A Developer's Roadmap to Building AI Applications with Spring AI

Section 1: Introduction: Bridging Spring and Artificial Intelligence

1.1 The AI Landscape for Java Developers: A New Frontier

The rapid advancements in Artificial Intelligence (AI), particularly in Generative AI and Large Language Models (LLMs), are reshaping the technological landscape. For Java developers, this presents a new frontier filled with opportunities to build innovative and intelligent applications. The demand for developers proficient in Java who also possess AI skills is on the rise, with generative AI skills reportedly boosting tech salaries significantly.[1] AI is no longer a niche field; it's becoming integral to enterprise software development. Common AI use cases in Java applications include developing sophisticated chatbots, creating personalized recommendation engines, leveraging Natural Language Processing (NLP) for text analysis and understanding, and even assisting in code generation and software development processes.[1] This evolving landscape underscores the need for robust frameworks that can seamlessly integrate AI capabilities into the Java ecosystem.

The integration of AI into enterprise Java applications is becoming increasingly critical. Enterprises have long relied on Java and the Spring Framework for building mission-critical systems due to their robustness, scalability, and extensive ecosystem. As AI becomes a transformative force, the ability to incorporate AI functionalities directly within these existing Java-based systems, or to build new AI-powered enterprise applications, necessitates a framework that aligns with established enterprise standards. These standards include reliability, maintainability, comprehensive observability, and the ability to manage complex dependencies and configurations. Spring AI emerges as a pivotal solution in this context, designed to bring AI capabilities into the Spring ecosystem while upholding these enterprise-grade requirements. It leverages Spring Boot's production-minded ethic, offering features like observability through Micrometer, support for GraalVM native images for performance, and seamless integration with development tools.[2] Furthermore, Spring AI's abstractions for AI models and vector stores provide a crucial advantage by de-risking technology choices, allowing enterprises to adapt to the rapidly changing AI landscape without being locked into specific vendors.[5] This positions Spring AI not merely as a developer tool but as a strategic component for enterprises aiming to embed AI deeply within their Java technology stack.

1.2 Introducing Spring AI: Purpose, Philosophy, and Core Benefits

The Spring AI project aims to simplify the development of applications that integrate artificial intelligence functionalities, abstracting away unnecessary complexity.[5] Inspired by Python projects such as LangChain and LlamaIndex, Spring AI is not a direct port but rather a fresh take tailored for the Java ecosystem. It operates on the belief that Generative AI applications will become ubiquitous across many programming languages, extending beyond Python's current dominance.[5] The core purpose of Spring AI is to apply the design principles of the Spring ecosystem—such as portability, modular design, and the promotion of Plain Old Java Objects (POJOs) as application building blocks—to the AI domain.[6] At its heart, Spring AI addresses the fundamental challenge of AI integration: connecting enterprise data and APIs with AI Models effectively and efficiently.[5]

The philosophy behind Spring AI is to serve as the de-facto standard for Java developers venturing into AI. For these developers, Spring AI offers several core benefits:

- **Portability:** Spring AI provides powerful abstractions for AI models (chat, embedding, image, etc.) and Vector Stores. This allows developers to swap out different AI providers (e.g., OpenAI for Anthropic, or ChromaDB for PGVector) with minimal code changes.[3] This portability is paramount in the fast-evolving AI landscape, preventing vendor lock-in and enabling applications to adapt to new, potentially superior, models or data stores as they emerge.
- **Spring Ecosystem Integration:** A significant advantage is its deep integration with the familiar Spring Boot framework. This includes auto-configuration for AI models and vector stores, dedicated Spring Boot starters, and integration with Micrometer for observability.[2] It also supports modern Java features like virtual threads for improved scalability, GraalVM native images for optimized performance and resource usage, and developer productivity tools like Spring Boot DevTools, Docker Compose, and Testcontainers support.[2] This means developers can leverage their existing knowledge and the robust features of the Spring ecosystem when building AI applications.
- **Simplified Development:** Spring AI offers fluent APIs, such as the ChatClient, which is designed to be idiomatically similar to WebClient and RestClient, making it intuitive for Spring developers.[5] Additionally, the Advisors API encapsulates recurring Generative AI patterns, transforming data sent to and from LLMs and providing portability across various models and use cases.[3]

For developers already steeped in the Spring Framework, the journey into AI development with Spring AI feels like a natural progression rather than a leap into an entirely new paradigm. The framework's intentional mirroring of Spring's design patterns significantly flattens the learning curve. Spring developers are accustomed to concepts like dependency injection, auto-configuration, starter dependencies, and fluent APIs.[2] Spring AI is meticulously built upon these very principles: it functions like any other Spring project, offering portable service abstractions, Spring Boot starters for various AI models and vector stores, externalized configuration properties, and automatic configuration of beans.[2] The ChatClient, a cornerstone of Spring AI, is explicitly designed to resemble the WebClient and RestClient APIs, which are staples in modern Spring development.[5] This inherent familiarity reduces the cognitive overhead often associated with learning new AI-specific frameworks, especially those originating from different language ecosystems. Consequently, developers can dedicate more of their focus to understanding core AI concepts—such as LLMs, Retrieval Augmented Generation (RAG), and prompt engineering—rather than grappling with entirely unfamiliar programming models or toolchains.

1.3 Overview of the Roadmap: Your Journey into Spring AI

This roadmap is designed to guide Java Spring Framework developers on their journey to mastering Spring AI and building sophisticated AI-powered applications. It follows a progressive learning path, starting with foundational AI concepts and Spring AI setup, moving through core API mastery and data-aware application development with RAG, and culminating in advanced topics like function calling, agentic patterns, and production considerations. A key emphasis throughout this roadmap is a hands-on, project-based approach, ensuring that theoretical knowledge is solidified with practical application development experience.

The following table provides a high-level summary of the stages in this roadmap:

Table 1: Roadmap Stage Summary

Stage	**Stage Title**	**Key Topics Covered**	**Core Learning Objectives**	**Example Project Titles**
1	Foundations: Your First Steps into Spring AI	Core AI Concepts (LLMs, Embeddings, Vector Stores), Spring AI Environment Setup, Basic Chatbot (OpenAI,	Understand fundamental AI terminology, set up a Spring AI development environment, build a first simple AI application using	"Hello, AI" Command-Line Chat App, Local LLM Query Tool with Model Switching

		Ollama)	ChatClient.	
2	Mastering Core Spring AI APIs	Deep Dive: ChatClient, EmbeddingClient; Effective Prompt Engineering; Handling Structured Outputs (POJO Mapping); Working with Various AI Models (Hugging Face)	Gain in-depth understanding and proficiency in using Spring AI's core APIs for diverse and effective interactions with LLMs and embedding models.	Advanced Q&A Bot with Prompt Engineering, Text Summarizer and Analyzer (POJO Output)
3	Building Data-Aware Applications with RAG	Retrieval Augmented Generation (RAG), Vector Store API, Integration (ChromaDB, PGVector, Redis), Document ETL, RAG with Advisors	Understand and implement RAG to build AI applications that leverage external knowledge bases, improving accuracy and relevance.	"Chat with Your PDF" Application, Semantic Search Engine for Product Documentation
4	Enhancing AI Applications with Advanced Interactions	Function Calling/Tool Usage, Chat Memory for Conversational Context, Multimodality (Image and Audio Models)	Extend AI application capabilities by enabling LLMs to interact with external tools, maintain conversation history, and process multimodal data.	AI Assistant with Real-time Data Access, Contextual Customer Service Bot
5	Towards Production-Ready and Agentic	AI Model Evaluation, Observability	Prepare AI applications for production by	RAG Output Evaluator, Simple Research

	AI Solutions	(Metrics, Tracing), Agentic Patterns, Best Practices, Common Challenges	implementing evaluation, monitoring, and adopting robust development practices. Explore agentic designs.	Agent (Chain Workflow + Function Calling)

Section 2: Stage 1 - Foundations: Your First Steps into Spring AI

This initial stage lays the groundwork for your journey into Spring AI. It covers essential AI concepts tailored for Spring developers, guides you through setting up a development environment, and culminates in building your first simple AI application.

2.1 Core AI Concepts for Spring Developers

Before diving into code, it's beneficial to understand a few core AI concepts that are central to working with Spring AI and modern LLMs:

- **LLMs (Large Language Models):** These are sophisticated AI models trained on vast amounts of text data. At a high level, they work by predicting the next word or token in a sequence, given the preceding text. This capability allows them to generate human-like text, answer questions, summarize information, translate languages, and much more. A crucial characteristic of many LLMs, especially when accessed via APIs, is their **statelessness**.[2] This means each interaction (request-response pair) is typically independent, and the model doesn't inherently remember past interactions unless explicitly provided with that context.
- **Prompts:** A prompt is the input provided to an LLM to elicit a response. Crafting effective prompts—a practice known as prompt engineering—is critical for guiding the LLM to produce the desired output. Prompts can range from simple questions to complex instructions with examples.
- **Embeddings:** Embeddings are numerical representations (typically vectors or lists of floating-point numbers) of data such as text, images, or audio. These vectors capture the semantic meaning or essence of the data. For instance, texts with similar meanings will have embeddings that are close to each other in the vector space. Embeddings are fundamental for tasks like semantic search, recommendation systems, and Retrieval Augmented Generation (RAG).
- **Vector Stores:** These are specialized databases designed to store and efficiently query large volumes of embeddings. They allow for fast similarity searches, finding vectors (and thus the original data they represent) that are semantically

closest to a given query vector. Examples include ChromaDB, PGVector (an extension for PostgreSQL), and Redis with vector search capabilities.
- **Tokens:** LLMs process text by breaking it down into smaller units called tokens. A token can be a word, part of a word, or even a single character, depending on the model's tokenizer. Understanding tokens is important because LLM context windows (the amount of text they can consider at once) are often measured in tokens, and API pricing is also frequently based on token usage.

A high-level overview of AI concepts and their representation within Spring AI can be found in the project's concepts documentation.[5]

2.2 Setting Up Your Spring AI Development Environment

To start developing with Spring AI, you'll need a standard Java development environment along with a few specific considerations:

- **Prerequisites:**
  - Java Development Kit (JDK): Version 17 or later is recommended.
  - Build Tool: Apache Maven or Gradle.
  - Spring Boot: A recent version (e.g., 3.x) is advisable to leverage the latest Spring AI features.
  - Docker: Highly recommended for running local LLMs (via Ollama) and vector databases.
- **Spring Initializr (start.spring.io):** The easiest way to bootstrap a new Spring AI project is by using the Spring Initializr. When creating your project, you can select the necessary Spring AI starters for the AI models (e.g., "Spring AI OpenAI Starter," "Spring AI Ollama Starter") and vector stores you intend to use.[3]
- **Dependencies:**
  - The core Spring AI dependency is typically org.springframework.ai:spring-ai-core.
  - You will add specific model starters based on your chosen AI provider, for example:
    - org.springframework.ai:spring-ai-openai-starter for OpenAI models.[8]
    - org.springframework.ai:spring-ai-ollama-starter for local models via Ollama.[10]
  - To ensure consistent versions across multiple Spring AI modules, it's best practice to include the Spring AI Bill of Materials (BOM) in your project's dependency management section.[8]

    XML
    ```xml
    // Example for Maven pom.xml
    <dependencyManagement>
    ```

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.ai</groupId>
            <artifactId>spring-ai-bom</artifactId>
            <version>${spring-ai.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- **API Keys & Configuration:**
  - If you're using cloud-based AI models like those from OpenAI, you'll need an API key. It is crucial to manage these keys securely. **Never hardcode API keys in your source code.** Instead, use environment variables, externalized configuration files (like application.properties or application.yml outside your packaged application), or a secure secrets management system.
  - Spring AI defines configuration properties for API keys, such as spring.ai.openai.api-key=<YOUR_OPENAI_KEY>.[6] This can be set in application.properties:
    Properties
    ```
    spring.ai.openai.api-key=${OPENAI_API_KEY}
    # For Ollama, if not running on default http://localhost:11434
    # spring.ai.ollama.base-url=http://your-ollama-host:port
    ```
    And the actual key provided via an environment variable OPENAI_API_KEY.

2.3 Your First Spring AI Application: A Simple Chatbot

The objective here is to create a minimal application that interacts with an LLM. This will introduce the ChatClient, the primary interface for chat model communication in Spring AI.

