eazy bytes

SPRING AI

Why Spring AI ?

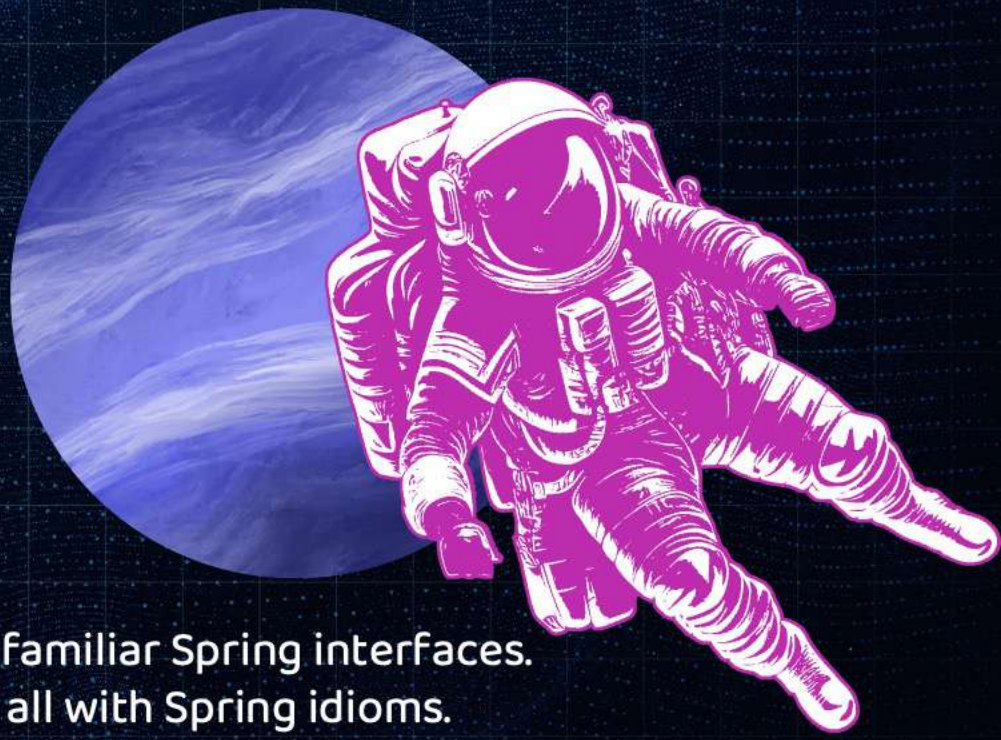
eazy bytes

Because Spring Developers Deserve Superpowers Too

Spring Boot gave us the power to build APIs, microservices, and full-stack backends effortlessly.
But in the world of AI and LLMs, Spring devs were stuck asking:

- "How do I call OpenAI safely?"
- "Where do I plug in a prompt?"

Spring AI is the answer.



Spring AI Makes Generative AI Spring-Native

- Spring AI wraps complex AI model APIs (OpenAI, Ollama, Bedrock) in familiar Spring interfaces.
- Gives you ChatClient, ImageModel, SpeechModel, and even Memory, all with Spring idioms.
- Makes AI integration feel like configuring a @Bean, not battling a black-box.
- So if you're already a Spring developer — you're 80% there.

What's on the Menu? (Course Agenda)

eazy bytes

Say Hello to Spring AI

01

We start with the basics. We will build our first Spring AI app with:

- OpenAI
- Ollama
- Docker Model Runner
- AWS Bedrock

Spring AI Essentials

02

We go deep into the brains of LLMs:

- Prompt templates, streaming, ChatOptions
- Built-in advisors, custom advisors, prompt stuffing
- Converting AI output into a POJO etc

What's on the Menu? (Course Agenda)

eazy bytes

Foundations of Generative AI & LLMs

03

Ever wondered how AI thinks?

