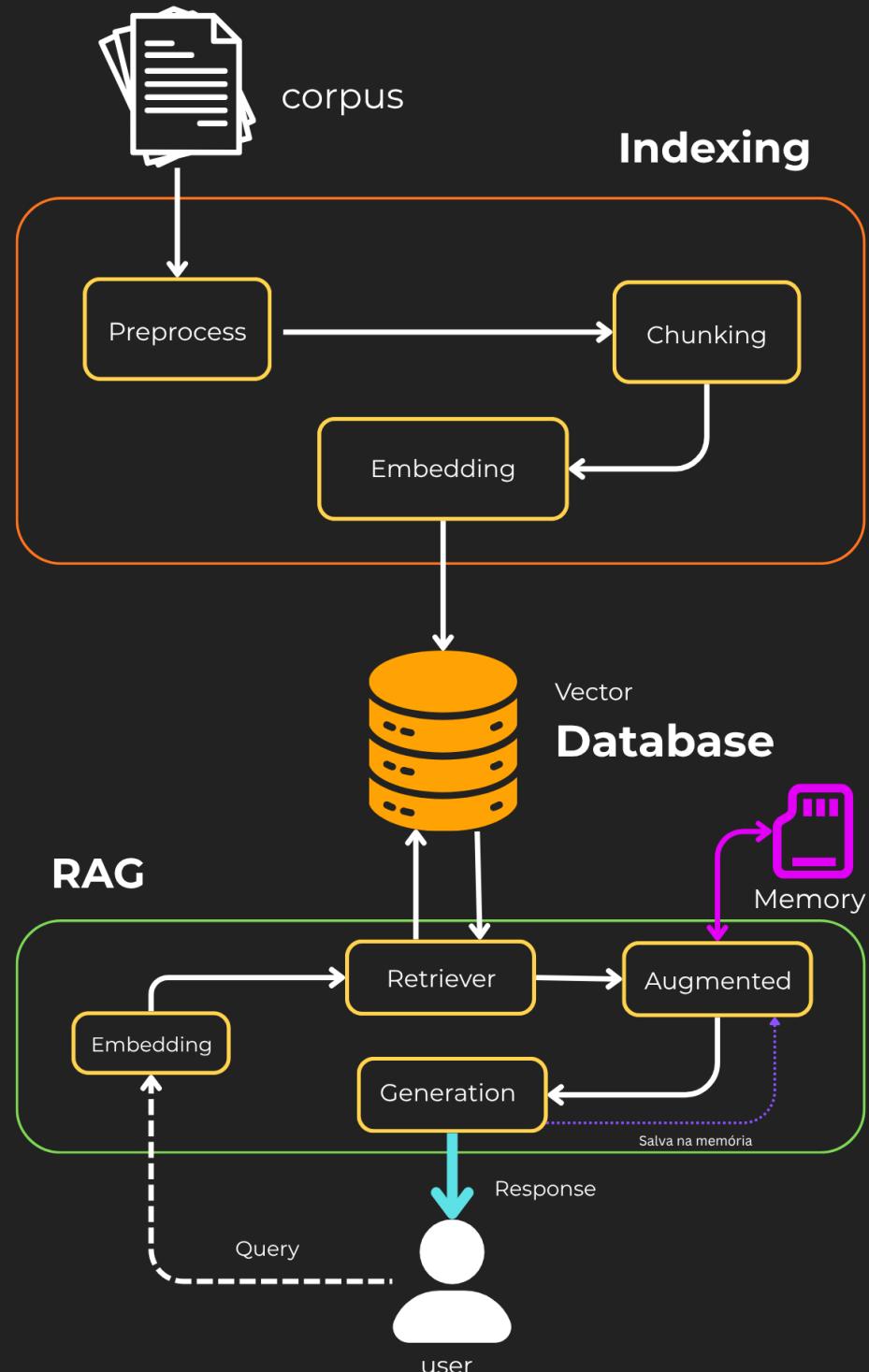


Figura 3.1: RAG com Memória é acoplado ao aumento de informações "Augmented!"

Baixando o Projeto

Para começar, faça o download do nosso projeto no link abaixo:

Link do projeto:

<https://bit.ly/sandeco-rag-memory>

3.1 CRIANDO A MEMÓRIA COM O REDIS

Para que o RAG consiga se lembrar de interações passadas, precisamos de um mecanismo de armazenamento rápido, confiável e que permita o acesso a dados em tempo real. Nesse ponto, entra o Redis. Pense no Redis como um quadro branco colocado no centro de uma sala: todos podem escrever nele, apagar ou atualizar anotações, e o acesso é imediato. Ele não é apenas um banco de dados tradicional, mas um sistema de armazenamento em memória, desenhado para velocidade extrema e simplicidade.

O Redis é um banco de dados do tipo chave-valor, o que significa que cada informação é armazenada como um par: uma chave única que identifica os dados e o valor correspondente que contém o conteúdo. Essa estrutura simples é poderosa porque permite recuperar informações em questão de milissegundos. No caso do RAG, cada conversa pode ser registrada sob uma chave que representa o identificador do usuário ou da conversa, enquanto o histórico de mensagens é o valor associado.

CHAVES E VALORES

Chave	Valor
chave1	valor1
chave2	valor2
chave3	valor3

Uma das grandes vantagens do Redis é sua capacidade de trabalhar totalmente em memória,

o que reduz drasticamente a latência. Em outras palavras, ao invés de buscar os dados em disco rígido como bancos relacionais tradicionais, o Redis mantém tudo diretamente na memória RAM. Isso faz com que operações de escrita e leitura sejam quase instantâneas, característica essencial para aplicações interativas como assistentes baseados em RAG que dependem de fluidez no diálogo.

Além disso, o Redis possui recursos adicionais que o tornam ainda mais adequado para o papel de memória. Ele permite configurar expiração automática de dados, garantindo que informações antigas possam ser descartadas após um tempo definido, por exemplo: 24h, 5 dias ou 1 semana. Isso é particularmente útil em cenários nos quais não faz sentido manter diálogos antigos para sempre. Dessa forma, conseguimos um equilíbrio entre desempenho, praticidade e controle do ciclo de vida das conversas armazenadas.

3.2 REDIS NO DOCKER

Não existe Redis para Windows e o Docker é uma ferramenta indispensável quando falamos de facilidade e padronização na instalação de aplicações como a Redis. Ele permite que você crie um ambiente isolado para rodar o banco, sem se preocupar com as configurações específicas do sistema operacional ou conflitos entre dependências. Com o Docker, tudo o que você precisa está encapsulado em um contêiner, garantindo que o ambiente de execução seja idêntico ao utilizado em produção. Essa abordagem elimina a famosa frase 'na minha máquina funciona', proporcionando consistência desde o desenvolvimento até o deploy.

Para os usuários do Windows, utilizaremos o Docker Desktop, uma interface amigável que facilita o gerenciamento dos contêineres e imagens necessários para rodar a Redis localmente. Esse processo não apenas agiliza a configuração do ambiente, mas também cria um espaço seguro para você desenvolver e testar a aplicação antes de implantá-la em uma VPS ou outro servidor remoto. O Docker Desktop transforma sua máquina local em uma plataforma robusta para experimentação, garantindo que você esteja pronto para levar seu projeto ao próximo nível com confiança.