The ChatClient provides a fluent API for sending prompts to an LLM and receiving responses. A basic example of its usage within a Spring Boot CommandLineRunner could look like this [6]:

Java

```java
import org.springframework.ai.chat.client.ChatClient;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@Component
class SimpleChatbot {

    // Assuming a ChatClient.Builder bean is auto-configured by Spring AI
    // based on your starter dependencies and application properties.
    private final ChatClient chatClient;

    public SimpleChatbot(ChatClient.Builder chatClientBuilder) {
        this.chatClient = chatClientBuilder.build();
    }

    @Bean
    public CommandLineRunner simpleChatRunner() {
        return args -> {
            String userMessage = "Tell me a joke about Java developers and AI.";
            System.out.println("User: " + userMessage);

            String response = chatClient.prompt()
                                  .user(userMessage)
                                  .call()
                                  .content();

            System.out.println("AI: " + response);
        };
    }
}
```

This example assumes that a default AI model (like OpenAI or Ollama) is configured in your application.properties or application.yml file.

2.3.1 Integrating with OpenAI

OpenAI's models (like GPT-3.5-turbo or GPT-4o) are a popular choice for many AI applications.

- **Dependency:** Add the spring-ai-openai-starter to your Maven pom.xml or Gradle build.gradle file.[8]

```xml
// Maven
<dependency>
   <groupId>org.springframework.ai</groupId>
   <artifactId>spring-ai-openai-starter</artifactId>
</dependency>
```

- **Configuration:** Set your OpenAI API key and optionally the model name in application.properties [8]:

  ```properties
  spring.ai.openai.api-key=${OPENAI_API_KEY}
  spring.ai.openai.chat.options.model=gpt-4o-mini // Or gpt-3.5-turbo, gpt-4o, etc.
  ```

  With these settings, the ChatClient.Builder will auto-configure an OpenAiChatModel.

2.3.2 Exploring Local LLMs with Ollama

Ollama allows you to run powerful open-source LLMs locally on your machine, which is excellent for development, experimentation, cost-saving, and applications requiring data privacy or offline capabilities.[10]

- **Setting up Ollama:**
  1. **Installation:** Download and install Ollama from the official website (https://ollama.com/download) for your operating system.[10]
  2. **Start Ollama Server:** Open a terminal and run ollama serve. This will start the Ollama server, typically listening on http://localhost:11434.[10]
  3. **Pulling Models:** Download LLMs to run locally. For example, to get the Mistral model, run ollama run mistral. For a multimodal model like Llava, use ollama run llava.[10] You can find a list of available models on the Ollama library website. For embedding tasks later, a model like nomic-embed-text can be pulled using ollama pull nomic-embed-text.[13]
- **Spring AI Configuration for Ollama:**
  - **Dependency:** Add the spring-ai-ollama-starter to your project.[10]

    ```xml
    // Maven
    <dependency>
       <groupId>org.springframework.ai</groupId>
       <artifactId>spring-ai-ollama-starter</artifactId>
    </dependency>
    ```

  - **Properties:** Configure the model to use in application.properties. The base

URL defaults to http://localhost:11434 but can be overridden if needed.[10]

```
Properties
spring.ai.ollama.chat.options.model=mistral // Or llava, llama3, etc.
# spring.ai.ollama.base-url=http://custom-ollama-host:11434 # If not default
```

With these settings, the ChatClient.Builder will auto-configure an OllamaChatModel.

The immediate benefit of Spring AI's design becomes apparent here. The ChatClient abstraction allows developers to write interaction logic that is largely independent of whether OpenAI or a local Ollama model is the underlying LLM. The core Java code for prompting the model, as shown in the SimpleChatbot example, remains consistent. The primary differences lie in the Maven/Gradle dependencies added and the configuration properties set in application.properties or application.yml. This early demonstration of portability reinforces one of Spring AI's key value propositions, providing developers with immediate confidence in the framework's design and its ability to adapt to different AI backends.

Furthermore, the straightforward integration with Ollama significantly lowers the barrier to AI experimentation for Java developers. Traditionally, setting up and running local LLMs might involve navigating Python environments, managing complex dependencies, and specific hardware considerations, which can be daunting for developers primarily working within the Java ecosystem. Ollama greatly simplifies this local LLM setup process with commands like ollama serve and ollama run <model_name>.[10] Spring AI builds on this by providing a dedicated starter dependency for Ollama, making its integration into a Java Spring Boot project almost trivial.[10] This empowers Java developers to easily and cost-effectively experiment with a diverse range of open-source LLMs directly on their local machines, accelerating their learning curve and fostering innovation without the immediate concern of API costs or the complexities of a Python-centric AI stack.

Table 2: Local vs. Cloud LLM Development with Spring AI (Initial Perspective)

Aspect	**Local (Ollama with Spring AI)**	**Cloud (e.g., OpenAI with Spring AI)**
Cost	Generally free (uses local resources)	Pay-per-use (API calls, token consumption) [Implicit]

Setup Complexity	Ollama installation + Spring AI starter (relatively simple) [10]	API key generation + Spring AI starter (simple) [8]
Model Variety	Growing list of open-source models available via Ollama [10]	Access to proprietary state-of-the-art models (e.g., GPT-4o) [5]
Performance	Depends on local hardware; may be slower than cloud APIs	Generally high performance, managed infrastructure
Privacy	High (data stays local) [12]	Data sent to third-party provider (consider terms of service)
Offline Use	Yes (once models are downloaded)	No (requires internet connectivity)
Experimentation	Excellent for free, rapid iteration	Can be costly for extensive experimentation
Spring AI Setup	spring-ai-ollama-starter, application.properties for model	spring-ai-openai-starter, application.properties for API key/model

2.4 Hands-on Project Proposals for Stage 1

To solidify the concepts learned in this stage, consider undertaking the following projects:

- **Project 1: "Hello, AI" Command-Line Chat App**
  - **Description:** Develop a simple Java command-line application using Spring Boot. The application should take text input from the user via the console, send this input as a prompt to an LLM (configurable to use either OpenAI or a locally running Ollama model via application.properties), and then print the LLM's response back to the console.
  - **Learning Objectives:** Master basic Spring AI project setup, understand ChatClient usage, and practice configuring different AI models.
  - **Key Steps:**
    1. Create a new Spring Boot project using Spring Initializr, including the spring-ai-openai-starter and spring-ai-ollama-starter.
    2. Configure API keys for OpenAI (if using) and set up Ollama with a model

like "mistral".
    3. Implement a CommandLineRunner bean that uses ChatClient to interact with the configured LLM.
    4. Use System.in to read user input and System.out.println to display AI responses.
    5. Demonstrate switching between OpenAI and Ollama by changing active Spring profiles or commenting/uncommenting properties.
- **Project 2: Local LLM Query Tool with Model Switching**
  - **Description:** Build a Spring Boot web application with a very simple user interface (e.g., using Thymeleaf for server-side rendering, or just REST endpoints testable with tools like cURL or Postman). This application should allow users to send queries to a locally running Ollama instance. Implement a feature that allows the user to specify which Ollama model to use for the query, perhaps via an application property that can be changed and the application restarted, or more dynamically via a request parameter if you feel adventurous (though this requires more advanced ChatClient handling not yet covered).
  - **Learning Objectives:** Deepen Ollama integration skills, understand basic dynamic model selection concepts, and practice creating simple web endpoints for AI interaction.
  - **Key Steps:**
    1. Set up Ollama and pull at least two different models (e.g., "mistral" and "llama3").
    2. Add spring-boot-starter-web and spring-ai-ollama-starter dependencies.
    3. Create a RestController with an endpoint (e.g., /askOllama) that accepts a user query.
    4. Inject and use ChatClient configured for Ollama.
    5. Read the desired Ollama model name from application.properties (e.g., spring.ai.ollama.chat.options.model) and use it in the ChatClient interaction. Demonstrate changing this property to use different local models.

Section 3: Stage 2 - Mastering Core Spring AI APIs

Having established a foundational understanding and built a basic application, Stage 2 focuses on gaining a deeper mastery of Spring AI's core APIs. This includes advanced ChatClient usage, understanding EmbeddingClient for semantic tasks, delving into prompt engineering, handling structured data outputs, and integrating with a wider variety of AI models.

3.1 Deep Dive: ChatClient for Conversational AI

The ChatClient is the central abstraction for interacting with chat-based LLMs in Spring AI. Its fluent API, reminiscent of Spring's WebClient and RestClient, provides a powerful and intuitive way to construct and send prompts.[5]

- Fluent API for Prompt Construction:
  The builder pattern allows for clear and expressive construction of prompts.
  ```java
  // Example of fluent prompt building
  ChatResponse response = chatClient.prompt()
    .system("You are a helpful assistant that translates English to French.")
    .user("Hello, how are you?")
    .call();
  // String content = response.getResult().getOutput().getContent();
  ```
- Message Types:
  Spring AI supports different types of messages to structure conversations effectively:
  - **UserMessage:** Represents input from the end-user.
  - **SystemMessage:** Used to provide instructions, context, or define the persona for the AI model. This helps in guiding the AI's behavior and response style throughout the conversation.[2]
  - **AssistantMessage:** Represents a response from the AI. These can also be included in subsequent prompts as part of few-shot examples or to provide conversational history. Prompts can be constructed using a List<Message> passed to a Prompt object:

  ```java
  // Example using Prompt object with a list of messages
  // import org.springframework.ai.chat.messages.SystemMessage;
  // import org.springframework.ai.chat.messages.UserMessage;
  // import org.springframework.ai.chat.prompt.Prompt;

  // SystemMessage systemMessage = new SystemMessage("Your persona description here...");
  // UserMessage userMessage = new UserMessage("User's query...");
  // Prompt prompt = new Prompt(List.of(systemMessage, userMessage));
  // ChatResponse response = chatClient.call(prompt);
  ```
- Streaming Responses:
  For applications requiring real-time interaction, such as chatbots displaying responses as they are generated, Spring AI supports streaming. The stream() method on the ChatClient's prompt builder returns a Flux<ChatResponse>, allowing reactive handling of incoming response chunks.12

```java
// Example of streaming (conceptual, ensure reactive dependencies are present)
// Flux<ChatResponse> stream = chatClient.prompt()
//     .user("Tell me a long story.")
//     .stream()
//     .chatResponse();

// stream.subscribe(
//     chatResponse -> {
//         String contentChunk = chatResponse.getResult().getOutput().getContent();
//         if (contentChunk!= null) {
//             System.out.print(contentChunk);
//         }
//     },
//     error -> System.err.println("Error streaming response: " + error),
//     () -> System.out.println("\nStreaming complete.")
// );
```

- ChatOptions:
  Model-specific parameters like temperature (randomness of output), max tokens (response length limit), and the specific model version can be configured using ChatOptions. These can be set globally in application.properties (e.g., spring.ai.openai.chat.options.model=gpt-4o-mini, spring.ai.openai.chat.options.temperature=0.7 8) or on a per-request basis using the ChatOptions.builder().14

```java
// Example of per-request ChatOptions
// import org.springframework.ai.chat.prompt.ChatOptionsBuilder;

// ChatResponse response = chatClient.prompt()
//     .user("Generate a creative poem.")
//     .options(ChatOptionsBuilder.builder()
//         .withModel("gpt-4o") // Overrides default if different
//         .withTemperature(0.9f)
//         .withMaxTokens(150)
//         .build())
//     .call();
```

- Error Handling:
  When interacting with external AI model APIs, network issues, API rate limits, or other errors can occur. Standard Spring error handling mechanisms (try-catch blocks, @ControllerAdvice for web applications) should be employed. Spring AI itself might throw specific exceptions related to API interactions, which can be caught and handled appropriately. More advanced observability and retry

mechanisms will be discussed later.

3.2 Deep Dive: EmbeddingClient for Semantic Understanding

The EmbeddingClient is the Spring AI interface for converting textual (and potentially other types of) data into numerical vector embeddings.[5] These embeddings capture the semantic meaning of the input, forming the backbone of many AI features like semantic search, RAG, classification, and clustering.

