Capítulo 17 – RAG Parte 1 (A Magia Feita à Mão em Python)

Aléssio Miranda Jr.

Outubro - 2026/2

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

O que você verá neste capítulo

Retrieval-Augmented Generation (RAG) é o padrão arquitetural que resolveu o maior problema dos LLMs: a **alucinação**. Neste capítulo, construiremos do zero, em Python puro, um sistema RAG completo que combina a busca semântica do Arco 2 com a geração de texto do Arco 3. Formalizaremos o fluxo Retrieve → Augment → Generate, introduziremos as métricas de avaliação de RAG (Faithfulness, Answer Relevancy), e discutiremos as armadilhas que destroem a qualidade em produção.

1 O Problema da Alucinação

Os LLMs são treinados em bilhões de textos da internet. Eles possuem um conhecimento vasto, porém **estático** (congelado na data de corte do treinamento) e **não citável** (o modelo não consegue indicar a fonte de cada afirmação). Quando confrontado com uma pergunta que excede seu treinamento ou que exige precisão factual, o LLM faz algo perigoso: ele **inventa** uma resposta plausível com confiança absoluta.

Esse fenômeno é chamado de **Alucinação** (*Hallucination*). O modelo gera texto gramaticalmente perfeito, estilisticamente convincente, mas factualmente **falso**.

△ Importante: title=Exemplos Reais de Alucinação

- Advogados nos EUA citaram jurisprudência gerada pelo ChatGPT que **não existia**. Foram multados pelo juiz.
- Sistemas médicos baseados em LLMs recomendaram dosagens de medicamentos com valores inventados.
- Chatbots de atendimento ao cliente prometeram descontos e políticas de reembolso que a empresa **nunca ofereceu**.

2 A Arquitetura RAG

O RAG, proposto por Lewis et al. (2020), resolve o problema da alucinação com uma abordagem engenhosa: em vez de pedir ao LLM que "lembre" da resposta, nós **buscamos** a informação relevante em uma base de dados confiável e a **injetamos** no Prompt antes da geração.

O fluxo tem três etapas:

1. **Retrieve (Recuperar):** A pergunta do usuário é convertida em vetor e buscada no Vector Store. Os Top-K documentos mais similares são retornados.
2. **Augment (Aumentar):** Os documentos recuperados são concatenados ao Prompt original, criando um *contexto enriquecido*.
3. **Generate (Gerar):** O LLM recebe o Prompt aumentado e gera a resposta **baseada exclusivamente nos documentos fornecidos**.

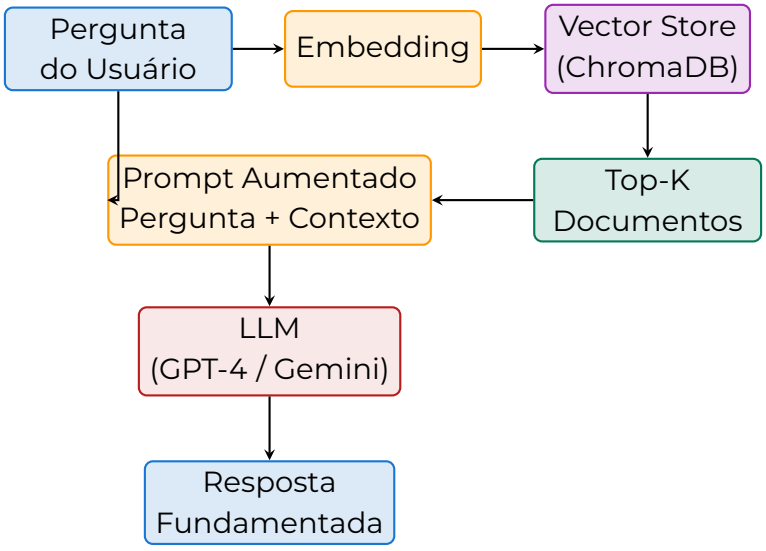

Figura 1: Fluxo completo do RAG: a pergunta é vetorizada, documentos relevantes são recuperados do Vector Store, e o LLM gera a resposta baseada exclusivamente no contexto fornecido.