Instalando o Docker Desktop para Windows

Vamos nessa, vamos instalar o Docker Desktop, que é a maneira mais fácil de gerenciar contêineres no Windows. Siga os passos abaixo para realizar a instalação de forma rápida e sem complicações:

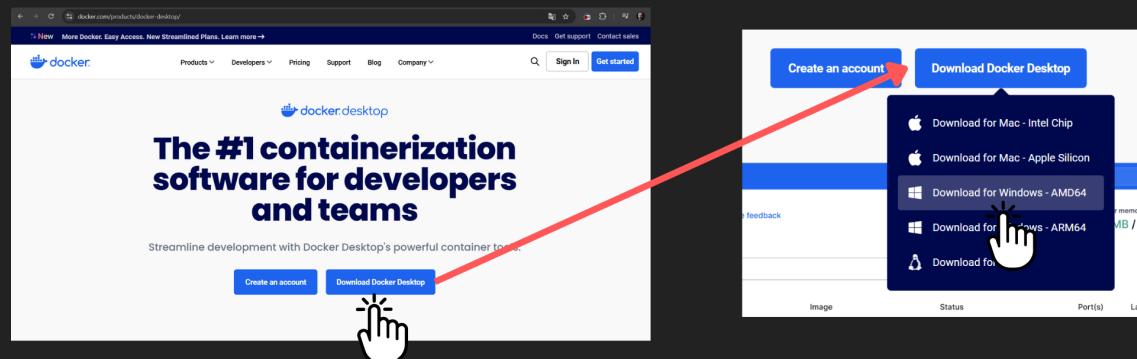
Passo 1: Baixar o Docker Desktop

Acesse o site oficial do Docker através do link <https://www.docker.com/products/docker-desktop> e baixe a versão do Docker Desktop compatível com o seu sistema operacional. Certifique-se de que

RAG - RETRIEVAL-AUGMENTED GENERATION

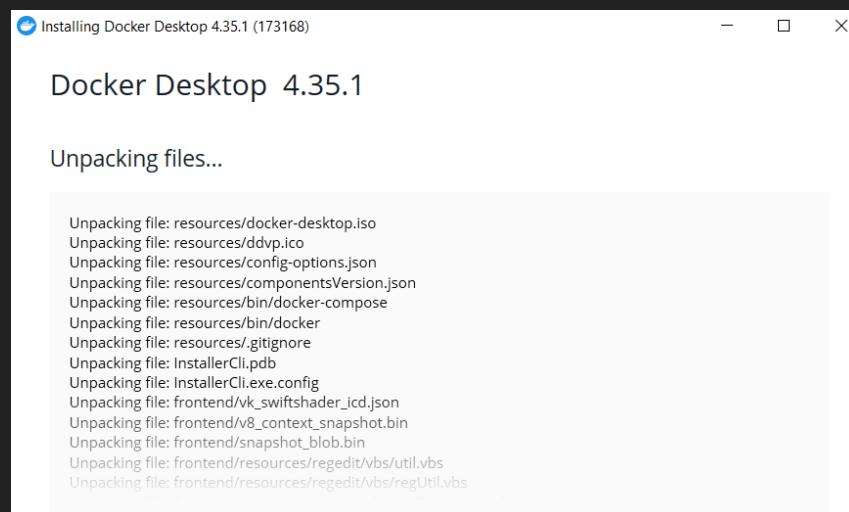
está utilizando o Windows 10 ou superior, com suporte para o WSL2 (Windows Subsystem for Linux), que é um requisito para rodar o Docker Desktop.

Baixando o Docker



Passo 2: Instalar o Docker Desktop

Após o download, execute o instalador e siga as instruções na tela. Finalize o processo e reinicie o computador.



RAG - RETRIEVAL-AUGMENTED GENERATION

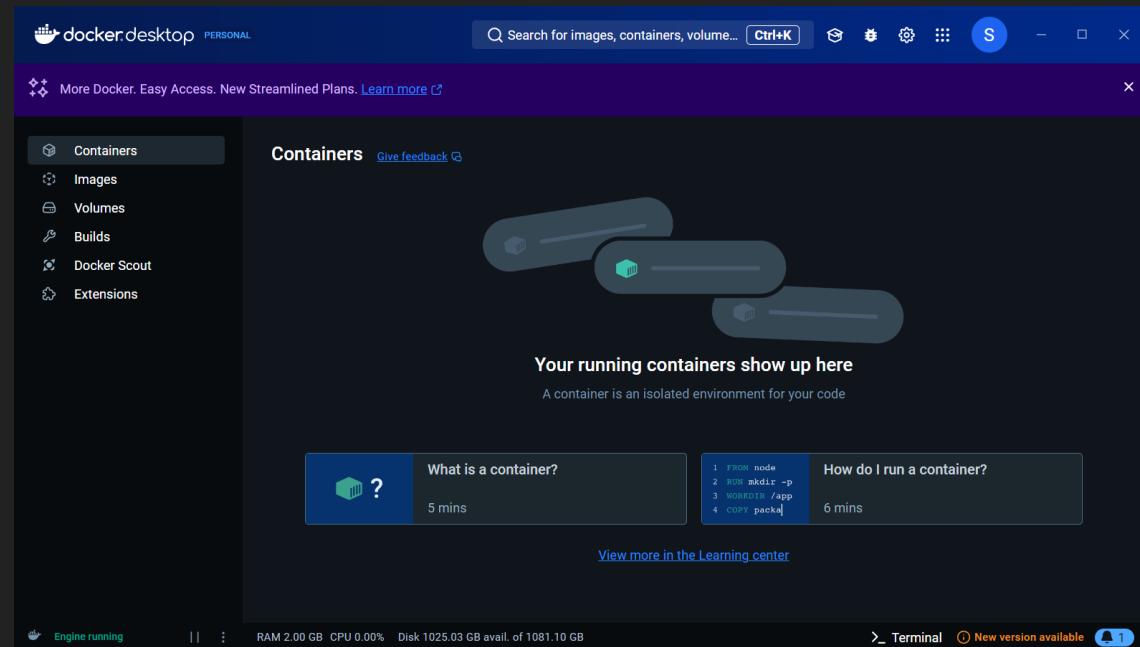
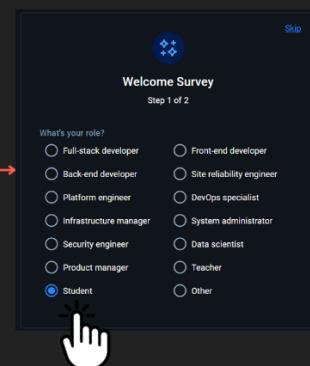
Passo 3: Iniciar o Docker Desktop

Com tudo configurado, abra o Docker Desktop. Na primeira execução, ele pode solicitar permissões administrativas para configurar o ambiente. Após a inicialização, verifique se o Docker está rodando corretamente observando o ícone na barra de tarefas.

Abra o docker



Responda o Survey



RAG - RETRIEVAL-AUGMENTED GENERATION

Passo 5: Testar a Instalação

Para garantir que o Docker está funcionando, abra o terminal ou o PowerShell e execute:

```
docker --version
```

```
C:\Users\sande>docker --version
Docker version 28.1.1, build 4eba377
```

Se o comando retornar a versão do Docker 28.1.1 ou superior instalada, tudo está pronto para usarmos o Docker no nosso projeto Redis!

3.3 INSTALANDO O REDIS

Para rodarmos o Redis de forma simples e organizada, utilizaremos o Docker Compose. O arquivo docker-compose.yml centraliza todas as configurações necessárias e permite que o serviço seja inicializado com apenas um comando.

Abra o arquivo docker-compose.yml na raiz do projeto com o conteúdo abaixo:

```
services:
  redis:
    image: redis:7-alpine
    container_name: redis-rag
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 3

volumes:
  redis_data:
    driver: local
```

Nesse arquivo, alguns pontos merecem destaque:

- Utilizamos a imagem **redis:7-alpine**, que é leve e otimizada para produção.
- O contêiner será criado com o nome **redis-rag**.