- Purpose and Usage:
  The primary role of the EmbeddingClient is to interact with an embedding model to generate these vector representations.
  - **Injection:** An EmbeddingClient bean is typically auto-configured by Spring AI if an appropriate starter (e.g., spring-ai-openai-starter, spring-ai-ollama-starter) is present and configured. It can then be injected into your services.
  - **Core Methods:**
    - embed(String text): Generates an embedding for a single string.
    - embed(Document document): Generates an embedding for the content of a Spring AI Document object.
    - embed(List<String> texts): Generates embeddings for a list of strings in a batch, which can be more efficient. A simple usage example [9]:

```java
// import org.springframework.ai.embedding.EmbeddingClient;
// import java.util.List;

// @Service
// public class MyEmbeddingService {
//     private final EmbeddingClient embeddingClient;

//     public MyEmbeddingService(EmbeddingClient embeddingClient) {
//         this.embeddingClient = embeddingClient;
//     }

//     public List<Double> getEmbeddingForText(String text) {
//         return embeddingClient.embed(text);
//     }
// }
```

- Configuration:
  Similar to chat models, embedding models need to be configured.
  - **Model Selection:** Different providers offer various embedding models (e.g., OpenAI's text-embedding-3-small, text-embedding-3-large, text-embedding-ada-002; Ollama can serve models like nomic-embed-text [13]

    or mxbai-embed-large). Spring AI also supports using local (ONNX) Transformer models for embeddings.[5]
  - **Properties:** Configuration is typically done via application.properties, for example:
    ```
    Properties
    # For OpenAI
    spring.ai.openai.embedding.options.model=text-embedding-3-small

    # For Ollama (assuming nomic-embed-text is pulled)
    spring.ai.ollama.embedding.options.model=nomic-embed-text
    ```

- Use Cases:
  While EmbeddingClient itself just generates vectors, these vectors are crucial inputs for:
  - **Semantic Search:** Finding documents or items similar in meaning to a query, not just keyword matches.
  - **Retrieval Augmented Generation (RAG):** Retrieving relevant context to feed into an LLM prompt.
  - **Classification/Clustering:** Grouping similar items or classifying new items based on their semantic properties.

3.3 Effective Prompt Engineering Techniques with Spring AI

As highlighted in [15], "Prompt engineering is crucial for effectively interacting with large language models (LLMs) to ensure accurate and useful results." While Spring AI provides the mechanisms for sending prompts, the content and structure of those prompts significantly impact the quality of the LLM's output.

- **Key Techniques with Spring AI Examples:**
  - **Zero-Shot Prompting:** This involves asking the LLM to perform a task without providing any explicit examples in the prompt. The model relies on its pre-trained knowledge. This is suitable for straightforward tasks where the model likely encountered similar patterns during its training.[14]
    ```
    Java
    // Example from [14] (adapted)
    // import org.springframework.ai.chat.client.ChatClient;
    // import org.springframework.ai.chat.prompt.ChatOptionsBuilder;

    // public enum Sentiment { POSITIVE, NEUTRAL, NEGATIVE }

    // public Sentiment classifyReview(ChatClient chatClient, String reviewText) {
    //     return chatClient.prompt()
    //         .user("Classify the sentiment of the following movie review. Respond with only
    ```

```
POSITIVE, NEUTRAL, or NEGATIVE.\nReview: \"" + reviewText + "\"\nSentiment:")
//       .options(ChatOptionsBuilder.builder() // Example options
//           .withModel("gpt-4o-mini") // Or your configured model
//           .withTemperature(0.1f) // Low temperature for deterministic classification
//           .withMaxTokens(5)      // Limit tokens for a single word response
//           .build())
//       .call()
//       .entity(Sentiment.class); // Directly maps to enum
// }
```

- **Few-Shot Prompting (and One-Shot):** This technique involves providing one (One-Shot) or more (Few-Shot, typically 3-5) examples of input-output pairs within the prompt. This helps the LLM understand the desired format, style, or reasoning pattern, especially for complex or ambiguous tasks.[14]

```java
// Example adapted from [14] for pizza order parsing
// String pizzaOrderPrompt = """
// Parse the customer's pizza order into valid JSON.

// EXAMPLE 1:
// Customer: I want a small pizza with cheese, tomato sauce, and pepperoni.
// JSON Response:
// ```json
// {
//   "size": "small",
//   "type": "normal",
//   "ingredients": ["cheese", "tomato sauce", "pepperoni"]
// }
// ```

// EXAMPLE 2:
// Customer: Can I get a large pizza with tomato sauce, basil and mozzarella.
// JSON Response:
// ```json
// {
//   "size": "large",
//   "type": "normal",
//   "ingredients": ["tomato sauce", "basil", "mozzarella"]
// }
// ```

// Now, parse the following order:
// Customer: I would like a medium pizza, half olives and mushrooms, half just extra cheese.
// JSON Response:
// """;
// String jsonResponse = chatClient.prompt(pizzaOrderPrompt)
```

```
//    .call()
//    .content();
```

Few-shot prompting is particularly effective when specific formatting is required or when dealing with nuanced tasks.[14]

- **System Prompts:** Using SystemMessage to set the overall context, define the AI's persona, specify constraints, or provide high-level instructions that apply to the entire conversation or interaction.[2]
- **Contextual Prompting:** Including relevant background information or data directly within the prompt to help the LLM generate a more informed response.[15] RAG is an advanced form of this.
- **Role Prompting:** Assigning a specific role to the AI (e.g., "You are an expert Java architect reviewing code snippets.") can significantly influence the tone and content of its responses.[15] This is often done via a SystemMessage.

- Prompt Templates:
  For dynamic prompts where parts of the text need to be filled in at runtime, Spring AI offers PromptTemplate. By default, it uses the StringTemplate engine for rendering.16

```
Java
// import org.springframework.ai.chat.prompt.PromptTemplate;
// import org.springframework.ai.chat.prompt.Prompt;
// import java.util.Map;

// PromptTemplate promptTemplate = new PromptTemplate("Tell me a {adjective} joke about
{topic}.");
// Map<String, Object> vars = Map.of("adjective", "funny", "topic", "programmers");
// Prompt renderedPrompt = promptTemplate.create(vars);
// String joke = chatClient.call(renderedPrompt).getResult().getOutput().getContent();
```

  This is useful for creating reusable prompt structures.[17]

- **Best Practices for Prompt Engineering** [15]**:**
  - **Be Specific:** Clearly define the task, expected output format, and any constraints.
  - **Provide Clear Examples:** For few-shot prompting, ensure examples are accurate and representative.
  - **Iterate and Refine:** Prompt engineering is often an iterative process. Test different phrasings, structures, and examples to see what yields the best results.
  - **Control Temperature and Max Tokens:** Adjust ChatOptions like temperature (for creativity vs. determinism) and max tokens (for response length) to suit the task.[15]

The ability of an LLM to perform a task effectively is profoundly influenced by the quality and clarity of the prompts it receives. While Spring AI furnishes the necessary tools and abstractions for interacting with these models—such as PromptTemplate for dynamic content, SystemMessage for setting behavioral context, and straightforward mechanisms for structuring few-shot examples [14]—the actual skill in crafting the content of these prompts remains with the developer. Understanding the principles of prompt engineering, such as providing clear instructions, relevant context, illustrative examples, and defining an appropriate persona for the AI [15], is critical. Therefore, mastering Spring AI's APIs must be complemented by a growing proficiency in prompt engineering to construct truly effective and intelligent AI applications. The framework acts as a facilitator, streamlining the technical integration, but it does not replace the nuanced art and science of communicating effectively with LLMs.

3.4 Handling Structured Outputs (POJO Mapping)

A powerful feature of Spring AI is its ability to directly map LLM responses to Plain Old Java Objects (POJOs). This simplifies integration into Java applications by providing type safety and eliminating manual parsing of JSON or text responses.[5]

- **How It Works:**
  1. **Prompt for Formatted Output:** The LLM is instructed via the prompt to return its response in a specific format, typically JSON, that matches the structure of the target POJO.
  2. **Spring AI Deserialization:** Use the entity(Class<T> type) or entity(ParameterizedTypeReference<T> type) method on the ChatClient's call() result. Spring AI, in conjunction with underlying libraries like Jackson, will then attempt to deserialize the LLM's JSON output into an instance of the specified POJO.

```java
// Define a POJO (Java Record for conciseness)
public record Book(String title, String author, int publicationYear, List<String> genres) {}

// Prompt the LLM to return JSON matching the Book structure
String bookQueryPrompt = """
Provide details for a fictional book titled 'The Art of AI Whispering'.
The author is 'Dr. Elara Vance', publication year is 2025.
Genres should include 'Technology', 'Artificial Intelligence', and 'Humor'.
Return the information as a JSON object with fields: "title", "author", "publicationYear", and
"genres" (as an array of strings).
JSON Response:
""";

// Use.entity() to map to the POJO
// Book bookDetails = chatClient.prompt(bookQueryPrompt)
//                       .call()
//                       .entity(Book.class);

// System.out.println("Book Title: " + bookDetails.title());
// System.out.println("Author: " + bookDetails.author());
```

For lists of objects, a ParameterizedTypeReference is used [10]:Java

```
// import org.springframework.core.ParameterizedTypeReference;
// List<Book> bookList = chatClient.prompt("Generate a list of 3 fictional sci-fi books as a JSON
array...")
//                       .call()
//                       .entity(new ParameterizedTypeReference<List<Book>>() {});
```

The capability to directly map LLM outputs to Java POJOs represents a significant enhancement for Java developers integrating AI. Traditionally, interactions with LLMs often yield responses as raw text or unstructured/semi-structured JSON strings. In a typical Java application environment that heavily relies on strongly-typed objects, developers would need to write boilerplate code to parse these string outputs, validate them, and then manually map them to their domain objects. This manual process is not only tedious but also prone to errors, especially if the LLM's output format varies slightly. Spring AI's .entity(MyPojo.class) feature [5] effectively automates this deserialization, provided the LLM is correctly prompted to produce output in a compatible JSON structure. This automation makes the consumption of AI-generated data as seamless and type-safe as consuming data from any other well-defined API or data source within a Spring application. Consequently, it greatly boosts developer productivity, reduces the likelihood of runtime parsing errors, and improves the overall robustness of the AI integration.

3.5 Working with Various AI Models (Hugging Face Integration Example)

Spring AI is designed for portability, supporting a wide array of AI model providers.[5] Hugging Face, with its vast collection of open-source models accessible via Inference Endpoints or the Inference API, is a valuable resource for developers.[11]

- **Hugging Face Integration with Spring AI:**
    - **Overview:** Spring AI allows integration with models hosted on Hugging Face, typically through their managed Inference Endpoints.
    - **Setup:**
        1. **Hugging Face Account & Resources:** A Hugging Face account is needed. For Inference Endpoints, you'll deploy a model and get an endpoint URL. An API token is required for authentication.[11]
        2. **Dependency:** Add the Spring AI starter for Hugging Face. The artifact ID might be spring-ai-starter-model-huggingface or

spring-ai-huggingface-spring-boot-starter.[11]

```xml
// Maven
<dependency>
   <groupId>org.springframework.ai</groupId>
   <artifactId>spring-ai-starter-model-huggingface</artifactId>
</dependency>
```

3. **Configuration:** Set the API key and inference endpoint URL in application.properties [11]:

```properties
spring.ai.huggingface.chat.api-key=${HUGGINGFACE_API_KEY}
spring.ai.huggingface.chat.url=${HUGGINGFACE_INFERENCE_ENDPOINT_URL}
# Optionally, specify model options if supported by the endpoint/model
# spring.ai.huggingface.chat.options...
```

- **Usage:** If using Spring Boot auto-configuration, a HuggingfaceChatModel bean will be created and can be injected. This can then be used with a ChatClient.Builder.

```java
// import org.springframework.ai.huggingface.HuggingfaceChatModel;
// import org.springframework.ai.chat.client.ChatClient;

// @Service
// public class HuggingFaceService {
//     private final ChatClient chatClient;

//     public HuggingFaceService(HuggingfaceChatModel hfChatModel) { // Auto-configured
//         this.chatClient = ChatClient.builder(hfChatModel).build();
//     }

//     public String getHuggingFaceResponse(String message) {
//         return chatClient.prompt().user(message).call().content();
//     }
// }
```

Manual configuration of HuggingfaceChatModel is also possible if not using auto-configuration.[11]

- **Model Selection:** Hugging Face hosts a vast number of models for various tasks including text generation, text-to-image, etc..[18] Ensure the selected Inference Endpoint is configured with a model suitable for chat/text generation tasks when using HuggingfaceChatModel. Some tutorials might show direct WebClient usage for Hugging Face APIs [18]; however, for consistency within this roadmap, the focus is on the Spring AI-provided abstractions like HuggingfaceChatModel.

3.6 Hands-on Project Proposals for Stage 2

These projects aim to provide practical experience with the core APIs and techniques covered in this stage.

