Lecture Notes: Retrieval-Augmented Generation (RAG)

1. What is RAG?

Definition

Retrieval-Augmented Generation (RAG) is the process of enhancing and optimizing the output of a large language model (LLM) by providing it with additional context retrieved from external data sources, ensuring more accurate, relevant, and grounded responses.

Why Do We Need RAG?

Standard LLMs are trained on a massive, but static, dataset. This leads to two major limitations:

1. **Stale Knowledge:** The model has no information about anything that happened after its training data was collected.
2. **Hallucinations:** The model can confidently "make up" facts or source citations that are plausible-sounding but incorrect.
3. **Lack of Specificity:** The model has no access to private, proprietary, or domain-specific information (e.g., your company's internal policies, a user's personal documents, or real-time financial data).

RAG solves this by connecting the LLM to a live, external knowledge source and "grounding" its answers in facts.

2. The Core RAG Workflow

At its core, a RAG system performs three steps in response to a user's prompt:

1. **Retrieve:** The user's prompt (e.g., "What is our company's policy on remote work?") is transformed into a query. The system searches an external data source (the "corpus") for relevant information.
2. **Augment:** The relevant information (e.g., the text from the "Remote Work Policy.pdf" document) is retrieved and combined with the original prompt. This new, "augmented" prompt now contains both the user's question and the factual context.
3. **Generate:** This augmented prompt is fed to the LLM. The LLM is instructed to generate an answer *based only on the provided context*. This forces the model to stick to the facts, effectively using it as a reasoning and summarization engine rather than a knowledge recall engine.

3. Methods of Retrieving External Information

The "retrieval" step is what makes RAG powerful. This external information can come from several sources.

A. The RAG Corpus (The Knowledge Base)

This is the most common method. The corpus is the private, curated set of documents that you want the LLM to have access to.

- What is a Corpus?
  A corpus is a large, organized collection of text and data used as the "knowledge library" for the RAG system. It is the "ground truth" that the LLM will consult.
- Role in RAG:
  The corpus serves as the long-term memory for the LLM. Instead of retraining the entire model (which is slow and expensive), you simply add new information to the corpus (e.g., add a new policy PDF).
- Composition (How it's built):

A RAG corpus is not just a folder of files. It's a highly optimized, searchable index, typically built using a Vector Database.

1. **Ingestion:** You load your raw data (e.g., PDFs, .txt files, Markdown, database entries, scraped web pages).
2. **Chunking:** The documents are broken down into smaller, manageable "chunks" (e.g., paragraphs or sentences). This is critical because you can't fit an entire 100-page document into the LLM's context window.
3. **Embedding:** Each chunk is passed through an *embedding model*, which converts the text into a numerical representation (a "vector"). This vector captures the semantic meaning of the text.
4. **Indexing:** These vectors are stored in a **Vector Database**. This database is optimized to find vectors (and their corresponding text chunks) that are "semantically similar" to the user's query vector.

B. Web Scraping

This is a method for *building* a corpus by pulling data directly from the public web.

- How it Works:
  Web scraping is an automated process that fetches web pages and extracts specific data from them. A "scraper" bot visits a URL, downloads the HTML, and then parses that HTML to pull out the desired text (e.g., blog post content, product descriptions, news articles).
- Using Python for RAG:
  Python is the most common language for this. The workflow for building a RAG corpus from scraped data often looks like this:
  1. **Fetch:** Use libraries like requests to download the web page.
  2. **Parse:** Use libraries like BeautifulSoup or Scrapy to navigate the HTML and extract the main text content, stripping away ads, navigation bars, and footers.
  3. **Clean:** Convert the extracted content into clean text or Markdown.
  4. **Ingest:** Feed this clean text into your RAG corpus pipeline (chunking, embedding, indexing) as described in section 3A.
  5. Frameworks like LlamaIndex and LangChain have built-in "Web Readers" (ScrapflyReader, RAGScraper) that can simplify this process into a few lines of code.

C. APIs (Application Programming Interfaces)

This method is used to retrieve highly dynamic, real-time, or structured data.