We'll trace the roots of generative AI, unpack large language models (LLMs), and decode buzzwords like

- Tokens
- Embeddings, Vectors
- Attention mechanisms

Teaching LLMs to Remember

04

Stateless by default? Not anymore.

- We explore ChatMemory with persistence, JDBC, per-user memory, and token management.
- Your AI will finally remember you (unlike your old college friends 😅)

What's on the Menu? (Course Agenda)

eazy bytes

Talking to Documents (RAG)

05

Forget prompt stuffing. It's time for RAG (Retrieval-Augmented Generation):

- We chunk, embed, vectorize, and query documents
- Using tools like Qdrant and Spring AI's VectorStore
- Now your app can search PDFs like a pro.

Tool Calling in Action

06

LLMs can't access the current data. So, we'll teach them with tool calling!

- We'll teach your AI to check the time, query databases, and basically do anything your Java code can do.
- This is where your app goes from chatty to actually useful

What's on the Menu? (Course Agenda)

eazy bytes

Model Context Protocol (MCP)

07

Think of MCP as the USB-C for AI tools:

- Learn about clients, servers, transports
- Build our own MCP server
- And even connect to GitHub's MCP Server

Evaluators: Testing AI Like a Boss

08

- AI can sometimes hallucinate (yep, it makes stuff up). We'll use Evaluator to keep it honest.
- Learn how to catch bad answers at runtime, validate responses and retry

What's on the Menu? (Course Agenda)

eazy bytes

Observability with Spring AI

09

- Keep an eye on our AI app with Spring Boot Actuator, Prometheus, and Grafana dashboards
- We'll trace AI operations with OTLP and Jaeger

Working with Voice & Image

10

Our apps will not only talk the talk but also create the visuals to match! We'll explore,

- Transcription
- Text-to-speech
- Image generation.

Pre-requisites

- ✓ Good understanding on Java
- ✓ Good understanding on Spring Boot
- ✓ Familiarity with Docker & Postman
- ✓ Curiosity for AI
- ✓ A Laptop
- ✓ Willing to spend 5 dollars for OpenAI bills*