3 Implementação em Python Puro

▶ Prática: title=RAG Completo em Python

```python
import chromadb
from openai import OpenAI

# 1. RETRIEVE: Buscar documentos relevantes
client_chroma = chromadb.Client()
collection = client_chroma.get_collection("meus_docs")

resultados = collection.query(
    query_texts=["O que e overfitting?"],
    n_results=3
)
contexto = "\n".join(resultados['documents'][0])

# 2. AUGMENT: Montar o prompt enriquecido
prompt_rag = f"""Responda a pergunta usando APENAS
o contexto fornecido. Se a resposta nao estiver no
contexto, diga "Nao encontrei essa informacao".

Contexto:
{contexto}

Pergunta: O que e overfitting?
Resposta:"""

# 3. GENERATE: Chamar o LLM
client_openai = OpenAI()
resposta = client_openai.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": prompt_rag}],
    temperature=0.0  # Determinismo maximo
)
print(resposta.choices[0].message.content)
```

4 Métricas de Avaliação de RAG (RAGAS)

Avaliar um sistema RAG vai além de "a resposta parece boa". O framework **RAGAS** (Retrieval Augmented Generation Assessment) define quatro métricas fundamentais:

- **Faithfulness (Fidelidade):** A resposta é fiel ao contexto? Cada afirmação na resposta pode ser sustentada pelos documentos recuperados?

- **Answer Relevancy (Relevância):** A resposta efetivamente responde à pergunta feita, ou divaga sobre temas tangenciais?
- **Context Precision:** Os documentos recuperados são realmente relevantes para a pergunta? Ou o sistema trouxe "lixo semântico"?
- **Context Recall:** O sistema recuperou **todos** os documentos necessários para responder completamente?

• Teoria: title=A Diferença entre Alucinação e Irrelevância

- **Alucinação:** O LLM **inventa** informação que não está no contexto. Faithfulness baixa.
- **Irrelevância:** O LLM responde corretamente algo que não foi perguntado. Answer Relevancy baixa.
- **Contexto ruim:** O Vector Store retornou documentos errados, e o LLM respondeu fielmente... ao documento errado. Context Precision baixa.

Cada falha exige uma intervenção diferente: Prompt Engineering, Reranking, ou refinamento do Chunking.

5 Conclusão

O RAG artesanal em Python demonstra que a arquitetura é conceitualmente simples: Buscar → Injetar → Gerar. Mas em produção, o Python puro apresenta limitações sérias: ausência de injeção de dependências, falta de tratamento de erros robusto, e dificuldade de escalar para múltiplos usuários simultâneos. Na próxima aula, faremos a transição para o ecossistema **Spring AI**, onde construiremos o mesmo RAG com a robustez de um framework corporativo Java.

✔ Takeaways

- **Alucinação** é o problema central dos LLMs: o modelo gera texto plausível mas factualmente falso, com confiança absoluta.
- O **RAG** resolve a alucinação injetando documentos reais no prompt: *Retrieve* (buscar no Vector Store) → *Augment* (concatenar ao prompt) → *Generate* (gerar resposta fundamentada).
- A instrução "responda apenas com base no contexto fornecido" é o mecanismo que ancora o LLM nos documentos, reduzindo alucinações.
- O framework **RAGAS** define quatro métricas essenciais: *Faithfulness*, *Answer Relevancy*, *Context Precision* e *Context Recall*.
- Cada tipo de falha (alucinação, irrelevância, contexto ruim) exige uma intervenção diferente: Prompt Engineering, Reranking ou refinamento do Chunking.
- O parâmetro `temperature=0.0` é obrigatório em RAG corporativo para maximizar o determinismo das respostas.

✎ Questões: Autoavaliação

1 **[Direta]** Descreva as três etapas do fluxo RAG (Retrieve, Augment, Generate) e explique o papel de cada componente técnico (Embedding, Vector Store, LLM).

2 **[Direta]** Qual a diferença entre *Faithfulness* baixa e *Context Precision* baixa? Dê um exemplo concreto de cada falha.

3 **[Direta]** No código Python apresentado, por que utilizamos `temperature=0.0`? O que aconteceria com a qualidade das respostas se usássemos `temperature=1.0`?

4 **[Reflexiva]** O RAG garante respostas 100% corretas? Discuta cenários em que mesmo um RAG bem implementado pode falhar (ex: documentos desatualizados, perguntas fora do domínio, chunking destrutivo).

5 **[Reflexiva]** Compare o custo-benefício de usar RAG versus Fine-Tuning para adaptar um LLM a um domínio específico. Em que situações cada abordagem é mais adequada?

Lab. 17: O RAG Feito à Mão

Capítulo em Revisão

Este conteúdo está sendo revisado. Algumas informações podem estar preliminares, conter erros de formatação ou sofrer alterações na versão final.