- **Project 1: Advanced Q&A Bot with Prompt Engineering**
  - **Description:** Create a Spring Boot application that functions as a Q&A bot on a specific, narrow topic (e.g., "History of the Java Programming Language" or "Principles of Object-Oriented Design"). Implement various prompt engineering techniques to refine the bot's answers. For instance, use system prompts to define its persona (e.g., "You are a patient and knowledgeable historian of Java"). Use few-shot examples to guide its response format for certain types of questions. Allow users (perhaps via different endpoints or parameters) to see responses generated using different prompting strategies.
  - **Learning Objectives:** Advanced ChatClient usage, practical application of zero-shot, few-shot, and system prompts, use of PromptTemplate for dynamic query formulation.
  - **Key Steps:**
    1. Define distinct personas or response styles for the bot.
    2. Create PromptTemplate instances for different question types or styles.
    3. Use SystemMessage to set the persona.
    4. Construct prompts that include few-shot examples for complex queries.
    5. Develop REST endpoints to trigger different prompting strategies and compare the outputs.
- **Project 2: Text Summarizer and Analyzer with Structured Output**
  - **Description:** Build an application that accepts a significant block of text as input. The application should use an LLM to:
    1. Generate a concise summary of the text.
    2. Extract key entities (e.g., names of people, organizations, locations mentioned).
    3. Determine the overall sentiment of the text (e.g., positive, negative, neutral). The final output of this analysis should be a structured Java POJO containing the summary, a list of entities, and the sentiment.
  - **Learning Objectives:** Using ChatClient for multiple analytical tasks (summarization, entity extraction, sentiment analysis), crafting prompts to elicit JSON or structured responses, and mastering POJO mapping with .entity().

- **Key Steps:**
    1. Design a Java POJO (e.g., TextAnalysisResult with fields for summary, List<Entity>, sentiment).
    2. Craft distinct prompts for summarization, entity extraction (specifying the types of entities), and sentiment analysis, instructing the LLM to format its output as JSON compatible with your POJO. This might involve multiple LLM calls or a complex single prompt.
    3. Use chatClient.prompt(...).call().entity(TextAnalysisResult.class) to get the structured output.

- **Project 3: Multi-Model Playground**
  - **Description:** Extend the "Local LLM Query Tool" from Stage 1. This enhanced version should allow users to send a query and have it processed by different AI models: a locally running Ollama model, an OpenAI model, and a model accessed via a Hugging Face Inference Endpoint. Implement a mechanism (e.g., using Spring Profiles to activate different configurations, or a request parameter to dynamically select the ChatClient implementation) to switch between these configured models.
  - **Learning Objectives:** Integrating multiple AI providers within a single Spring AI application, understanding the specific configuration requirements for each (API keys, endpoint URLs, model names), and exploring basic strategies for dynamic ChatClient selection or configuration.
  - **Key Steps:**
    1. Configure credentials/endpoints for Ollama, OpenAI, and Hugging Face in application.properties (potentially using profile-specific properties).
    2. Create separate ChatClient beans or service components tailored for each provider if not relying solely on default auto-configuration switching.
    3. Develop a controller that can route user requests to the appropriate model interaction logic based on the selected provider.

Table 3: Prompt Engineering Techniques in Spring AI

Technique	**Description**	**Spring AI Implementation**	**Use Case Example**
Zero-Shot	Ask LLM to perform a task without examples, relying on its pre-trained	UserMessage with direct instruction. .entity(EnumType.class) for classification.[14]	Simple sentiment analysis, direct questions, basic translation.

	knowledge.[14]		
Few-Shot (One-Shot)	Provide 1 to N input-output examples in the prompt to guide format or reasoning.[14]	UserMessage containing examples before the actual query. Can involve AssistantMessage for example AI responses.	Parsing specific formats (e.g., JSON from text [14]), complex classification, style imitation.
System Prompt	Use SystemMessage to set AI's role, persona, constraints, or high-level instructions.[2]	SystemMessage object included in the Prompt's message list or via ChatClient.prompt().system(...).	Defining chatbot persona, setting output rules (e.g., "always be concise").
Role Prompting	Assign a specific role to the AI (e.g., "You are a senior software architect").[15]	Typically implemented via SystemMessage.	Code review, expert Q&A, generating text in a specific professional style.
Prompt Template	Use PromptTemplate for dynamic prompts with placeholders, rendered with runtime data.[16]	org.springframework.ai.chat.prompt.PromptTemplate class, create(Map<String, Object> model) method.	Generating personalized messages, dynamic query construction, reusable prompts.
Contextual Prompting	Including relevant background information or data directly within the prompt.[15]	Part of UserMessage or SystemMessage content. RAG is an advanced form.	Answering questions based on provided text, summarizing specific documents.

Section 4: Stage 3 - Building Data-Aware Applications with RAG

This stage focuses on one of the most impactful patterns in applied AI: Retrieval Augmented Generation (RAG). RAG empowers LLMs by connecting them to external knowledge sources, enabling them to generate more accurate, relevant, and contextually grounded responses.

4.1 Understanding Retrieval Augmented Generation (RAG)

LLMs, despite their vast training data, have inherent limitations: they don't possess knowledge of private, domain-specific, or real-time information, and they can sometimes "hallucinate" or generate plausible but incorrect facts.[2] Retrieval Augmented Generation (RAG) is a technique designed to mitigate these issues by grounding LLM responses in verifiable external data.[4]

- **How RAG Works:** The RAG process typically involves the following steps:
  1. **Query Reception:** The system receives a user's query or prompt.
  2. **Retrieval:** The query is used to search a knowledge base (often a vector store containing embeddings of documents or data snippets). The most relevant documents or pieces of information are retrieved based on semantic similarity to the query.
  3. **Augmentation:** The retrieved information (the "context") is then injected into a new prompt, along with the original user query.
  4. **Generation:** This augmented prompt is sent to an LLM, which uses both the original query and the provided context to generate a final response.
- **Benefits of RAG [2]:**
  - **Improved Accuracy:** Responses are based on factual data from the knowledge base, rather than solely on the LLM's parametric memory.
  - **Reduced Hallucinations:** Grounding responses in retrieved context makes the LLM less likely to invent information.
  - **Access to Current/Proprietary Data:** Allows LLMs to utilize information not present in their original training set, such as recent news, internal company documents, or product specifications.
  - **Transparency and Citability:** Since responses are based on retrieved documents, it's often possible to cite the sources, increasing user trust.

Spring AI provides robust support for implementing RAG workflows, recognizing its critical importance for building practical and reliable AI applications.[3]

RAG is pivotal for making LLMs truly valuable in enterprise settings. While generic LLMs can perform a variety of tasks, their lack of access to proprietary, up-to-date, or domain-specific enterprise data limits their direct applicability to many business problems.[2] Enterprises require AI solutions that can reason over their unique datasets—be it internal knowledge bases, product documentation, customer interaction histories, or financial records. RAG provides the essential mechanism to dynamically "feed" this private and relevant data to an LLM at the time of query.[5] Spring AI's significant emphasis on RAG, evident through its comprehensive VectorStore API, integrated ETL framework for data ingestion, and Advisors designed to streamline RAG implementation [4], positions it as a powerful toolset for Java developers. This focus ensures that developers can build AI applications that are not just generally intelligent but are specifically tailored and highly relevant to the unique context and data assets of their organizations.

4.2 The Role of Vector Stores in Spring AI

Vector stores are specialized databases crucial for the "Retrieval" step in RAG. They are designed to efficiently store, manage, and search through large quantities of vector embeddings.[21]

- Spring AI VectorStore API:
  Spring AI offers a VectorStore interface, which provides a consistent abstraction for interacting with various underlying vector database implementations.21 This portability is a key advantage.
  - **Key Methods:**
    - add(List<Document> documents): Adds a list of Document objects (which will be embedded by the EmbeddingClient if not already embeddings) to the vector store.[13]
    - delete(List<String> idList): Deletes documents by their IDs.[21]
    - similaritySearch(SearchRequest request): Performs a similarity search based on a query string or vector, returning the most similar documents.[13]
  - **Document Object:** Encapsulates the content (text) to be stored and searched, along with associated metadata (key-value pairs like filename, category, creation date, etc.).[13] Metadata is vital for filtering search results.
  - **SearchRequest:** Allows specification of the query, the number of top K results to retrieve, a similarity threshold (to filter results below a certain relevance score), and metadata filters.[13]
- Portable SQL-like Metadata Filtering:
  A standout feature of Spring AI's VectorStore API is its novel SQL-like metadata filter API.4 This allows developers to write complex metadata filtering expressions in a familiar syntax that is portable across different vector store implementations.
  Example using FilterExpressionBuilder 21:

  ```java
  // import org.springframework.ai.vectorstore.filter.FilterExpressionBuilder;
  // import org.springframework.ai.vectorstore.filter.Filter.Expression;

  // FilterExpressionBuilder b = new FilterExpressionBuilder();
  // Expression expression = b.eq("category", "technical_manual")
  //                        .and(b.gte("version", 2.0))
  //                        .build();
  // SearchRequest request = SearchRequest.query("find feature X")
  //                                .withFilterExpression(expression)
  ```

```
//                              .withTopK(5);
// List<Document> results = vectorStore.similaritySearch(request);
```

- Schema Initialization:
  Many vector stores require some form of schema or index initialization. Spring AI allows this to be automated via a configuration property, specific to each vector store provider (e.g., spring.ai.vectorstore.chroma.initialize-schema=true, spring.ai.vectorstore.redis.initialize-schema=true).7

The landscape of vector databases is diverse and rapidly evolving, with numerous options available, each possessing unique strengths and operational characteristics.[5] Traditionally, integrating with different vector databases would require developers to learn and implement against each database's specific API and proprietary query language, particularly for advanced operations like metadata filtering.[4] This not only increases development time but also leads to code that is tightly coupled to a particular vendor, hindering future flexibility. Spring AI's VectorStore abstraction significantly mitigates these challenges. By providing a common interface for fundamental operations like adding documents, deleting documents, and performing similarity searches [21], it simplifies the initial integration. More profoundly, the introduction of a portable, SQL-like metadata filter API [4] is a game-changer. It allows developers to define sophisticated metadata-based search criteria using a consistent syntax, which Spring AI then translates to the native query language of the underlying vector store. This dramatically reduces the learning curve associated with individual databases and makes RAG applications more maintainable and adaptable, as switching to a different vector store in the future would require minimal code changes related to these core interactions.

4.3 Integrating Vector Stores: Practical Examples

Spring AI provides starters and auto-configuration for many popular vector stores. The general integration pattern involves adding the specific starter dependency, configuring connection details in application.properties, and then injecting the VectorStore bean.

4.3.1 ChromaDB Integration

ChromaDB is an open-source embedding database designed for ease of use.

- **Setup:** Typically run using Docker.[13] For local development and testing, Testcontainers can manage the ChromaDB lifecycle.[13]
- **Dependency:** org.springframework.ai:spring-ai-chroma-store-spring-boot-starter.[13]

- **Configuration (application.properties):**
  ```
  Properties
  # For a local ChromaDB instance running via Docker on default port
  spring.ai.vectorstore.chroma.client.host=localhost
  spring.ai.vectorstore.chroma.client.port=8000
  spring.ai.vectorstore.chroma.collection-name=my_documents_collection # Optional, defaults to SpringAiCollection
  spring.ai.vectorstore.chroma.initialize-schema=true # Creates the collection if it doesn't exist

  # An embedding model must also be configured, e.g., Ollama
  spring.ai.ollama.embedding.options.model=nomic-embed-text
  spring.ai.ollama.base-url=http://localhost:11434 # Ensure Ollama is running
  ```
  ChromaDB stores embeddings, so an EmbeddingClient (like one configured for Ollama or OpenAI) must be available in the Spring application context.[13]

4.3.2 PGVector Integration

PGVector is a PostgreSQL extension that adds vector similarity search capabilities to a standard PostgreSQL database.