RAG - RETRIEVAL-AUGMENTED GENERATION

- A porta **6379** é exposta localmente, permitindo a comunicação com a aplicação.
- O volume **redis_data** garante persistência dos dados.
- O comando **redis-server --appendonly yes** ativa o modo de persistência em disco (AOF), evitando perda de dados em reinícios.
- A política de reinício **unless-stopped** assegura que o contêiner volte a rodar caso o sistema reinicie.
- O **healthcheck** executa periodicamente o comando **redis-cli ping** para verificar se o Redis está saudável.

Com o arquivo pronto, basta iniciar o Redis rodando no terminal (CMD) do Windows:

```
docker-compose up -d
```

Depois, confirme se o Redis está ativo com:

```
docker exec -it redis-rag redis-cli ping
```

Se tudo estiver correto, a resposta será 'PONG', indicando que o Redis está em pleno funcionamento e pronto para ser usado como memória no RAG.

3.4 CRIANDO A MÉMÓRIA

Para simplificar o processo de adoção da memória, criamos uma classe chamada *Memory* no arquivo 'memory.py'. Essa classe foi organizada em partes que facilitam tanto a inicialização quanto o gerenciamento do histórico de conversas no Redis. Vamos entender cada uma delas.

Primeiro, observe os imports que são utilizados. Execute o carregamento das bibliotecas necessárias: o pacote *redis* para conexão com o banco, o *json* para serialização dos dados, o *time* para controle de tempo em alguns trechos, e os tipos do *typing* para melhor documentação do código.

```
import redis
import json
import time
from typing import List, Dict, Any, Optional
```

Em seguida, definimos a classe *Memory*. Nela declaramos uma constante que representa o tempo padrão de expiração das conversas em segundos, equivalente a 24 horas. Esse tempo será utilizado caso não seja passado nenhum valor diferente na inicialização da classe.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
class Memory:  
    """  
        Uma classe para gerenciar o histórico de conversas de chat no Redis.  
    """  
  
    DEFAULT_EXPIRATION_SECONDS = 24 * 60 * 60 # 24 horas
```

Agora, implemente o método construtor. Aqui, criamos uma conexão com o Redis especificando o host, a porta e o banco a ser usado. Verifique se a conexão foi estabelecida utilizando o comando *ping*. Caso esteja tudo correto, exiba uma mensagem de confirmação. Caso contrário, dispare uma exceção interrompendo a execução. Note que também definimos a variável *expiration*, que vai armazenar o tempo de vida configurado para as chaves.

```
def __init__(self, expiration_seconds: int = DEFAULT_EXPIRATION_SECONDS):  
    redis_client = redis.Redis(host='localhost', port=6379, db=0,  
    decode_responses=True)  
    ping_result = redis_client.ping()  
    if ping_result:  
        print("Conectado ao Redis!\n")  
    else:  
        print("Falha na conexão com o Redis.")  
        raise Exception("Falha na conexão com o Redis.")  
  
    self.redis = redis_client  
    self.expiration = expiration_seconds  
    print(f"ChatManager inicializado. As conversas expirarão em {self.expiration}  
segundos.")
```

Na sequência, implemente um método auxiliar chamado *_get_key*. Ele serve para padronizar o formato das chaves que serão utilizadas no Redis. Escreva esse método de modo que, ao receber um *talk_id*, ele devolva uma chave de string com prefixo 'conversation:' seguido do identificador da conversa.

```
def _get_key(self, talk_id: str) -> str:  
    """Método auxiliar para gerar a chave padronizada do Redis."""  
    return f"conversation:{talk_id}"
```

Em seguida, utilize o método *add_memory*. Esse é o coração da classe, pois adiciona ou atualiza mensagens de uma conversa. Primeiro, obtenha a chave padronizada. Depois, busque no Redis se já existe algum histórico. Caso exista, carregue o conteúdo em formato JSON. Se não existir, initialize com uma lista vazia e informe que uma nova conversa foi criada. Após isso, insira a nova mensagem no início da lista, serialize novamente em JSON e salve de volta no Redis, aplicando a expiração

RAG - RETRIEVAL-AUGMENTED GENERATION

configurada.

```
def add_memory(self, talk_id: str, role: str, message: str) -> None:
    key = self._get_key(talk_id)
    existing_history_json = self.redis.get(key)

    if existing_history_json:
        history = json.loads(existing_history_json)
    else:
        history = []
        print(f"Nova conversa sendo criada para o usuário '{talk_id}'.")

    history.insert(0, {"role": role, "content": message})

    updated_history_json = json.dumps(history)
    self.redis.set(key, updated_history_json, ex=self.expiration)

    print(f"Memória de '{talk_id}' atualizada com a mensagem de '{role}'.")
```

Crie também o método `get_conversation`. Ele deve recuperar uma conversa completa a partir do `talk_id`. Para isso, obtenha a chave, faça a leitura no Redis e, caso o conteúdo exista, carregue o JSON para devolver uma lista de mensagens. Caso contrário, retorne `None`.

```
def get_conversation(self, talk_id: str) -> Optional[List[Dict[str, Any]]]:
    key = self._get_key(talk_id)
    history_json = self.redis.get(key)

    if history_json:
        return json.loads(history_json)

    return None
```

Implemente também o método `delete_conversation`. Ele deleta o histórico de uma conversa armazenada no Redis. Se a exclusão ocorrer, exiba uma mensagem confirmando. Se não houver conversa associada ao `talk_id`, mostre apenas uma informação de que nada foi removido.

```
def delete_conversation(self, talk_id: str) -> None:
    key = self._get_key(talk_id)
    if self.redis.delete(key) > 0:
        print(f"Conversa para o usuário '{talk_id}' foi deletada.")
    else:
        print(f" Nenhuma conversa para o usuário '{talk_id}' foi encontrada para deletar.")
```

Por fim, observe o bloco principal. Execute a criação de um objeto da classe `Memory`. Defina

RAG - RETRIEVAL-AUGMENTED GENERATION

um `TALK_ID` para identificar a conversa e, em seguida, delete qualquer histórico anterior com esse identificador. Adicione mensagens na sequência, simulando um diálogo entre usuário e sistema. Use `time.sleep` para criar pequenas pausas, e por último recupere e exiba a conversa final, incluindo o tempo de vida restante (`ttl`) no Redis.

```
if __name__ == "__main__":
    try:
        chat_manager = Memory()

        TALK_ID = "sandeco-upsert-test"

        chat_manager.delete_conversation(TALK_ID)
        print("-" * 30)

        chat_manager.add_memory(TALK_ID, "user", "Qual o primeiro livro de Isaac Asimov?")
        time.sleep(1)
        chat_manager.add_memory(TALK_ID, "system", "O primeiro romance publicado por Isaac Asimov foi 'Pebble in the Sky' (1950).")

        time.sleep(1)
        chat_manager.add_memory(TALK_ID, "user", "Obrigado!")