⋆ Objetivo do Laboratório

Neste laboratório, você construirá o pipeline completo da arquitetura RAG (Retrieval-Augmented Generation) do zero, usando Python. O script fará uma busca num banco ChromaDB, extrairá contextos e injetará o texto em um prompt rígido, forçando o LLM a atuar como um sumarizador fiel dos seus dados locais.

6 Atividade Prática

1. Pré-Requisitos e Setup

Certifique-se de que as bibliotecas necessárias estão instaladas. Usaremos o `chromadb` para a busca semântica e o provedor generativo (ex: `google-generativeai`).

```
pip install chromadb google-generativeai python-dotenv
```

Crie um arquivo chamado `rag_manual.py`.

2. O Código: Construindo a Arquitetura

Este script pressupõe que você já tenha populado um diretório do ChromaDB (conforme visto na Aula 12 de Embeddings).

```
import os
import chromadb
import google.generativeai as genai
from dotenv import load_dotenv

# 1. Carregamento e Configuração
load_dotenv()
genai.configure(api_key=os.getenv("API_KEY"))

# 2. Conectando ao Banco de Dados Vetorial (ChromaDB)
```

```
# Ajuste o path para a pasta do seu banco populado
chroma_client = chromadb.PersistentClient(path="./meu_banco_chroma")
colecao = chroma_client.get_collection(name="documentos_empresa")

# 3. A Pergunta do Usuário
pergunta_usuario = "Quais são as regras para home-office na empresa?"

# 4. Passo 1 do RAG: Recuperação (Retrieval)
print("Buscando no banco de dados...")
resultados_busca = colecao.query(
    query_texts=[pergunta_usuario],
    n_results=2 # Retorna os 2 parágrafos mais relevantes
)

# Extrai os textos relevantes encontrados (o 'Contexto')
contextos_encontrados = resultados_busca['documents'][0]
contexto_concatenado = "\n".join(contextos_encontrados)

# 5. Passo 2 do RAG: Construção do Prompt Aumentado
prompt_rag = f"""
Você é um assistente corporativo. Responda à pergunta do usuário usando APENAS
o contexto fornecido abaixo. Não invente informações.
Se o contexto não contiver a resposta, diga "Desculpe, não encontrei a informação".

CONTEXTO:
{contexto_concatenado}

PERGUNTA DO USUÁRIO:
{pergunta_usuario}
"""

# 6. Passo 3 do RAG: Geração (Generation)
print("Invocando a IA Generativa...\n")
modelo = genai.GenerativeModel('gemini-1.5-flash')
resposta = modelo.generate_content(prompt_rag)

print("--- Resposta do RAG ---")
print(resposta.text)
```

3. Entrega via Git

1. Execute o script e faça testes. Tente fazer uma pergunta que SABEMOS estar no banco, e comprove que o LLM respondeu perfeitamente com os dados injetados.
2. Faça uma pergunta fora do contexto da empresa (ex: "Quem ganhou a copa de 2002?"). O modelo DEVE se recusar a responder, graças à sua instrução restrita.
3. Comite as modificações com `git commit -m "Implementa arquitetura RAG pura em Python"` e envie ao GitLab.

7 Critério de Avaliação

O script deve demonstrar claramente os três passos: Conexão com VectorDB para busca, Template de Interpolação (f-string) misturando Contexto e Pergunta, e Chamada da API em Nuvem com esse Prompt Aumentado.

✔ Takeaways

- Você construiu uma arquitetura **RAG completa do zero** em Python puro: Retrieve (ChromaDB) → Augment (f-string) → Generate (Gemini).
- A instrução “responda APENAS com base no contexto fornecido” é o **mecanismo anti-alucinação** — o modelo é forçado a se ancorar nos dados reais.
- O teste de “pergunta fora do domínio” comprova que o guardrail funciona: o LLM recusa responder quando o contexto não contém a informação.
- O pipeline inteiro cabe em menos de 30 linhas de Python — a simplicidade é intencional para que você entenda cada peça antes de usar frameworks.

◇ Informação

Gancho para a Próxima Aula: O RAG em Python funciona, mas tem limitações sérias: sem injeção de dependências, sem tipagem estática, sem tratamento de erros robusto. Na próxima aula, faremos a transição para o **Spring AI** — o framework corporativo Java que transforma o mesmo RAG em um componente testável, injetável e pronto para produção.