- **Setup:** Requires a PostgreSQL instance with the PGVector extension enabled. Docker is a common way to set this up (e.g., using the pgvector/pgvector image).[9]
- **Dependency:** org.springframework.ai:spring-ai-pgvector-store-spring-boot-starter.[22]
- **Configuration (application.properties):** Uses standard Spring Data JPA/JDBC datasource properties to connect to PostgreSQL, plus PGVector-specific properties.[9]
  ```
  Properties
  # Standard PostgreSQL connection
  spring.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
  spring.datasource.username=user
  spring.datasource.password=pass

  # PGVector specific
  spring.ai.vectorstore.pgvector.table-name=my_vector_store_table # Optional, defaults
  spring.ai.vectorstore.pgvector.dimensions=384 # Must match your embedding model's output dimensions (e.g., nomic-embed-text is 768, some OpenAI models are 1536)
  spring.ai.vectorstore.pgvector.index-type=HNSW # Or IVFFLAT
  spring.ai.vectorstore.pgvector.distance-type=COSINE # Or L2, INNER_PRODUCT
  spring.ai.vectorstore.pgvector.initialize-schema=true # Creates table and extension if not present

  # An embedding model must also be configured
  spring.ai.openai.embedding.options.model=text-embedding-3-small # Example, dimensions: 1536
  ```

4.3.3 Redis Integration

Redis, known for its speed as an in-memory data store, can also function as a vector store with modules like RediSearch.

- **Setup:** A Redis instance with vector search capabilities. Docker is convenient, and the Spring Initializr can even generate a compose.yml for Redis.[7]
- **Dependency:** org.springframework.ai:spring-ai-redis-vector-store-spring-boot-starter (or a similarly named starter, verify the exact artifact ID from Spring Initializr or documentation).
- **Configuration (application.properties):** Standard Redis connection properties, plus Spring AI specific properties for the vector store.[7]

  ```properties
  Properties
  # Standard Redis connection (if not using defaults or Docker Compose auto-config)
  # spring.data.redis.host=localhost
  # spring.data.redis.port=6379

  # Spring AI Redis Vector Store specific
  spring.ai.vectorstore.redis.index-name=my_app_index
  spring.ai.vectorstore.redis.prefix=doc_vectors: # Prefix for keys in Redis
  spring.ai.vectorstore.redis.initialize-schema=true # Creates the index if it doesn't exist

  # An embedding model must also be configured
  # spring.ai.openai.embedding.options.model=text-embedding-ada-002
  ```

4.4 Spring AI's Document Ingestion ETL Framework

To effectively use RAG, data must first be processed and loaded into a vector store. Spring AI includes a lightweight, configurable ETL (Extract, Transform, Load) framework to streamline this data preparation pipeline.[4]

- **Components (Conceptual, based on [4]):**
  - **Readers (ResourceReader implementations):** These components are responsible for extracting data from various sources.
    - JsonReader: For reading data from JSON files or resources. It can be configured to extract specific fields to map to Document content and metadata.[7]
    - TextReader: For plain text files.
    - PDF Readers (often based on Apache Tika): For extracting text content from PDF documents.

        - Other specialized readers for various document formats (e.g., Word, Markdown).
    - **Transformers (DocumentTransformer implementations):** These modify or enhance Document objects before they are stored.
        - **Document Splitters:** Large documents often need to be split into smaller chunks to fit within LLM context windows and improve retrieval relevance. Splitting can be done by token count, sentence boundaries, paragraph breaks, or custom strategies.
        - **Metadata Enrichers:** Add or modify metadata associated with documents (e.g., adding timestamps, source URIs, categories).
        - Embedding Transformers (though embedding is often handled by VectorStore.add() implicitly using the configured EmbeddingClient).
    - **Writers (VectorStoreWriter):** These components are responsible for loading the processed Document objects into the target VectorStore.
- Example Usage:
  The JsonReader used in the Redis integration example for loading beer data demonstrates a simple ETL step 7:

```java
// From [7] (adapted for clarity)
// import org.springframework.ai.reader.JsonReader;
// import org.springframework.ai.document.Document;
// import org.springframework.core.io.ClassPathResource;
// import java.util.List;

// Resource jsonDataResource = new ClassPathResource("/data/beers.json");
// String jsonKeysToExtract = {"name", "style", "brewery", "description"}; // Fields for Document metadata or content
// JsonReader jsonReader = new JsonReader(jsonDataResource, jsonKeysToExtract);
// List<Document> documents = jsonReader.get();
// vectorStore.add(documents); // VectorStore handles embedding and storage
```

  More complex ETL pipelines can be constructed by chaining multiple readers and transformers.

4.5 Implementing RAG with Advisors

Spring AI's Advisors API provides an elegant way to implement RAG by intercepting ChatClient calls and modifying the prompt, typically by injecting retrieved documents as context.[3]

- QuestionAnswerAdvisor:
  This advisor is specifically designed for Q&A scenarios using RAG. It takes a VectorStore instance in its constructor. When a prompt is processed, it uses the

user's query to search the VectorStore for relevant documents and then adds these documents to the context of the prompt sent to the LLM.7
It's typically added to the ChatClient.Builder's default advisors:

```java
// import org.springframework.ai.vectorstore.VectorStore;
// import org.springframework.ai.chat.client.advisor.QuestionAnswerAdvisor;
// import org.springframework.ai.chat.client.ChatClient;

// VectorStore myVectorStore =... ; // Injected
// ChatClient.Builder chatClientBuilder =... ; // Injected

// ChatClient ragChatClient = chatClientBuilder
//                        .defaultAdvisors(new QuestionAnswerAdvisor(myVectorStore))
//                        .build();

// String userAnswer = ragChatClient.prompt()
//                         .user("What are the benefits of using RAG?")
//                         .call()
//                         .content();
```

- RetrievalAugmentationAdvisor:
  This is a more generic and configurable advisor for RAG. It can also incorporate advanced techniques like query rewriting (using an LLM to rephrase the user's query for better retrieval results) before hitting the vector store.20

Advisors significantly simplify the implementation of RAG logic, allowing developers to focus on data preparation and prompt design rather than low-level prompt manipulation.

4.6 Hands-on Project Proposals for Stage 3

These projects will provide practical experience in building RAG-enabled applications.

- **Project 1: "Chat with Your PDF" Application**
  - **Description:** Develop a Spring Boot application that allows a user to upload a PDF document. The application should then:
    1. Extract text content from the PDF (using a library like Apache Tika, which Spring AI might use under the hood for some readers).
    2. Split the extracted text into manageable chunks.
    3. Generate embeddings for each chunk using a configured EmbeddingClient.
    4. Store these embeddings and their corresponding text chunks (as Document objects) in a chosen vector store (e.g., ChromaDB or

PGVector).
5. Provide an interface (web or command-line) where the user can ask questions about the content of the uploaded PDF. The application should use a RAG approach with QuestionAnswerAdvisor to answer these questions.

- **Learning Objectives:** Document loading and processing (basic ETL), text splitting strategies, embedding generation, populating a vector store, implementing a full RAG pipeline with QuestionAnswerAdvisor.
- **Key Steps:**
  1. Integrate a PDF parsing library or use Spring AI's document readers if available for PDFs.
  2. Implement logic for splitting text into chunks (e.g., by paragraph or a fixed number of tokens).
  3. Use EmbeddingClient to generate embeddings for each chunk.
  4. Use VectorStore.add() to store the Document objects.
  5. Configure ChatClient with QuestionAnswerAdvisor pointing to your vector store.
  6. Build a simple UI or CLI for PDF upload and Q&A.

- **Project 2: Semantic Search Engine for Product Documentation**
  - **Description:** Gather a collection of product documentation (e.g., as markdown files, plain text files, or scraped web pages). Ingest this documentation into a vector store, creating Document objects with relevant metadata (e.g., product name, document section, version). Implement a semantic search API endpoint that accepts a natural language query from the user and returns the most relevant document sections from the vector store, potentially using metadata filters. Optionally, extend this to use an LLM to synthesize a concise answer based on the top N retrieved sections.
  - **Learning Objectives:** ETL pipeline for multiple files/sources, creating Document objects with rich metadata, advanced VectorStore.similaritySearch() usage including metadata filtering, and optionally, LLM-based answer synthesis from retrieved context.
  - **Key Steps:**
    1. Implement file readers for your documentation format(s).
    2. Define a strategy for chunking documents and extracting/creating metadata.
    3. Populate the vector store using VectorStore.add().
    4. Create a REST endpoint that takes a search query and optional filter parameters.
    5. Use VectorStore.similaritySearch() with a SearchRequest incorporating

the query and filters.
6. (Optional) Pass the content of the top search results to a ChatClient to generate a synthesized answer.

Table 4: Comparison of Featured Vector Stores for Spring AI

Vector Store	**Type**	**Key Features/Strengths**	**Spring AI Setup Highlights (Dependencies /Configs)**	**Ideal Use Cases**
ChromaDB	Standalone, open-source embedding database	Easy to set up and use, good for local development and smaller projects, Python-native but accessible via HTTP client.[13]	spring-ai-chroma-store-spring-boot-starter, configure host/port, collection name, initialize-schema. Requires an EmbeddingClient bean.[13]	Rapid prototyping, local development, projects where a separate, lightweight vector DB is preferred.
PGVector	PostgreSQL Extension	Leverages existing PostgreSQL infrastructure, supports transactional data alongside vectors, mature SQL capabilities for metadata.[9]	spring-ai-pgvector-store-spring-boot-starter, standard Spring DataSource config, dimensions, index-type, initialize-schema. Requires an EmbeddingClient bean.[22]	Applications already using PostgreSQL, scenarios needing strong consistency between vector and relational data.
Redis	In-memory data store (can persist)	High speed, versatile (caching, messaging, vector store with RediSearch), good for low-latency	spring-ai-redis-vector-store-spring-boot-starter (verify name), standard Redis config, index-name, prefix,	Applications requiring very fast retrieval, systems already using Redis for other purposes, caching embeddings.

		retrieval.[7]	initialize-schema. Requires an EmbeddingClient bean.[7]	

Section 5: Stage 4 - Enhancing AI Applications with Advanced Interactions

With a solid understanding of RAG and core Spring AI APIs, this stage explores more advanced interaction patterns. These include enabling LLMs to call external tools (function calling), managing conversational context effectively with chat memory, and working with multimodal AI models that can process more than just text.

5.1 Leveraging Function Calling / Tool Usage for External Actions

Function calling, also referred to as tool usage, is a powerful capability that allows LLMs to go beyond text generation by requesting the execution of predefined, client-side functions (your application's code).[2] This enables LLMs to access real-time information, interact with external APIs, query databases, or perform virtually any action your application can execute.

- **How Function Calling Works with Spring AI:**
  1. **Define Tools:** Java methods within your Spring application (often in Spring beans) are designated as "tools" that the LLM can invoke.
  2. **Describe Tools to LLM:** Information about these tools—their names, a description of what they do, and the parameters they expect—is provided to the LLM as part of the prompt's options. This allows the LLM to understand when and how to use them.
  3. **LLM Requests Tool Execution:** If the LLM determines that calling one of these tools is necessary to fulfill the user's request, its response will not be a direct answer but a special message indicating which tool to call and with what arguments.
  4. **Application Executes Tool:** Spring AI (often automatically for Spring bean tools) or your custom logic intercepts this request, finds the corresponding Java method, and executes it with the arguments provided by the LLM.
  5. **Return Result to LLM:** The return value (output) of the executed Java function is then sent back to the LLM in a subsequent request.
  6. **LLM Generates Final Response:** The LLM uses the information obtained from the tool's execution to formulate its final response to the user.
- **Spring AI Implementation Details:**
  - **Tool Definition:** Typically, methods in Spring @Service or @Component

    beans can serve as tools. They should have clear names, parameters, and return types.
  - **Declaring Functions to LLM:** This is done using FunctionCallingOptions (a generic Spring AI option) or model-specific options (like OpenAiChatOptions) when building the ChatClient prompt.[10] You register the names of the functions (tools) that the LLM should be aware of.
    ```java
    // Conceptual example of registering function names
    // ChatClient chatClient = chatClientBuilder
    //    .defaultOptions(OpenAiChatOptions.builder()
    //        .withFunction("getCurrentWeather") // Name of your Java method/tool
    //        .withFunction("getStockPrice")
    //        .build())
    //    .build();
    ```
  - **Tool Callbacks:** Spring AI provides ToolCallback interfaces. For functions that are Spring beans and whose names match those declared to the LLM, Spring AI can often automatically find and execute these methods. The spring.ai.openai.chat.options.tool-mode=AUTO property (or similar for other providers) can enable this.
  - **Model Context Protocol (MCP):** For more complex scenarios, especially involving remote tool execution or inter-service communication for tools, Spring AI integrates with the Model Context Protocol (MCP). This involves components like McpSyncClient and MethodToolCallbackProvider.[23]
- **Use Cases:**
  - Fetching real-time weather information for a given location.
  - Retrieving current stock prices for a specified ticker symbol.[24]
  - Querying a relational database for specific customer order details.
  - Sending an email or notification.
  - Interacting with any internal enterprise API to fetch data or trigger business processes.