        conversa_final = chat_manager.get_conversation(TALK_ID)
        if conversa_final:
            key = chat_manager._get_key(TALK_ID)
            ttl = chat_manager.redis.ttl(key)
            print(f"\n--- Conversa Final Recuperada (expira em {ttl}s) ---")
            for msg in conversa_final:
                print(f" [{msg['role'].upper()}]: {msg['content']}")
            print("-" * 50)

    except redis.exceptions.ConnectionError as e:
        print(f"Falha na conexão com o Redis: {e}")
```

3.5 ADICIONANDO A MEMÓRIA AO RAG

Para que o RAG passe a trabalhar com memória, precisamos alterar o ponto onde ocorre a junção das informações. Essencialmente, a mudança está concentrada na classe `Augmentation`, ou seja, no **A do RAG**. Essa classe é responsável por reunir os dados recuperados (**Retriever**) e organizar o contexto que será enviado para a `Generation`, o **G do RAG**.

RAG - RETRIEVAL-AUGMENTED GENERATION

O que fazemos agora é acrescentar ao *Augmentation* a capacidade de lidar com memória. Isso significa que, além de juntar os *chunks* recuperados, o prompt também será complementado com o histórico das conversas anteriores. Dessa forma, o modelo não responde de maneira isolada a cada consulta, mas leva em conta a sequência do diálogo. Observe que o parâmetro *talk_id* identifica a conversa, permitindo que diferentes usuários tenham seus históricos separados.

Outro ponto importante é que, no código principal, precisamos enviar para a memória tudo aquilo que a *Generation* produzir. Isso garante que, ao gerar uma nova resposta, ela será armazenada e poderá ser recuperada em interações futuras. A memória então passa a registrar tanto a pergunta formatada pelo *Augmentation* quanto a resposta produzida pela *Generation*.

Veja a seguir como a classe *Augmentation* foi modificada. Compare com a versão anterior, onde apenas o prompt era construído. Agora, além de gerar o prompt, adicionamos métodos para limpar a memória, registrar novas interações e recuperar o histórico:

```
from memory_rag import MemoryRAG

class Augmentation:
    def __init__(self, talk_id):
        self.talk_id = talk_id
        self.memory_rag = MemoryRAG()
        self.prompt = ""

    def generate_prompt(self, query_text, chunks):
        separador = "\n\n-----\n\n"
        chunks_formatados = f"Conhecimento\n-----\n\n{separador}.join(chunks)"

        self.prompt = f"""Responda em pt-br e em markdown, a query do usuário
delimitada por <query>
usando apenas o conhecimento dos chunks delimitados por <chunks>
e tenha em mente o histórico das conversas anteriores delimitado por <
histórico>.
Combine as informações para responder a query de forma unificada. A
prioridade
das informações são: query=1, chunks=2, histórico=3.

Se por acaso o conhecimento não for suficiente para responder a query,
responda apenas que não temos conhecimento suficiente para responder
a Pergunta.

<chunks>
{chunks_formatados}
</chunks>

<query>
{query_text}
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
</query>

<historico>
{self.memory_rag.get_conversation(self.talk_id)}
</historico>
"""

return self.prompt

def clear_memory(self):
    self.memory_rag.delete_conversation(self.talk_id)

def add_memory(self, llm_response):
    try:
        self.memory_rag.add_memory(self.talk_id, "user", self.prompt)
        self.memory_rag.add_memory(self.talk_id, "system", llm_response)
        return True
    except Exception as e:
        print(f"Erro ao adicionar memória: {str(e)}")
        return False

def get_conversation(self):
    return self.memory_rag.get_conversation(self.talk_id)
```

Adaptando o código principal

Agora que já estruturamos as classes individuais do RAG com memória, podemos observar o código principal que conecta todas elas em um fluxo contínuo. Esse trecho é responsável por coordenar a **recuperação**, a **montagem do prompt com memória** e a **geração da resposta**. Vamos entender como cada parte funciona.

Primeiro, faça as importações das três classes centrais: *Retriever*, *Augmentation* e *Generation*. São elas que representam, respectivamente, o R, o A e o G do RAG.

```
# arquivo main.py do projeto

from retriever import Retriever
from augmentation import Augmentation
from generation import Generation
```

Em seguida, defina um identificador de conversa chamado *TALK_ID*. Esse valor será usado pela memória para associar as perguntas e respostas a uma mesma sessão. Depois disso, inicialize as instâncias de cada classe: o *Retriever* apontando para a coleção de dados a ser consultada, o *Augmentation* recebendo o *TALK_ID*, e o *Generation* configurado com o modelo de linguagem escolhido.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
TALK_ID = "sandeco-chat-001"

retriever = Retriever(collection_name="synthetic_dataset_papers")
augmentation = Augmentation(talk_id=TALK_ID)
generation = Generation(model="gemini-2.5-flash")
```

Agora, construa o laço principal da aplicação. Dentro do *while True*, capture a entrada do usuário simulando um chat. Se a pessoa digitar a palavra 'sair', o sistema interrompe a execução e encerra o loop.

```
while True:
    user_query = input("Sua Pergunta: ")

    if user_query.lower() == 'sair':
        print("Até a próxima!")
        break
```

Caso a entrada não seja 'sair', execute o fluxo do RAG. Primeiro, use o *Retriever* para buscar os *chunks* relevantes. Em seguida, utilize o *Augmentation* para gerar o prompt que combina a query, os chunks recuperados e o histórico da memória. Depois, chame o *Generation* para produzir a resposta final a partir desse prompt. Note que logo após gerar a resposta, o sistema a envia de volta para a memória utilizando *augmentation.add_memory(response)*. Por fim, exiba a resposta na tela.

```
chunks = retriever.search(user_query, n_results=10, show_metadata=False)
prompt = augmentation.generate_prompt(user_query, chunks)
response = generation.generate(prompt)

augmentation.add_memory(response)

print(response)
```

Esse fluxo garante que a cada interação o modelo consulte informações externas, construa um contexto atualizado com o histórico da conversa e registre a nova resposta na memória. Assim, o RAG deixa de ser uma sequência de chamadas independentes e passa a manter coerência ao longo de todo o diálogo.

Com isso, criamos um fluxo contínuo: o **Retriever** busca a informação, o **Augmentation** constrói o prompt incluindo o histórico, o **Generation** gera a resposta, e finalmente essa resposta é enviada de volta para a memória. Dessa forma, o RAG não apenas responde, mas passa a acompanhar e registrar toda a interação. Agora é só executar a main.py e conversar com o RAG.

3.6 EXERCÍCIOS

1. Explique, com suas próprias palavras, qual a diferença entre um RAG tradicional e um RAG com memória. Use a analogia do professor apresentada na abertura do capítulo como apoio.
2. Analise o papel da classe *Memory*. O que acontece se o método *add_memory* não for chamado após cada resposta gerada pelo modelo?
3. No fluxo principal, identifique em que ponto a resposta do modelo é enviada de volta para a memória. Por que esse passo é fundamental para manter a coerência do diálogo?
4. Modifique o código da classe *Augmentation* para que apenas as últimas três interações sejam incluídas no histórico ao gerar o prompt. Qual seria o impacto dessa modificação no comportamento do RAG?
5. Crie um teste em que duas sessões de conversa (*TALK_ID* diferentes) sejam executadas simultaneamente. Mostre como o sistema mantém o histórico separado para cada sessão.
6. Reflita sobre cenários em que a expiração das conversas seja vantajosa. Em quais aplicações faz sentido que as memórias se apaguem após um tempo definido? E em quais não?
7. No código principal, troque a condição de parada de 'sair' para outro comando de sua escolha. Execute o programa e verifique se a lógica continua funcionando corretamente.
8. Observe a ordem em que as mensagens são armazenadas na memória. Justifique por que a decisão de inserir a mensagem no início da lista é importante para a recuperação do histórico.
9. Implemente uma função que mostre, a qualquer momento, o histórico completo da conversa atual. Utilize os métodos já existentes na classe *Memory*.
10. Imagine que você está aplicando esse RAG com memória em um ambiente educacional. Propõa um exemplo de como essa funcionalidade poderia melhorar a experiência do estudante em comparação a um RAG sem memória.

