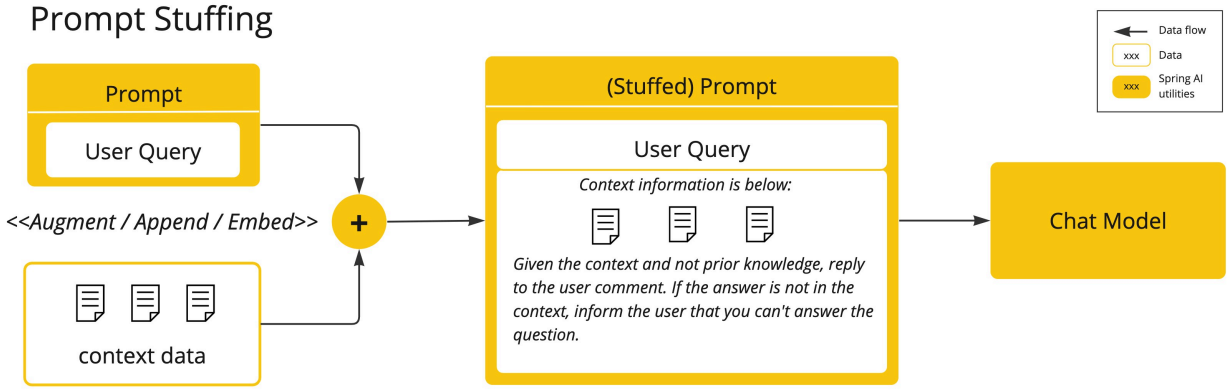
RAG (Retrieval Augmented Generation)

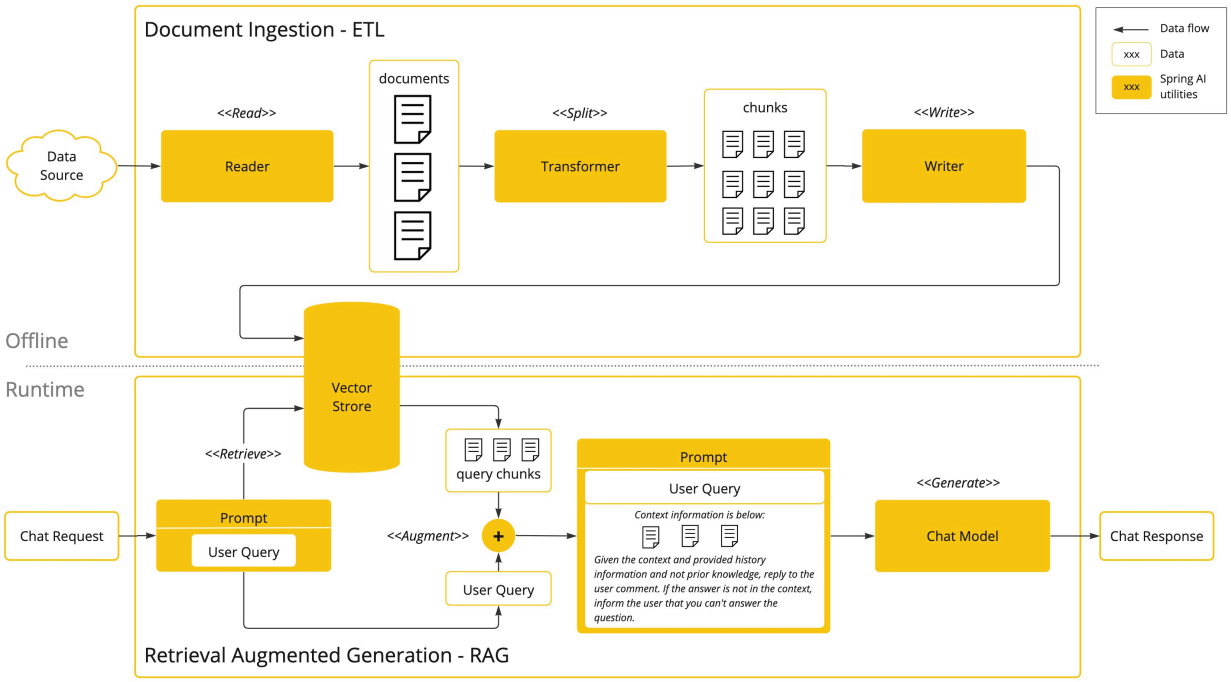
1. The Problem:



You cannot simply paste 500 pages of documentation into a prompt.