- Role in RAG:
  Instead of just searching a static database, the RAG system can be given "tools," and one of those tools can be an API. This is essential when an answer depends on live data.
- **How APIs Contribute:**
  - **Free & Open Data Sources:** Imagine a user asks, "What's the weather in San Francisco and what's the top news story about AI?" A RAG system can be configured to:
    1. Parse the prompt into two sub-questions.
    2. Call a free **Weather API** for the San Francisco data.
    3. Call a free **News API** (like GNews) for the AI story.
    4. Combine both API responses (which are usually in a structured JSON format) into the "context."
    5. Feed this context to the LLM to generate a single, natural-language answer.
  - **Internal/Proprietary APIs:** A chatbot for an e-commerce store could use an internal API to check a user's "order status" or "product availability" from the company's live database.

4. Real-World Use Cases

RAG is not theoretical; it's already powering many applications you use.

- **Customer Support Chatbots (e.g., Shopify, DoorDash):** When you ask a chatbot a question, it retrieves

information from the company's *actual* help center articles, product manuals, and internal FAQs to give you a correct, specific answer instead of a generic guess.

- **Healthcare (e.g., IBM Watson Health):** A RAG system can assist doctors by taking a patient's symptoms (the prompt) and retrieving the most relevant, up-to-date research papers, clinical trial data, and treatment guidelines (the corpus) to support a diagnosis.
- **Finance (e.g., Bloomberg):** An analyst can ask, "Summarize the key risks from Apple's latest quarterly earnings report." The RAG system retrieves the *actual 10-Q report* (which was just released) and provides it to the LLM for summarization, ensuring the answer is based on real-time, factual data.
- **Internal Knowledge Management (e.g., Assembly, Bell):** Employees at a large company can use an internal "Ask HR" bot. This bot uses RAG to search all internal documents (employee handbooks, policy PDFs, intranet sites) to answer questions like "How do I submit an expense report?"

5. Building a RAG System with Vertex AI (Google Cloud)

Google Cloud provides a powerful, managed platform to build and scale RAG systems using **Vertex AI**.

Key Components

Vertex AI simplifies the RAG process by integrating all the necessary components:

1. **Vertex AI RAG Engine:** This is the overarching service that manages the entire workflow, from data ingestion to generation.
2. **Data Ingestion & Connectors:** You don't have to build complex data pipelines. You can point Vertex AI to your data sources, including:
    - Google Cloud Storage (for PDFs, .txt, etc.)
    - Google Drive
    - Third-party sources like Jira, Slack, or Salesforce.
3. **RagCorpus:** This is the Vertex AI-managed service that automatically handles the data processing. When you add a document, it automatically performs **chunking** and **parsing**.
4. **Vertex AI Vector Search:** This is the underlying vector database that creates the embeddings and performs the ultra-fast semantic search.
5. **Generative Models (Gemini):** The retrieved context is automatically fed to a Google Gemini model for the final, grounded answer generation.

The Vertex AI Workflow (Simplified)

While the console provides a "clicks-not-code" interface, you can also build this system programmatically using the Python SDK.

1. **Initialize Vertex AI:** Connect to the Vertex AI platform.
   ```
   import vertexai
   vertexai.init(project="your-project-id", location="us-central1")
   ```

2. **Create a Corpus:** Create a RagCorpus to hold your data.
   ```
   from vertexai.preview import rag
   rag_corpus = rag.create_corpus(display_name="my-company-corpus")
   ```

3. **Import Files:** Upload your files (e.g., from a Google Cloud Storage bucket) into the corpus. Vertex AI handles the chunking and embedding automatically.
   ```
   rag.import_files(
     rag_corpus.name,
     ["gs://my-bucket/my-policy-doc.pdf"],
   ```

```
    chunk_size=1024
)
```

4. **Set up the RAG Tool:** Create a RAG retrieval "tool" that the LLM can use.
```
from vertexai.preview.generative_models import GenerativeModel, Tool

rag_retrieval_tool = Tool.from_retrieval(
    rag.Retrieval(source=rag.VertexRagStore(rag_corpora=[rag_corpus.name]))
)
```

5. **Generate a Grounded Answer:** Initialize a Gemini model and give it the RAG tool. Now, when you ask a question, the model will *automatically* use the tool to find context from your corpus before answering.
```
model = GenerativeModel(
    "gemini-1.5-flash-001",
    tools=[rag_retrieval_tool]
)

prompt = "What is the company policy on remote work?"
response = model.generate_content(prompt)

print(response.text)
# This output will be grounded in the "my-policy-doc.pdf"
```