CAPÍTULO 4

RAG Autônomo com Agentic RAG

Imagine que agora você tem várias bases de dados vetoriais espalhadas, cada uma especializada em um domínio específico: direito, saúde, engenharia, literatura ou mesmo datasets sintéticos para experimentos. O desafio é que o usuário faz uma consulta e o RAG precisa saber para onde direcionar essa pergunta, escolhendo o fluxo mais adequado para encontrar a resposta correta. É aqui que entra o AgenticRAG.



Figura 4.1: AgenticRAG - Direcionador de Fluxos

Pense no AgenticRAG como um guarda de trânsito cognitivo. Ele não dirige os carros, mas observa o fluxo e decide quem deve seguir, quem deve parar e qual caminho cada veículo deve tomar para chegar ao destino certo. Da mesma forma, o AgenticRAG direciona a consulta do usuário para a base de dados mais apropriada, organiza o fluxo de recuperação e garante que a resposta volte de forma coerente e contextualizada.

RAG - RETRIEVAL-AUGMENTED GENERATION

Enquanto um RAG tradicional consulta uma fonte única ou depende de configuração estática, o AgenticRAG atua com autonomia: ele avalia a pergunta, decide qual vetorstore ou qual ferramenta acionar e orquestra as etapas de raciocínio. Em vez de depender apenas de regras pré-programadas, o agente opera de forma adaptativa, mudando a estratégia conforme a natureza da consulta.

Ao longo deste capítulo, vamos detalhar como projetar essa arquitetura, explorando a lógica de roteamento, os blocos que conferem autonomia ao agente e exemplos práticos de como o AgenticRAG pode se tornar o 'guarda de trânsito' do conhecimento. Dessa forma, o RAG deixa de ser apenas um mecanismo de busca e passa a agir como um orquestrador inteligente, capaz de conduzir cada consulta para o fluxo mais eficiente.

Baixando o Projeto

Para começar, faça o download do nosso projeto no link abaixo:

Link do projeto:

<https://bit.ly/sandeco-agentic-rag>

4.1 AGENTICRAG

Com o crescimento do uso de Agentes de IA, surge a necessidade de integrar essa autonomia ao próprio RAG, criando o que chamamos de **AgenticRAG**. Nesse modelo, o agente não é apenas um componente auxiliar, mas a peça central que direciona o fluxo de informações de acordo com a consulta feita pelo usuário. Ele atua como um orquestrador, capaz de decidir de forma inteligente para qual base de dados vetorial encaminhar a pergunta, quando acionar embeddings adicionais e como estruturar o processo de recuperação antes de entregar o resultado para a geração. Essa camada de decisão transforma o RAG em um sistema mais flexível e adaptativo, pronto para lidar com cenários complexos onde múltiplas fontes e estratégias podem estar envolvidas.

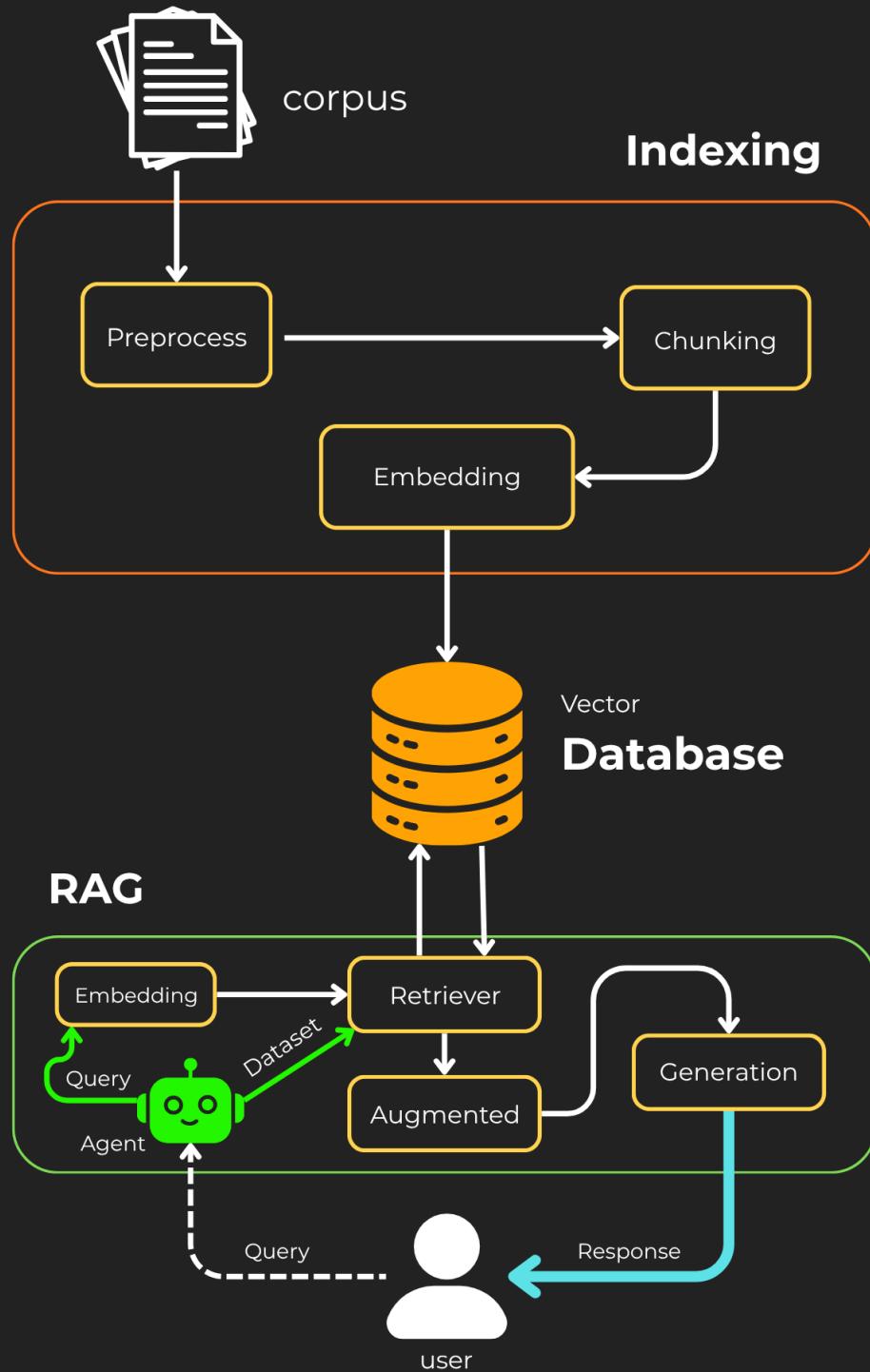
Ao adotar essa abordagem, o RAG deixa de ser apenas uma sequência linear de etapas e passa a se comportar como um sistema dinâmico, no qual o agente avalia continuamente a melhor rota para cada consulta. Pense no AgenticRAG como um guarda de trânsito: ele não dirige os carros, mas observa o tráfego, interpreta a situação e indica o caminho correto. Assim, em vez de enviar todas as queries para a mesma direção, o agente distribui as consultas para o fluxo mais apropriado, garantindo que a resposta seja construída de maneira mais eficiente e contextualizada.

Na Figura do fluxo RAG, você pode observar claramente essa mudança. O usuário envia a query, mas ela não segue diretamente para o **Retriever**. O primeiro ponto de contato é o **Agente**, destacado em verde, que assume o papel de orquestrador. Veja como o fluxo é conduzido: o agente recebe a

RAG - RETRIEVAL-AUGMENTED GENERATION

consulta, pode acionar o módulo de **Embedding** para enriquecer a representação semântica e, em seguida, decide qual dataset será utilizado.