Function calling dramatically expands the capabilities of LLMs, transforming them from passive information recall systems into active agents that can interact with and manipulate their environment. Standard LLMs are typically confined to the knowledge present in their training data and can only generate textual outputs.[2] Function calling effectively breaks these models out of their isolated "sandbox" by allowing them to request the execution of external code—the "tools" you define in your Spring application.[5] These tools can interface with any system or API that your Java application has access to, be it public web services, internal databases, or proprietary enterprise systems. Consequently, an LLM, orchestrated through Spring AI, can effectively "use" other software and services to gather up-to-the-minute data, perform calculations beyond its intrinsic abilities, or trigger real-world actions. This paradigm shift means that AI-powered applications can evolve from being purely informational (e.g., answering questions based on static knowledge) to being highly interactive and operational, driven by the intelligent decision-making of an LLM that can now delegate tasks to specialized functions.

5.2 Managing Conversational Context with Chat Memory

By default, LLMs are stateless, meaning each interaction is treated as independent, with no inherent memory of previous exchanges.[2] For applications like chatbots or virtual assistants, this is a significant limitation, as users expect a coherent conversation where the AI remembers what was discussed earlier. Spring AI provides ChatMemory features to address this.

- **Spring AI ChatMemory Abstractions** [5]**:**
  - **ChatMemory Interface:** The core abstraction for managing conversation history. Implementations decide which messages to retain and when to evict older ones.
  - **ChatMemoryRepository Interface:** Responsible for the underlying storage and retrieval of chat messages. Spring AI provides several implementations:
    - InMemoryChatMemoryRepository: Default implementation, stores messages in a ConcurrentHashMap in memory. Suitable for single-session or development scenarios.[25]
    - JdbcChatMemoryRepository: Stores chat history in a relational database via JDBC. Requires a database schema and dialect configuration.[25]
    - CassandraChatMemoryRepository: For storing chat history in Apache Cassandra.[25]
    - Other implementations may exist for Neo4j, MongoDB, etc.
  - **Memory Management Strategies:**
    - MessageWindowChatMemory: The most common strategy. It maintains a sliding window of the most recent messages, up to a specified maximum number (default is 20 messages). When the limit is exceeded, older messages are removed, typically while preserving any initial system messages.[25]
- Advisors for Chat Memory Integration 25:
  ChatMemory is often integrated into ChatClient interactions using advisors:
  - **MessageChatMemoryAdvisor:** On each interaction, this advisor retrieves the relevant conversation history from the configured ChatMemory and includes it in the prompt as a collection of Message objects (User, Assistant,

System). This is generally the preferred way as it aligns with how models like ChatGPT are typically fine-tuned.
    - **PromptChatMemoryAdvisor:** This advisor retrieves the history and appends it to the system prompt as plain text. This might be suitable for some models or specific prompting strategies but requires the system prompt template to have placeholders for memory content.
- Configuration:
  Spring AI auto-configures a ChatMemory bean by default, using an InMemoryChatMemoryRepository and MessageWindowChatMemory if no other ChatMemoryRepository bean is found in the context.25 This can be customized:

```java
// Example of customizing ChatMemory
// import org.springframework.ai.chat.memory.ChatMemory;
// import org.springframework.ai.chat.memory.MessageWindowChatMemory;
// import org.springframework.ai.chat.memory.store.InMemoryChatMemoryRepository;
// import org.springframework.context.annotation.Bean;
// import org.springframework.context.annotation.Configuration;

// @Configuration
// public class AiConfig {
//     @Bean
//     public ChatMemory chatMemory() {
//         return MessageWindowChatMemory.builder()
//                 .withChatMemoryStore(new InMemoryChatMemoryRepository()) // Or inject a persistent
one
//                 .withMaxTokenCount(1000) // Example: limit by token count instead of message count
//                 .build();
//     }
// }
```

  Then, this custom ChatMemory bean can be used with MessageChatMemoryAdvisor.

The expectation from users interacting with chatbots or AI assistants is that the system will remember the flow of the conversation—what was said previously, what topics were discussed, and any relevant information exchanged [Implicit]. However, the underlying LLMs powering these interactions are fundamentally stateless; each API call is processed in isolation without inherent knowledge of prior calls.[2] Manually managing this conversational history—collecting all user and assistant messages, deciding which ones to include in the context for subsequent prompts (while being mindful of token limits), and formatting them correctly—can become quite complex. Spring AI's ChatMemory abstraction, along with its repository implementations (like

InMemoryChatMemoryRepository or JdbcChatMemoryRepository) and windowing strategies (like MessageWindowChatMemory), greatly simplifies this task.[25] Furthermore, advisors such as MessageChatMemoryAdvisor seamlessly integrate this managed memory into the ChatClient's request-response cycle.[25] This means developers can add sophisticated and persistent conversational context to their AI applications with minimal boilerplate code, leading to a significantly improved and more natural user experience in multi-turn dialogues.

5.3 Exploring Multimodality: Image and Audio Models

Multimodality refers to AI models that can process, understand, or generate information across multiple types of data (modalities), such as text, images, and audio. Spring AI provides support for several multimodal capabilities.

- **Spring AI Support for Multimodal Models [5]:**
    - **Text-to-Image Models:**
        - Interface: ImageModel.
        - Supported Providers: OpenAI (DALL·E), Azure OpenAI, Stability AI, and others.
        - Usage: Create an ImagePrompt with a text description and options like image dimensions (height, width), number of images to generate (N), and desired response format (e.g., URL to the image, or base64 encoded data).[10]

          ```java
          // import org.springframework.ai.image.ImageClient; // Or ImageModel directly
          // import org.springframework.ai.image.ImagePrompt;
          // import org.springframework.ai.image.ImageOptionsBuilder;
          // import org.springframework.ai.image.ImageResponse;

          // ImageClient imageClient =... ; // Injected
          // ImagePrompt imagePrompt = new ImagePrompt("A futuristic cityscape at sunset,
          synthwave style",
          //     ImageOptionsBuilder.builder()
          //       .withHeight(1024)
          //       .withWidth(1024)
          //       .withN(1) // Number of images
          //       .withResponseFormat("url") // or "b64_json"
          //       .build()
          // );
          // ImageResponse imageResponse = imageClient.call(imagePrompt);
          // String imageUrl = imageResponse.getResult().getOutput().getUrl();
          ```

    - **Audio Transcription API (Speech-to-Text):**

        - Interface: AudioTranscriptionModel (or similar, verify exact API).
        - Supported Providers: OpenAI (Whisper), Azure OpenAI.
        - Usage: Provide an audio file to get its text transcription.
    - **Text-To-Speech (TTS) API:**
        - Interface: SpeechModel (or similar).
        - Supported Providers: OpenAI.
        - Usage: Convert text input into spoken audio.
    - **Multimodal Inputs to Chat Models (e.g., Vision):** Some chat models, like Llava (accessible via Ollama) or GPT-4V (via OpenAI), can process images alongside text in a single prompt. Spring AI supports this by allowing Media objects (representing images, audio, etc.) to be included in a UserMessage.[10]

```java
// Example for sending an image with text to a multimodal chat model (e.g., Llava via Ollama)
// import org.springframework.ai.chat.messages.UserMessage;
// import org.springframework.ai.chat.messages.Media;
// import org.springframework.ai.chat.prompt.Prompt;
// import org.springframework.core.io.ClassPathResource;
// import org.springframework.util.MimeTypeUtils;

// Media imageMedia = new Media(MimeTypeUtils.IMAGE_PNG, new
ClassPathResource("images/my_image.png").getURL());
// UserMessage userMessageWithImage = new UserMessage(
//     "What objects are in this image, and what is the general scene?",
//     List.of(imageMedia)
// );
// Prompt prompt = new Prompt(List.of(userMessageWithImage));
// ChatResponse response = chatClient.call(prompt);
// String description = response.getResult().getOutput().getContent();
```

- **Considerations:**
    - **Model Availability:** Not all providers or models support all modalities.
    - **API Costs:** Multimodal interactions, especially image generation or processing, can be more expensive than text-only interactions.
    - **Specific Use Cases:** Choose multimodal capabilities based on clear application requirements (e.g., image generation for content creation, image understanding for visual Q&A, transcription for voice commands).

5.4 Hands-on Project Proposals for Stage 4

These projects will help you apply the advanced interaction patterns learned in this stage.

- **Project 1: AI Assistant with Real-time Data Access (Function Calling)**

  - **Description:** Create an interactive AI assistant (web or command-line) that can answer questions requiring real-time data. For example, it could fetch the current stock price for a given company symbol using a financial data API (like Twelve Data, mentioned in [24], or another free alternative) or get the current weather for a specified city using a weather API. The LLM should decide when to call these external tools.
  - **Learning Objectives:** Master the implementation of function calling/tool usage in Spring AI, define tools as Java methods, handle LLM requests for tool execution, and integrate with live external APIs.
  - **Key Steps:**
    1. Identify and get API keys for a free stock data API and a weather API.
    2. Create Spring service methods that call these external APIs (e.g., getStockPrice(String symbol), getCurrentWeather(String city)).
    3. Configure your ChatClient with FunctionCallingOptions, describing these methods as tools to the LLM.
    4. Implement logic to handle the LLM's function call requests, execute the Java methods, and return the results to the LLM for final response generation.
    5. Test with queries like "What's the current stock price for AAPL?" or "What's the weather like in London?".
- **Project 2: Contextual Customer Service Bot with Chat Memory**
  - **Description:** Develop a customer service chatbot for a fictional e-commerce website. The bot should be able to handle multi-turn conversations, remembering previous user statements and products discussed within the current session to provide more relevant and personalized assistance. For simplicity, use an in-memory ChatMemory store.
  - **Learning Objectives:** Implement and configure ChatMemory in Spring AI, use MessageChatMemoryAdvisor (or similar) to integrate memory into ChatClient interactions, and design conversational flows that leverage context.
  - **Key Steps:**
    1. Define a scope for the bot (e.g., order tracking, product inquiries, return requests).
    2. Configure an InMemoryChatMemoryRepository and MessageWindowChatMemory (or use Spring AI defaults).
    3. Integrate chat memory into your ChatClient using an appropriate advisor.
    4. Design prompts and conversation flows where remembering past interactions is beneficial (e.g., "You mentioned you were interested in laptops. Do you have a specific brand in mind?" or "Regarding your order #12345, what specific issue are you facing?").

    5. Test with multi-turn dialogues.
- **Project 3: Image Captioning and Q&A Application (Multimodality)**
  - **Description:** Build a Spring Boot application where users can upload an image. The application should then:
    1. Use a multimodal LLM (e.g., Llava running via Ollama, or GPT-4V if you have access) to generate a descriptive caption for the uploaded image.
    2. Allow the user to ask follow-up questions specifically about the content of that image. The image itself (and optionally the generated caption) should be provided as context to the LLM for answering these questions.
  - **Learning Objectives:** Work with multimodal inputs in Spring AI (using Media objects), interact with LLMs that have vision capabilities, and combine image understanding with textual Q&A.
  - **Key Steps:**
    1. Set up Ollama and pull a vision-capable multimodal model like "llava".
    2. Create an endpoint for image uploads.
    3. For captioning, send the image as Media in a UserMessage to the multimodal ChatClient, prompting for a description.
    4. For Q&A, send the image again (and optionally the caption) along with the user's question in a UserMessage to the ChatClient.
    5. Display the caption and answers to the user.

Section 6: Stage 5 - Towards Production-Ready and Agentic AI Solutions

This final stage of active development focuses on aspects crucial for building robust, reliable, and advanced AI applications. It covers evaluating AI model outputs, ensuring observability, introducing agentic patterns for more autonomous behavior, adhering to best practices, and addressing common challenges in AI development.

6.1 AI Model Evaluation: Ensuring Accuracy and Relevance

A significant challenge with LLMs is their potential to "hallucinate" (generate incorrect or nonsensical information) or produce responses that are irrelevant to the user's query or the provided context.[2] Model evaluation is essential for building trust and ensuring the quality of AI-generated content.