- **Context Window Limits:** Models have limits (e.g., 8k, 128k tokens). A 500-page PDF is ~250k tokens. It won't fit.
- **Cost:** You pay per token. Sending a book for every "Hello" is prohibitively expensive.
- **"Lost in the Middle":** LLMs are great at remembering the beginning and end of a prompt but often hallucinate or forget details buried in the middle of massive text blocks.
- **Solution: RAG**. Only send the 3-5 most relevant paragraphs (chunks) needed to answer the specific question.

2. The RAG "ETL" Pipeline (Extract, Transform, Load)



Before you can chat, you must prepare your data. This is a one-time "ingestion" process.

- **Document Loaders (Extract):** Spring AI provides specific readers to pull text from raw files.
  - **PDF:** `PagePdfDocumentReader` (uses Apache PdfBox).
  - **Text/Markdown:** `TextReader`.
  - **JSON:** `JsonReader` (great for structured data).
- **Text Splitters (Transform):** You must break documents into "bite-sized" pieces (chunks) for the AI.
  - `TokenTextSplitter` **(The Standard):** Splits by token count (e.g., 800 tokens). Safe for LLM context windows. Preserves context by keeping "edges" of paragraphs intact.
  - **Semantic Chunking:** Advanced. Uses an LLM to decide where a "topic" ends. Slower but higher quality.
  - Use **JChunk** or similar libraries to support different types of chunking: https://docs.jchunk.io/docs/category/chunkers
- **Embedding & Storing (Load):**

- Convert chunks to vectors.
- Save to `VectorStore` (Postgres/PGVector).

3. Configuring Retrieval: Accuracy vs. Recall

When the user asks a question, how do we find the "right" chunks? You tune the SearchRequest inside the advisor.

- `topK` **(Default: 4):** How many chunks to retrieve.
  - *Low (1-2):* Cheaper, focused, but might miss context.
  - *High (10+):* Comprehensive, but risks "noise" and higher cost.
- `similarityThreshold` **(0.0 to 1.0):** The "Quality Gate".
  - *0.7 - 0.8:* High relevance. "Only show me exact matches."
  - *0.5:* Loose relevance. "Show me anything vaguely related."

4. Hands-on: Building a "Spring Boot Q&A Bot"

```
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-pdf-document-reader</artifactId>
</dependency>
```

Step 1: The Ingestion Service (Run once on startup)

This loads the Spring Boot Reference PDF, splits it, and saves it to the Vector DB.

```
@Component
public class IngestionService {

  private final VectorStore vectorStore;
  @Value("classpath:spring-boot-reference.pdf")
```

```
    private Resource pdfResource;

    public IngestionService(VectorStore vectorStore) {
        this.vectorStore = vectorStore;
    }

    public void ingest() {
        // 1. Load: Read PDF
        var pdfReader = new PagePdfDocumentReader(pdfResource);
        List<Document> rawDocs = pdfReader.get();

        // 2. Transform: Split into 800-token chunks
        var splitter = new TokenTextSplitter();
        List<Document> chunks = splitter.apply(rawDocs);

        // 3. Load: Store in PGVector
        vectorStore.add(chunks);
        System.out.println("Ingested " + chunks.size() + " chunks!");
    }
}
```

Step 2: The ChatBot (The RAG Endpoint)

We configure the QuestionAnswerAdvisor to use the "Stuff" strategy with strict filtering.

```
@RestController
public class DocChatController {

    private final ChatClient chatClient;
    private final VectorStore vectorStore;

    public DocChatController(ChatClient.Builder builder, VectorStore vectorStor
e) {
        this.vectorStore = vectorStore;
```

```
        this.chatClient = builder.build();
    }

    @GetMapping("/chat")
    public String chat(@RequestParam String query) {
        return chatClient.prompt()
            .advisors(
                // RAG Advisor with custom tuning
                QuestionAnswerAdvisor.builder(vectorStore)
                    .searchRequest(SearchRequest.builder()
                        .topK(4)                    // Fetch 4 chunks
                        .similarityThreshold(0.7)   // Only high relevance
                        .build())
                    .build()
            )
            .user(query)
            .call()
            .content();
    }
}
```