RAG - RETRIEVAL-AUGMENTED GENERATION



Note que o **Augmented** continua a organizar o contexto antes de passá-lo para a **Generation**, mas agora todo esse processo é enriquecido pela decisão inicial do agente. A lógica do fluxo se torna: usuário envia a query → agente interpreta e direciona → embeddings e datasets são acionados → recuperação é feita → augmentation organiza → geração devolve a resposta. Essa mudança estrutural é o que diferencia o AgenticRAG, tornando-o mais robusto e inteligente no direcionamento das consultas.

4.2 CLASSE DE REGISTRO DE DATASETS

A primeira coisa que precisamos fazer é criar um registrador de datasets. Para criar um novo dataset (collection no ChromaDB), você deve usar a classe do código 'semantic_encoder.py', explicada na Seção 2.8 do Capítulo 2.

Após a criação, registre o novo dataset na classe **Datasets**, que terá a responsabilidade de armazenar todas as coleções de dados disponíveis para o Agentic RAG. Essa classe será usada para informar ao nosso agente de roteamento de fluxo quais datasets e quais assuntos estão disponíveis.

Observe que, no construtor da classe **Datasets**, representado pelo método `__init__`, é inicializada uma lista chamada **datasets**. Nela, escreva cada base de conhecimento como um dicionário, definindo três atributos fundamentais: o nome interno do dataset, a descrição que explica em que contexto ele deve ser pelo agente e o **locale**, que indica a língua predominante do material.

O primeiro registro é o dataset chamado **synthetic_dataset_papers**, que possui uma descrição clara de que deve ser usado quando a consulta envolver a construção, utilização ou detecção com dados sintéticos. Seu locale é definido como 'en', reforçando que esse conjunto está em inglês. Em seguida, temos o segundo registro: o dataset chamado **direito_constitucional**, que será escolhido sempre que a consulta envolver temas jurídicos, leis, processos ou jurisprudência. Nesse caso, o locale é 'pt-br', garantindo que a base de dados esteja em português e seja adequada ao contexto legal brasileiro.

Informar o **locale** do dataset pode ajudar a traduzir a query para a língua predominante nos chunks, gerando uma consulta mais efetiva e possibilitando ao usuário flexibilidade para escrever a query em sua língua nativa.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
class Datasets:  
    def __init__(self):  
        self.datasets = [  
            {"dataset": "synthetic_dataset_papers",  
             "description": "A construção, utilização ou detecção usando  
             datasets sintéticos",  
             "locale": "en"},  
  
            {"dataset": "direito_constitucional",  
             "description": "Se a consulta envolver direito, leis,  
             processos ou jurisprudência",  
             "locale": "pt-br"}  
        ]
```

Agora vamos começar a criar nossos agentes. Existem duas formas principais de fazer isso. A primeira consiste em enviar a consulta diretamente para a LLM de sua escolha, utilizando a API correspondente. A segunda forma é criar o agente por meio de um framework, como CrewAI, ADK ou outros semelhantes. Independentemente da forma escolhida, todos os agentes que você desenvolver devem estender a classe **AgentRAGAbstract**, o que facilitará a construção e a organização do fluxo do RAG.

4.3 AGENTE ABSTRATO

Aqui vamos abordar um conceito diferenciado na criação de agentes: a definição de um agente abstrato que servirá como template para os demais. Essencialmente, esse agente abstrato é implementado como uma classe abstrata.

Pense em uma classe abstrata como a planta de um prédio que ainda não foi construído. A planta define quantos andares existirão, onde estarão os elevadores, as escadas e a estrutura básica que todos os andares devem seguir. Porém, cada andar só ganha vida quando os engenheiros e arquitetos decidem como será decorado, quais materiais serão usados e quais salas existirão. Assim também funciona a classe abstrata: ela define a estrutura mínima que todas as subclasses precisam respeitar, mas deixa a cargo de cada uma a implementação específica dos detalhes.

Uma classe abstrata é um recurso essencial da programação orientada a objetos quando se deseja criar um molde para outras classes. Execute a criação de uma classe abstrata sempre que precisar definir uma estrutura mínima obrigatória que outras classes deverão seguir. Ela não pode ser instanciada diretamente, mas serve como um contrato que obriga suas subclasses a implementar determinados métodos. Isso garante que todas as classes derivadas mantenham um padrão de comportamento, mesmo que internamente a implementação varie.

RAG - RETRIEVAL-AUGMENTED GENERATION

Além disso, utilize classes abstratas quando quiser organizar hierarquias de objetos que compartilham características comuns, mas que ainda exigem especialização em pontos específicos. Ao declarar métodos abstratos, você determina que qualquer classe filha seja obrigada a escrever sua própria versão desses métodos. Isso evita inconsistências e assegura que o fluxo definido como regra seja respeitado em toda a aplicação, mantendo uniformidade no projeto.

No código da classe **AgentRAGAbstract**, observe que ela herda de **ABC**, que é a classe base para construção de classes abstratas no Python. Dentro do construtor, execute a inicialização da variável **datasets**, que recebe a instância da classe **Datasets**. Essa escolha garante que qualquer agente que estenda **AgentRAGAbstract** já terá acesso imediato ao registrador de datasets, sem precisar reimplementar essa lógica a cada nova classe. Com isso, você centraliza e padroniza o acesso às bases de dados disponíveis no RAG.

Por fim, repare no método **query**, declarado com o decorador **@abstractmethod**. Esse detalhe indica que não existe implementação padrão para esse método, e que todas as subclasses da **AgentRAGAbstract** deverão obrigatoriamente escrevê-lo. Faça a implementação de cada agente, definindo como ele receberá a consulta, processará a query e interagirá com os datasets. Essa estratégia garante que todos os agentes mantenham uma interface comum, facilitando a integração no fluxo do RAG e a consistência entre diferentes tipos de agentes. Você verá que todos os nossos agentes terão esse método **query**.

```
#agentRAGAbstract.py dentro da pasta agentRAG

from abc import ABC, abstractmethod
from datasets import Datasets


class AgentRAGAbstract(ABC):

    def __init__(self):
        self.datasets = Datasets()

    @abstractmethod
    def query(self, query):
        pass
```

4.4 AGENTE COM API DA LLM

Como já mencionei anteriormente, é possível criar agentes utilizando frameworks ou chamando diretamente a API de uma LLM. No entanto, adoto uma regra prática: se o projeto exige apenas um único agente que somente acessa uma LLM, não faz sentido carregar todo um framework de agentes na memória apenas para realizar uma chamada simples à LLM. Nesses casos, prefira a chamada direta, pois ela torna o fluxo mais leve, eficiente e sem sobrecarga desnecessária. Por isso, vamos criar aqui nesta seção um código de agente simples sem framework e na próxima seção usaremos um framework para exemplificar, ok?

Gemini Agent

Vamos criar a classe **AgentRAGemini** no arquivo `agenticRAGemini.py`. Veja que a classe estende **AgentRAGAbstract**, portanto, obrigatoriamente ela deve implementar o método **query**. A função desse agente é interagir com a API do Gemini, utilizando o modelo especificado, para escolher dinamicamente qual dataset será usado conforme a consulta recebida do usuário.

```
#instale o Google Genai
uv add google-genai
```

A primeira parte do código trata das importações. Execute a importação dos módulos essenciais como **os**, que será usado para acessar variáveis de ambiente, o pacote **genai** do Google, responsável pela interação com o modelo Gemini, e o módulo **json**, necessário para manipular a resposta que virá no formato JSON. Também faça a importação da classe **AgentRAGAbstract**, que será estendida aqui para manter a padronização da arquitetura.