- Spring AI Evaluator Interface 4:
  Spring AI provides an Evaluator functional interface for assessing the quality of AI model responses. The evaluation process itself can leverage an AI model, potentially a different one specialized for evaluation tasks.
  - **EvaluationRequest:** Contains the inputs for evaluation:

        - userText: The original user input/query.
        - dataList (or context): Contextual data, such as documents retrieved in a RAG workflow.
        - responseContent: The AI model's response that needs to be evaluated.
    - **EvaluationResponse:** Contains the outcome of the evaluation, often a boolean (pass/fail) and/or a score or feedback.
- **Built-in Evaluators in Spring AI** [4]**:**
    - **RelevancyEvaluator:**
        - **Purpose:** Designed to assess whether an AI-generated response is relevant to the user's query, especially considering the context provided (e.g., in a RAG system). It helps determine if the RAG flow is working correctly by checking if the LLM's answer aligns with the retrieved documents.
        - **Mechanism:** Uses a customizable prompt template to ask an evaluation LLM if the main LLM's response is relevant given the query and context. The default prompt asks for a "YES" or "NO" answer.
    - **FactCheckingEvaluator:**
        - **Purpose:** Verifies the factual accuracy of an AI's response (the "claim") against a provided document or context. This is crucial for detecting and reducing hallucinations.
        - **Mechanism:** Presents the claim and the document to an evaluation LLM. Smaller, more efficient models dedicated to fact-checking, such as Bespoke's Minicheck (which can be used via Ollama), are recommended to reduce evaluation costs compared to using flagship models like GPT-4 for this task.
- Usage:
  Evaluators are particularly useful in:
    - **Integration Tests:** To automatically assess the quality of AI responses as part of your test suites.
    - **CI/CD Pipelines:** To continuously monitor AI quality and catch regressions.
    - **Benchmarking:** To compare the performance of different models or prompting strategies.

6.2 Observability in Spring AI: Metrics and Tracing

For production AI applications, understanding performance, identifying bottlenecks, monitoring costs (token usage), and troubleshooting issues are critical. Spring AI integrates with Micrometer to provide comprehensive observability.[4]

- Integration with Micrometer 4:

  Spring AI provides metrics and tracing capabilities for its core components: ChatClient (including its Advisor chain), ChatModel, EmbeddingModel, ImageModel, and VectorStore.
  - **Metrics:**
    - **Model Latency:** Measures the time taken for AI models to respond (e.g., gen_ai.client.request.duration).
    - **Token Usage:** Tracks the number of input and output tokens per request (e.g., gen_ai.client.token.usage). This is vital for monitoring and optimizing API costs associated with LLM usage.
    - **Tool Calls and Retrievals:** Provides insights into when and how often function calls (tools) are invoked or data is retrieved from vector stores. These metrics can be exported to monitoring systems like Prometheus and visualized in dashboards (e.g., Grafana).
  - **Tracing:** Full distributed tracing support is available via Micrometer Tracing. Spans are created for each major step in an AI model interaction, allowing developers to visualize the flow of requests and identify performance issues across different components (e.g., time spent in an advisor, in the actual model call, or in a vector store retrieval).
- Logging 26:
  Spring AI supports configurable logging to aid in debugging and troubleshooting.
  - **Content Logging:** It's possible to log the content of prompts, LLM completions, and vector store query responses. However, this should be used with caution, especially in production, due to the risk of exposing sensitive or private information contained within prompts or responses.
  - **Configuration Properties:** Properties like spring.ai.chat.client.observations.log-prompt (renamed from older versions) control whether prompt content is logged. Similar properties exist for completions and other components.

Effective observability is non-negotiable for taking AI applications from prototype to production. While a basic AI application might simply make calls to an LLM, an enterprise-grade AI system must be reliable, its performance monitorable, and its outputs trustworthy. Spring AI directly addresses these operational concerns by deeply integrating with the Spring ecosystem's observability tools. The Evaluator framework provides mechanisms for quality control and assessing the factuality and relevance of AI responses.[4] Micrometer integration delivers detailed telemetry, including metrics on request latency, crucial token usage data for cost management, and tracing information to diagnose complex interaction flows.[4] These features are not mere add-ons but are designed as integral parts of the framework. This comprehensive support for evaluation and observability significantly accelerates the development of AI applications that are not just functionally capable but are also robust, manageable, and ready for the rigors of production deployment.

6.3 Introduction to Agentic Patterns with Spring AI

Agentic AI systems represent a step towards more autonomous AI, where LLMs can not only respond to queries but also plan, decompose tasks, use tools, and iteratively refine their approach to achieve complex goals.

- **Workflows vs. Agents** [27]**:**
    - **Workflows:** Structured sequences where LLMs and external tools follow predefined execution paths. They offer predictability and control.
    - **Agents:** More dynamic and autonomous systems where the LLM itself dictates the process, selecting tools and determining the sequence of actions needed to accomplish a task.
- **Key Agentic Patterns supported or buildable with Spring AI** [4]**:**
    - **Chain Workflow:** Tasks are broken down into sequential steps, with the output of one step feeding into the next. Useful for tasks requiring a linear progression of logic or refinement.
    - **Parallelization Workflow:** Multiple tasks or sub-problems are processed concurrently by LLMs, and their outputs are aggregated. Good for tasks involving independent sub-components or requiring diverse perspectives.
    - **Routing Workflow:** An LLM analyzes an input and intelligently routes it to the most appropriate specialized prompt, tool, or sub-workflow. Useful for handling diverse types of requests.
    - **Orchestrator-Workers:** A central LLM (the orchestrator) decomposes a complex task into smaller subtasks, which are then handled by specialized LLMs or tools (the workers). The orchestrator then synthesizes the results.
    - **Evaluator-Optimizer:** An iterative process where one LLM generates a solution, and another LLM (or the same LLM with a different prompt) evaluates it and provides feedback for refinement. Spring AI's Evaluator interface can be a key component here.
- Spring AI Implementation:
  These patterns can be constructed using Spring AI's foundational components: ChatClient, function calling/tool usage, Evaluators, and ChatMemory. For more advanced, self-directed agent capabilities, the incubating "Spring MCP Agent" project is exploring features like dynamic tool discovery via the Model Context Protocol (MCP) and maintaining an execution memory for tracking progress and decisions.4

While the allure of fully autonomous AI agents is strong, a pragmatic approach is often more effective, especially in enterprise contexts where predictability and reliability are paramount.[27] The evolution towards agentic systems is exciting, but Spring AI encourages developers to start with simpler, more controllable workflow patterns and incrementally add complexity as needed. This aligns with the enterprise preference for systems that are not only intelligent but also maintainable and understandable. Spring AI provides the building blocks for these structured agentic patterns (like Chain, Routing, and Orchestrator-Worker workflows) [4], allowing developers to construct sophisticated AI systems that remain manageable. The guiding principle, as suggested by best practices, is to "Start Simple" and employ the "simplest pattern that meets your requirements".[27] This philosophy ensures that developers can harness the power of more advanced AI behaviors without sacrificing control or introducing undue complexity, thereby creating solutions that are both innovative and enterprise-ready.

6.4 Best Practices for Developing and Deploying Spring AI Applications

Adhering to best practices is crucial for building high-quality, maintainable, and efficient Spring AI applications.

- **Start Simple, Iterate:** Begin with the simplest workflow or pattern that meets your initial requirements. Add complexity (e.g., more advanced prompting, more tools, agentic behaviors) incrementally as needed and justified.[27]
- **Design for Reliability:** Implement robust error handling for API calls, tool executions, and data processing. Use type-safe responses (POJO mapping) wherever possible. Validate inputs and outputs at each step of a workflow.[27]
- **Consider Trade-offs:** Consciously balance factors like latency vs. accuracy (e.g., more complex RAG might increase accuracy but also latency), and decide when parallel processing is beneficial versus sequential execution.[27]
- **Iterate and Refine Prompts:** Prompt engineering is an ongoing process. Continuously test, evaluate, and refine your prompts to improve the quality and relevance of LLM responses.[15]
- **Secure API Key Management:** Re-emphasize the importance of never hardcoding API keys. Use environment variables, Spring Cloud Config with Vault, or other secure secret management solutions.
- **Cost Management:** Be mindful of API costs. Monitor token usage closely (using observability features). Choose the smallest/cheapest model that can effectively perform the task. Utilize local models (Ollama) for development and suitable production scenarios to reduce costs.
- **Scalability:** Leverage Spring Boot's capabilities for building scalable applications.

For I/O-bound operations like calling LLM APIs, enable virtual threads in Java 21+ (spring.threads.virtual.enabled=true) for improved throughput.[2]

- **Performance with GraalVM Native Images:** For applications requiring fast startup times and reduced memory footprints, consider compiling your Spring AI application into a GraalVM native image. This requires careful configuration of runtime hints for reflection and resources.[2]
- **Modular Design:** Structure your AI logic into well-defined services and components, following Spring best practices.

6.5 Addressing Common Challenges in AI Application Development

Developing AI applications comes with a unique set of challenges that developers should be aware of:

- **Skill Gaps and Learning Curves:** AI, LLMs, and associated concepts are relatively new for many Java developers. A commitment to continuous learning and upskilling is essential.[29] This roadmap aims to bridge some of that gap for Spring AI.
- **Ethical Considerations and Bias:** LLMs are trained on vast datasets, which may contain societal biases. These biases can be reflected or even amplified in the model's outputs, leading to unfair or discriminatory outcomes.[29] Developers must be vigilant and consider fairness in their application design. Spring AI's support for Moderation Models can help filter out certain types of harmful content.[5]
- **Data Privacy and Security:** When using RAG with proprietary or sensitive enterprise data, or when user inputs contain personal information, data privacy and security are paramount.[29] Using local LLMs (via Ollama) can be an advantage here as data does not leave the local environment.[12] For cloud models, understand the provider's data handling policies.
- **Dependence on AI Tools / Over-Reliance:** While AI tools can greatly enhance productivity, it's important to maintain critical thinking and problem-solving skills. AI should augment human capabilities, not entirely replace them.[29]
- **Cost and Resource Constraints:** Accessing powerful LLM APIs can be expensive, based on token usage or per-call fees.[29] Efficient prompting, choosing appropriately sized models for the task, implementing caching strategies, and leveraging local models where feasible are important for cost control.
- **Hallucinations:** LLMs can generate responses that sound plausible but are factually incorrect or nonsensical.[2] RAG is a primary technique to mitigate this by grounding responses in factual data.[20] Model evaluation techniques, like fact-checking evaluators, also play a crucial role.[16]

6.6 Hands-on Project Proposals for Stage 5

These projects focus on production-readiness and exploring basic agentic patterns.

- **Project 1: RAG Output Evaluator**
  - **Description:** Take one of the RAG-based projects developed in Stage 3 (e.g., the "Chat with Your PDF" application). Implement a RelevancyEvaluator and/or a FactCheckingEvaluator to programmatically assess the quality of the answers generated by your RAG system. For a set of predefined questions and corresponding PDF contexts, run the RAG pipeline, capture the answers, and then use the evaluators to score these answers. Log the evaluation results (e.g., pass/fail, relevance score).
  - **Learning Objectives:** Gain practical experience with Spring AI's Evaluator interface, understand how to set up and use RelevancyEvaluator and FactCheckingEvaluator, and learn how to integrate automated evaluation into an AI workflow.
  - **Key Steps:**
    1. Identify a set of test questions for your RAG application.
    2. If using FactCheckingEvaluator with a model like Minicheck, set up Ollama to serve it.
    3. Instantiate and configure the chosen evaluators in your Spring application.
    4. For each test question, run your RAG pipeline to get an answer and the retrieved context.
    5. Create an EvaluationRequest with the query, context, and answer.
    6. Call the evaluate() method and log or store the EvaluationResponse.
    7. Analyze the evaluation results to identify areas for improvement in your RAG system.
- **Project 2: Simple Research Agent (Chain Workflow + Function Calling)**
  - **Description:** Design a simple "research agent" that takes a broad research topic as input from the user. The agent should perform the following steps in a chain:
    1. **Step 1 (LLM - Task Decomposition):** Use an LLM to break down the given research topic into 2-3 specific, answerable key questions.
    2. **Step 2 (Function Calling - Information Retrieval):** For each key question generated in Step 1, use a function calling mechanism to query an external search engine API (you can mock this tool if a live API is too complex, or use a simple web search library). The tool should return a few relevant text snippets for each question.
    3. **Step 3 (LLM - Synthesis):** Use an LLM to synthesize the information gathered from the search snippets (from Step 2) in the context of the key

    questions (from Step 1) into a concise summary report on the original research topic.
- **Learning Objectives:** Implement a multi-step AI workflow (a basic chain), combine LLM-driven reasoning (task decomposition, synthesis) with tool usage (information retrieval), and manage state/data flow between steps.
- **Key Steps:**
    1. Design prompts for the LLM for task decomposition (Step 1) and synthesis (Step 3).
    2. Implement the search "tool" as a Java method that can be called via function calling. This method will take a question string and return a list of text snippets.
    3. Orchestrate the flow:
        - Get topic from user.
        - Call LLM for Step 1 to get key questions.
        - Loop through key questions, call the search tool for each via LLM function calling (or directly if the LLM just generates the questions as text).
        - Collect all search snippets.
        - Call LLM for Step 3, providing the original topic, key questions, and all collected snippets as context, to generate the final report.

Table 5: Production Readiness Checklist for Spring AI Applications

Aspect	**Key Spring AI Features/Strategies**	**Considerations/Best Practices**
Evaluation	Evaluator interface, RelevancyEvaluator, FactCheckingEvaluator.[4]	Define clear evaluation criteria. Use in tests and CI/CD. Consider human evaluation for nuanced tasks. Choose appropriate evaluation models (e.g., smaller ones for cost-effectiveness).[16]
Observability	Micrometer integration for metrics (latency, token usage, tool calls) and tracing. Configurable logging of	Monitor key metrics in production. Use tracing to debug performance issues. Be cautious with logging sensitive data. Integrate with

	prompts/responses.[4]	existing monitoring infrastructure (e.g., Prometheus, Grafana).[26]
Error Handling	Standard Spring error handling. Retry mechanisms (e.g., Spring Retry) can be applied to ChatClient calls.	Implement comprehensive error handling for API calls, tool failures, and data processing issues. Provide graceful fallbacks or user feedback.
Security	Secure API key management (env vars, Vault). For RAG, consider data sanitization and access controls for the knowledge base. Use Moderation Models if needed.[5]	Never hardcode secrets. Follow least privilege principles for API keys. Be aware of prompt injection vulnerabilities. Ensure data privacy for user inputs and RAG sources.
Scalability	Spring Boot's inherent scalability. Support for virtual threads (spring.threads.virtual.enabled=true) for I/O-bound LLM calls.[2]	Design for statelessness where possible. Use asynchronous processing for long-running AI tasks. Load test your application.
Cost Management	Token usage metrics via Micrometer.[4] Choice of models (smaller/cheaper vs. larger/expensive). Caching strategies. Use of local LLMs (Ollama).[12]	Monitor token consumption closely. Optimize prompts to be concise yet effective. Select the most cost-efficient model for each task. Implement caching for frequently requested, non-dynamic AI responses.
Maintainability	Modular design using Spring components. Portable abstractions for models and vector stores.[5] Clear prompt management.	Write clean, well-documented code. Keep prompts version-controlled and separate from business logic if complex. Leverage Spring AI's abstractions to avoid vendor lock-in.

Configuration	Externalized configuration via application.properties/yml. Spring Profiles for environment-specific settings.	Manage AI model versions, endpoint URLs, and options through configuration. Use profiles for dev, test, prod environments.

Section 7: Conclusion and Future Learning Paths

This roadmap has charted a course for Java Spring Framework developers to navigate the exciting and rapidly evolving world of AI application development using Spring AI. The journey progresses from understanding fundamental AI concepts and setting up the Spring AI environment to mastering its core APIs, building data-aware applications with Retrieval Augmented Generation (RAG), enhancing interactions with function calling and chat memory, and finally, considering production-readiness and exploring agentic AI patterns.

7.1 Recap of Key Learnings from the Roadmap

Throughout this journey, several key themes and capabilities of Spring AI have been highlighted:

- **Simplified AI Integration:** Spring AI successfully lowers the barrier to entry for Java developers by providing familiar Spring-like abstractions, auto-configuration, and fluent APIs, making the integration of complex AI functionalities feel like a natural extension of the Spring ecosystem.
- **Portability and Flexibility:** The framework's design emphasizes portability across different AI model providers (OpenAI, Ollama, Hugging Face, etc.) and vector stores (ChromaDB, PGVector, Redis, etc.), allowing developers to choose the best tools for their needs and adapt to future changes without major code overhauls.
- **Powerful Core Features:** Developers gain proficiency with essential tools like ChatClient for diverse conversational interactions, EmbeddingClient for semantic understanding, robust support for RAG to connect LLMs with external data, function calling to enable LLMs to interact with external systems, and ChatMemory for building coherent conversational experiences.
- **Focus on Practical Application:** The hands-on project proposals at each stage are designed to ensure that theoretical knowledge is translated into practical skills, enabling developers to build tangible AI-powered applications.
- **Enterprise-Ready Capabilities:** Spring AI addresses critical enterprise concerns through features like structured POJO outputs, comprehensive observability via Micrometer, model evaluation tools, and alignment with Spring Boot's

production-grade features like GraalVM support and virtual threads.

7.2 The Evolving Landscape of Spring AI and Generative AI

The field of Generative AI is characterized by exceptionally rapid innovation. New models, techniques, and tools are emerging at an unprecedented pace.[1] Spring AI is a dynamic project committed to keeping pace with this evolution, regularly incorporating support for new AI models, enhancing its feature set, and exploring advanced concepts like the Model Context Protocol (MCP) for more sophisticated agentic interactions.[4]

This rapid evolution means that continuous learning is not just beneficial but essential for developers working in this space. The principles and patterns learned through this roadmap provide a strong foundation, but staying updated with the latest Spring AI releases, new features in supported AI models, and emerging best practices in the broader AI community will be crucial for long-term success and innovation.

The journey into AI development is ongoing, as the field itself is in a constant state of advancement. New large language models with improved capabilities, novel prompt engineering techniques, more efficient vector search algorithms, and increasingly sophisticated agentic frameworks are released frequently. Spring AI, as a project within the vibrant Spring ecosystem, is also evolving, as evidenced by its progression to a 1.0 General Availability release and the incubation of advanced projects like the MCP Agent.[4] Therefore, while this roadmap provides a comprehensive pathway to achieving proficiency, it should be viewed as a foundational step in a continuous learning process. Developers are encouraged to actively engage with the Spring AI community, monitor official announcements, and explore new research and tools to keep their skills sharp and leverage the full potential of AI in their Java applications.

7.3 Resources for Continued Learning

To support this continuous learning journey, several resources are invaluable:

- **Official Spring AI Documentation:** The primary source for detailed information, API references, and configuration options (https://docs.spring.io/spring-ai/reference/index.html).[5]
- **Spring Blog:** Features articles on new Spring AI releases, tutorials on specific features, and discussions on best practices and patterns (https://spring.io/blog/).[2]
- **Community Forums:**
  - Stack Overflow: Use tags like spring-ai, spring-boot, and specific AI model tags for questions and answers.
  - Reddit: Subreddits like r/SpringBoot and r/java may have relevant

discussions.[30]

- **Tutorials and Blogs by Experts:**
    - Websites like Baeldung often feature Spring and Spring AI tutorials.
    - Blogs from contributors to the Spring AI ecosystem, such as Piotr Minkowski, provide practical examples and insights into advanced usage.
- **"Awesome Spring AI" List:** A curated list of resources, projects, and articles related to Spring AI, often mentioned in official documentation or community channels.[26]
- **Online Courses:** Platforms like Coursera may offer specializations or courses relevant to Generative AI for Java developers, which can complement practical Spring AI learning.[1]

By leveraging these resources and maintaining a proactive approach to learning, Java Spring developers can continue to expand their expertise in AI, build increasingly sophisticated applications with Spring AI, and stay at the forefront of this transformative technology.

Works cited

1. Generative AI for Java and Spring Developers Specialization - Coursera, accessed May 25, 2025, https://www.coursera.org/specializations/generative-ai-for-java-and-spring-developers
2. Your First Spring AI 1.0 Application, accessed May 25, 2025, https://spring.io/blog/2025/05/20/your-first-spring-ai-1/
3. Spring AI 1.0 Released, Streamlines AI Application Development with Broad Model Support, accessed May 25, 2025, https://www.infoq.com/news/2025/05/spring-ai-1-0-streamlines-apps/
4. Spring AI 1.0 GA Released, accessed May 25, 2025, https://spring.io/blog/2025/05/20/spring-ai-1-0-GA-released/
5. Introduction :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/index.html
6. Spring AI, accessed May 25, 2025, https://spring.io/projects/spring-ai/
7. Build fast, production-worthy AI apps with Spring AI and Redis - Redis, accessed May 25, 2025, https://redis.io/blog/build-fast-production-worthy-ai-apps-with-spring-ai-and-redis/
8. OpenAI Chat :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/api/chat/openai-chat.html
9. Getting Started With Spring AI and PostgreSQL PGVector - DZone, accessed May 25, 2025, https://dzone.com/articles/spring-ai-with-postgresql-pgvector
10. Using Ollama with Spring AI - Piotr's TechBlog, accessed May 25, 2025, https://piotrminkowski.com/2025/03/10/using-ollama-with-spring-ai/

11. Hugging Face | Spring AI1.0.0-M6中文文档|Spring官方文档 ..., accessed May 25, 2025, https://www.spring-doc.cn/spring-ai/1.0.0-M6/api_chat_huggingface.en.html
12. Local AI with Spring: Building Privacy-First Agents Using Ollama - Foojay.io, accessed May 25, 2025, https://foojay.io/today/local-ai-with-spring-building-privacy-first-agents-using-ollama/
13. Spring AI With ChromaDB Vector Store | Baeldung, accessed May 25, 2025, https://www.baeldung.com/spring-ai-chromadb-vector-store
14. Prompt Engineering Techniques with Spring AI, accessed May 25, 2025, https://spring.io/blog/2025/04/14/spring-ai-prompt-engineering-patterns/
15. Mastering Prompt Engineering with Spring AI: Techniques and Best Practices | daily.dev, accessed May 25, 2025, https://app.daily.dev/posts/mastering-prompt-engineering-with-spring-ai-techniques-and-best-practices-tmq0txyfz
16. Evaluation Testing :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/api/testing.html
17. Getting Started with Spring AI: Add LLM Power to Your Spring Boot Apps - DEV Community, accessed May 25, 2025, https://dev.to/haraf/getting-started-with-spring-ai-add-llm-power-to-your-spring-boot-apps-cm3
18. Build and Evaluate AI Models (Spring Boot and Hugging Face) - DEV Community, accessed May 25, 2025, https://dev.to/dilipkumar_0418/effortless-ai-model-integration-build-and-evaluate-ai-models-spring-boot-and-hugging-face-oal
19. Hugging Face Chat :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/api/chat/huggingface.html
20. Using RAG and Vector Store with Spring AI - Piotr's TechBlog, accessed May 25, 2025, https://piotrminkowski.com/2025/02/24/using-rag-and-vector-store-with-spring-ai/
21. Vector Databases :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/api/vectordbs.html
22. Implementing Semantic Search Using Spring AI and PGVector ..., accessed May 25, 2025, https://www.baeldung.com/spring-ai-pgvector-semantic-search
23. Google Cloud and Spring AI 1.0, accessed May 25, 2025, https://cloud.google.com/blog/topics/developers-practitioners/google-cloud-and-spring-ai-10
24. Spring AI with Azure OpenAI - Piotr's TechBlog, accessed May 25, 2025, https://piotrminkowski.com/2025/03/25/spring-ai-with-azure-openai/
25. Chat Memory :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/api/chat-memory.html
26. Observability :: Spring AI Reference, accessed May 25, 2025, https://docs.spring.io/spring-ai/reference/observability/index.html
27. Building Effective Agents with Spring AI (Part 1), accessed May 25, 2025,

https://spring.io/blog/2025/01/21/spring-ai-agentic-patterns/
28. Ai Agent Patterns with Spring AI - DEV Community, accessed May 25, 2025, https://dev.to/lucasnscr/ai-agent-patterns-with-spring-ai-43gl
29. AI in Software Development: Key Challenges You Can't Ignore - Litslink, accessed May 25, 2025, https://litslink.com/blog/the-impact-of-ai-on-software-development-with-key-opportunities-and-challenges
30. Spring AI tutorial for beginners : r/SpringBoot - Reddit, accessed May 25, 2025, https://www.reddit.com/r/SpringBoot/comments/1jluavv/spring_ai_tutorial_for_beginners/