```
import os
from google import genai
import json

from .agentRAGAbstract import AgentRAGAbstract
```

Em seguida, observe a definição da classe **AgentRAGemini**. No construtor, utilize o **super()** para herdar as inicializações da classe abstrata. Logo depois, configure a chave da API do Gemini com a variável de ambiente **GEMINI_API_KEY** e defina o modelo a ser utilizado, neste caso **gemini-2.5-flash**. Execute essa configuração inicial para que qualquer instância criada do agente já esteja preparada para se comunicar com a API do Gemini.

RAG - RETRIEVAL-AUGMENTED GENERATION

```
class AgentRAGemini(AgentRAGAbstract):

    def __init__(self):
        super().__init__()

    # Configurar API key do Gemini
    self.model = "gemini-2.5-flash"
    self.client = genai.Client(api_key=os.getenv('GEMINI_API_KEY'))
```

A terceira parte do código é o método `create_prompt`, responsável por construir dinamicamente o prompt que será enviado ao modelo. Escreva aqui a lógica que percorre todos os datasets registrados e formata suas descrições, nomes e locales dentro do texto. Note que o prompt exige que a saída seja estritamente em formato JSON, contendo os atributos `dataset_name`, `locale` e `query`. O detalhe importante é que a query deve ser traduzida para o idioma definido pelo locale escolhido, garantindo consistência na comunicação com a base. Repare também nas instruções imperativas dentro do prompt, que restringem a saída apenas ao JSON, sem justificativas ou textos adicionais.

```
def create_prompt(self, query):
    # Construir descrição dos datasets
    prompt = f'''
        Sua missão:
        Com base na solicitação do usuário "{query}" escolha
        somente um dataset é mais apropriado da lista abaixo:
        ...

    for dataset in self.datasets.datasets:
        prompt += f"- {dataset['description']} escreva ->
            {dataset['dataset']}. Dataset
            Locale: {dataset['locale']}"

    prompt += r"""\\n
    Eu quero como saída um json com as
    seguintes informações do dataset
    escolhido:
    - dataset_name:
    - locale:
    - query:

    A query deve ser traduzida para o locale do dataset escolhido.
    Não adicione explicações, justificativas ou qualquer outro texto.
    Não adicione caracteres especiais ou caracteres de escape.
    Não adicione '''json ou ''' envolvendo o json de resposta,
    se vc colocar isso você será demitido.
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
Exemplo de saída:  
{"dataset_name": "dataset", "locale": "en", "query": "This is my question"}  
  
É IMPERATIVO:  
Não escreva nada além do json de resposta.  
"""  
  
return prompt
```

Por fim, analise o método `query`, que é a implementação obrigatória da classe abstrata. Execute a criação do prompt usando o método anterior, depois envie esse prompt ao modelo Gemini por meio da função `generate_content`. O retorno da API chega como texto, portanto use o `json.loads` para convertê-lo em um dicionário Python. Ao final, retorne esse dicionário, que conterá as informações do dataset escolhido e a query traduzida. Essa etapa completa o ciclo: da consulta do usuário até a seleção automática do dataset adequado.

```
def query(self, query):  
    """  
    Implementa o método abstrato query para conectar com o Gemini  
    """  
    # Criar prompt com contexto  
    prompt = self.create_prompt(query)  
  
    response = self.client.models.generate_content(  
        model=f"models/{self.model}",  
        contents=[prompt]  
    )  
  
    response = json.loads(response.text)  
    return response
```

Viu como é simples nosso Agente com conexão direta com o Gemini?

Ah! Uma coisa importante, não esqueça de colocar a sua GEMINI_API_KEY no arquivo `.env`.

```
GEMINI_API_KEY="sua_chave_aqui"
```

4.5 AGENTIC RAG COM CREWAI

Já exploramos em profundidade o uso do **CrewAI** nos meus dois livros *Agentes Inteligentes Vol. 1* e *Vol. 2*. Por isso, vamos direto ao ponto na criação de um Agente RAG utilizando esse framework.

Primeiro devemos instalar o CrewAI no nosso projeto:

```
uv add crewai
```

Vamos criar a classe **AgentRAGCrewAI** no arquivo `agenticRAGcrewAI.py`. Veja que a classe também estende **AgentRAGAbstract** e, portanto, obrigatoriamente deve implementar o método **query**. A diferença aqui é que esse agente não utiliza a API do Gemini, mas sim o framework CrewAI para organizar o fluxo de decisão sobre qual dataset será usado. Essa integração permite que o agente use o conceito de equipes (**Crew**), compostas por agentes e tarefas, para gerenciar de forma mais estruturada o processo de escolha do dataset adequado.

Na primeira parte do código, execute a importação dos módulos necessários. O pacote **dotenv** é carregado para acessar variáveis de ambiente. Em seguida, importamos as classes **Crew**, **Agent**, **Task** e **Process** da biblioteca CrewAI, que são fundamentais para construir o fluxo. Também fazemos a importação da classe **Datasets** e de **AgentRAGAbstract**, que será estendida. Por fim, a biblioteca **json** será utilizada para manipular a saída do modelo. Logo abaixo, a função **load_dotenv()** é executada para carregar as variáveis de ambiente definidas em um arquivo `.env`.

```
from dotenv import load_dotenv
from crewai import Crew, Agent, Task, Process
from datasets import Datasets
from .agentRAGAbstract import AgentRAGAbstract

import json

load_dotenv()
```

Na sequência, definimos a classe **AgentRAGCrewAI**. Dentro do construtor `__init__`, atribuímos ao atributo **llm** o modelo que será utilizado, no caso, **gpt-5-mini**.

```
class AgentRAGCrewAI(AgentRAGAbstract):

    def __init__(self):
        self.llm = "gpt-5-mini"
```

RAG - RETRIEVAL-AUGMENTED GENERATION

A parte seguinte é o método `create_crew`, responsável por construir a equipe (`Crew`) que fará a decisão sobre qual dataset utilizar. Primeiro, criamos um agente chamado `router`, com um papel e objetivo claros: decidir, com base na solicitação do usuário, qual base vetorial deve ser usada. Ele recebe também uma `backstory`, que fornece contexto ao agente sobre sua função, e parâmetros como `reasoning` e `verbose`, que controlam o nível de raciocínio e detalhamento da execução. Em seguida, é construída a descrição da tarefa usando o método `create_description`, que gera um enunciado detalhado para o agente. Essa descrição é usada para compor um objeto `Task`, que define o nome da tarefa, o agente responsável, a descrição e o formato esperado da saída, que deve ser um JSON. Por fim, toda essa estrutura é organizada em uma instância de `Crew`, onde a tarefa é registrada e o processo de execução é configurado como sequencial.

```
def create_crew(self, query):

    router = Agent(
        role="Agente Roteador de RAG",
        goal=(
            "Decidir, com base na solicitação do usuário, "
            "qual base vetorial de conhecimento deve ser usada "
            "para responder da forma mais adequada."
        ),
        backstory=(
            "Você é um especialista em recuperação de informação e agente RAG. "
            "Sua função é interpretar a solicitação e determinar "
            "de forma precisa qual "
            "dataset de conhecimento deve ser consultado. "
        ),
        reasoning=True,
        verbose=True,
        llm=self.llm
    )

    description, frase_dataset_name = self.create_description(query)

    task = Task(
        name="Decidir dataset RAG",
        agent=router,
        description=description,
        expected_output="Um json com as informações do dataset escolhido e com a
query traduzida para o locale do dataset escolhido",
        llm=self.llm
    )

    self.crew = Crew(
        agents=[router],
        tasks=[task],
        process=Process.sequential
```

RAG - RETRIEVAL-AUGMENTED GENERATION

```
)
```

Agora observe o método **create_description**. Ele monta a descrição que será usada pela tarefa, seguindo a mesma lógica explicada anteriormente no AgentRAGemini. O texto apresenta ao agente sua missão, lista todos os datasets disponíveis com suas descrições, nomes e locales, e instrui claramente que a saída deve ser apenas um JSON com os campos **dataset_name**, **locale** e **query**. O detalhe importante é que a query deve ser traduzida para o locale do dataset escolhido. As instruções também reforçam restrições rígidas, como não adicionar explicações extras ou caracteres especiais, garantindo que a saída seja limpa e padronizada.

```
def create_description(self, query):
    # Construir descrição dos datasets
    description = f'''
        Sua missão:
        Com base na solicitação do usuário "{query}" escolha
        somente um dataset é mais apropriado da lista abaixo:
        '''

    for dataset in self.datasets.datasets:
        description += f"- {dataset['description']} escreva -> {dataset['dataset']}
        ]. Dataset Locale: {dataset['locale']}"

    description += r"""\\n
    Eu quero como saída um json com as seguintes informações do dataset escolhido
    :
    - dataset_name:
    - locale:
    - query:

    A query deve ser traduzida para o locale do dataset escolhido.
    Não adicione explicações, justificativas ou qualquer outro texto.
    Não adicione caracteres especiais ou caracteres de escape.
    Não adicione '''json ou ''' envolvendo o json de resposta, se vc colocar isso
    vc será demitido.

    Exemplo de saída:
    {"dataset_name": "dataset", "locale": "en", "query": "This is my question"}

    É IMPERATIVO:
    Não escreva nada além do json de resposta.
    """
    """

    return description
```

Por fim, temos os métodos **kickoff** e **query**. O método **kickoff** é responsável por iniciar a execução

RAG - RETRIEVAL-AUGMENTED GENERATION

da Crew criada anteriormente, retornando o resultado produzido pelo agente. Já o método `query` chama `kickoff`, recebe a resposta e, em seguida, converte o conteúdo bruto para o formato JSON usando a função `json.dumps`. Esse processamento final garante que a resposta esteja no formato correto, pronta para ser interpretada pelo restante do fluxo do RAG. Dessa forma, completamos a implementação de um agente baseado no CrewAI, totalmente integrado ao padrão definido pela classe abstrata.

```
def kickoff(self, query):
    self.create_crew(query)

    response = self.crew.kickoff()
    return response

def query(self, query):
    response = self.kickoff(query)

    # converta o json em string para que possa ser convertido em um objeto json
    response.raw = json.dumps(response.raw)

    return response.raw
```

4.6 EXECUTANDO O AGENTIC RAG

Agora vamos criar uma "main" para executar o nosso Agentic RAG. Essa será a parte responsável por orquestrar todo o fluxo: receber a consulta do usuário, decidir qual dataset utilizar, recuperar os documentos relevantes, montar o prompt e, por fim, gerar a resposta. É aqui que todos os componentes que já desenvolvemos anteriormente se conectam em um pipeline contínuo.

Na primeira parte do código, execute a importação das classes que serão utilizadas. Três delas são fundamentais para compor o fluxo tradicional do RAG: `Retriever`, `Augmentation` e `Generation`. Além disso, importamos o `AgentRAGemini`, que será o agente responsável por decidir qual dataset utilizar com base na query recebida do usuário. Esse conjunto de importações garante que todos os módulos estejam disponíveis para a execução.

```
from retriever import Retriever
from augmentation import Augmentation
from generation import Generation

from agentRAG.agenticRAGemini import AgentRAGemini
```

Em seguida, observe a definição da query do usuário. Aqui, a pergunta "O que é direito constitui-

RAG - RETRIEVAL-AUGMENTED GENERATION

cional fala do abandono afetivo?" será usada como exemplo. Execute a criação de uma instância de **AgentRAGemini** e chame o método **query**, que retorna um dicionário em formato JSON. Esse dicionário contém, entre outras informações, o campo **dataset_name**, indicando qual base vetorial foi escolhida pelo agente para responder à consulta. Armazene essa informação na variável **dataset_escolhido** para utilizá-la nos próximos passos.

```
query = "O que é direito constitucional fala do abandono afetivo?"
dataset = AgentRAGemini().query(query)

dataset_escolhido = dataset['dataset_name']
```

Na terceira parte, initialize os três módulos centrais do RAG. O **Retriever** será criado recebendo como parâmetro a coleção correspondente ao dataset escolhido. O **Augmentation** é instanciado em seguida, sendo responsável por montar o prompt final. Já o **Generation** é inicializado especificando o modelo a ser usado, neste caso **gemini-2.5-flash**. Essa etapa configura o fluxo para que cada componente saiba exatamente qual papel desempenhar.

```
retriever = Retriever(collection_name=dataset_escolhido)
augmentation = Augmentation()
generation = Generation(model="gemini-2.5-flash")
```

Agora execute a recuperação e geração de resposta. Use o **Retriever** para buscar documentos relevantes à query, pedindo dez resultados e omitindo metadados. Em seguida, passe esses resultados ao método **generate_prompt** do **Augmentation**, que cria um prompt estruturado combinando a query e os chunks recuperados. Esse prompt é então enviado ao módulo **Generation**, que gera a resposta final. Por último, exiba o resultado com o comando **print**, completando o ciclo do Agentic RAG em funcionamento.

```
# Buscar documentos
chunks = retriever.search(query, n_results=10, show_metadata=False)
prompt = augmentation.generate_prompt(query, chunks)

# Gerar resposta
response = generation.generate(prompt)

print(response)
```

Se você quiser saber algo sobre os datasets sintéticos, agora você pode escrever a query em "pt-br"

```
query = "O que é dataset sintético?"
dataset = AgentRAGemini().query(query)
```

```
dataset_escolhido = dataset['dataset_name']
```

E a saída será um texto voltado para esse assunto. Legal, né?

4.7 EXERCÍCIOS

1. Explique, com suas próprias palavras, qual a principal diferença entre um RAG tradicional e o AgenticRAG.
2. Use a analogia do guarda de trânsito apresentada no capítulo para justificar como o AgenticRAG atua no direcionamento do fluxo de informações.
3. Analise a importância da classe **Datasets**. Por que ela é fundamental para que o agente consiga decidir qual base vetorial deve ser consultada?
4. Imagine que você deseja adicionar um novo dataset sobre saúde. Escreva como esse dataset deveria ser registrado na classe **Datasets**, incluindo o nome, a descrição e o locale.
5. Reflita sobre o papel da classe **AgentRAGAbstract**. Por que ela é declarada como uma classe abstrata e o que isso garante na arquitetura do RAG?
6. Explique como o método **create_prompt** da classe **AgentRAGGemini** contribui para que a resposta esteja sempre padronizada no formato JSON.
7. No fluxo do AgenticRAG, em qual momento ocorre a decisão sobre qual dataset será usado? Descreva passo a passo como essa escolha é realizada.
8. Analise as vantagens de usar o framework **CrewAI** para criar agentes, em comparação com a chamada direta à API de uma LLM.
9. No código da "main", identifique o papel de cada uma das três classes principais: **Retriever**, **Augmentation** e **Generation**. Explique como elas se complementam no fluxo.
10. Proponha um cenário prático em que o AgenticRAG poderia ser aplicado em sua área de interesse, explicando como o agente escolheria o dataset adequado e quais etapas do fluxo seriam mais relevantes.
11. Essa agora é **hard**. E se você quisesse usar uma ferramenta de busca na web ao invés de usar os chunks do **Retriever** para enviar ao **Generation**, o que você faria?