Spring AI Framework

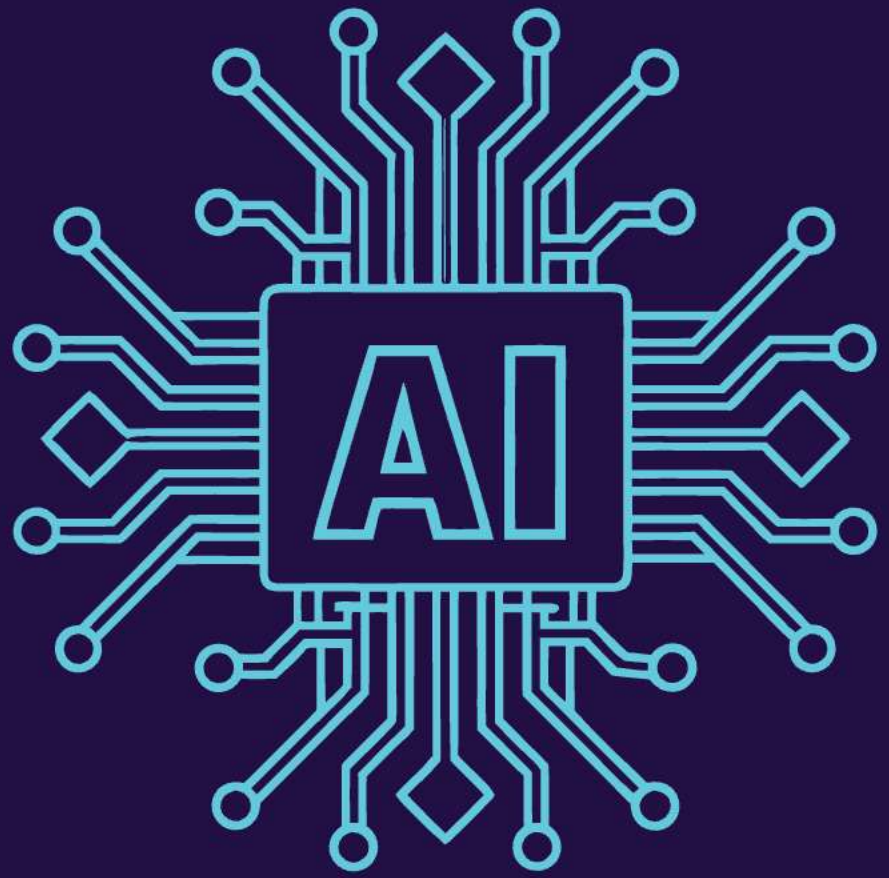
eazy bytes

What is Spring AI?

- Framework for integrating AI into Spring applications.
- Simplifies adding generative AI using Spring's modular, POJO-based design.
- Connects enterprise data and APIs with AI models seamlessly.

Core Features

- Multi-Provider Support: Integrates with OpenAI, Anthropic, AWS, Google, Hugging Face for chat, embeddings, text-to-image, audio, and moderation.
- MCP support: Allows building MCP clients and servers easily
- Vector Store Integration: Supports RAG with Pinecone, Redis, PostgreSQL/pgVector, MongoDB, etc.
- Advanced Patterns: Chat memory, tool/function calling, and Advisors API for reusable AI logic.
- Observability & Guardrails: Monitors AI operations and evaluates outputs to ensure reliability.



Spring AI Framework

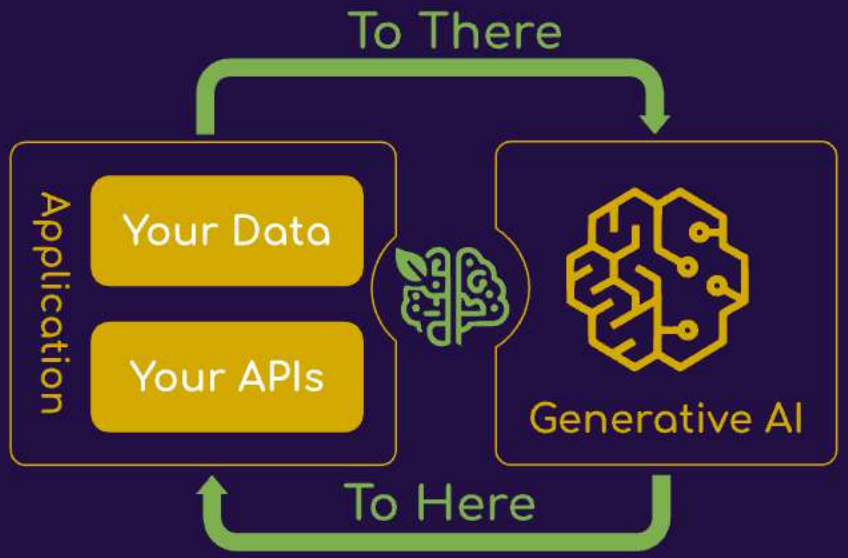
eazy bytes

Spring Ecosystem Integration

- Spring Boot: Auto-configuration and starters for quick setup with AI models and vector DBs.
- Familiar Model: Uses Spring's dependency injection and abstractions, behaves like any Spring Bean.
- Modular: Easily swap AI providers or vector stores without changing core logic.

Typical Use Cases

- AI-powered customer support: Build intelligent assistants with conversation memory and API integration.
- Natural language search over documents: Summarize or query proprietary data using RAG and vector stores.
- Personalized recommendations: Power intent-based search and recommendations with embeddings.
- Data Processing: Add AI to pipelines for content moderation, transcription, or text generation.



Building a "Hello World" App with Spring AI & Open AI

eazy bytes

1. Add Required Dependencies - Enables Spring Boot web and OpenAI integration

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-openai</artifactId>
</dependency>
```


2. Generate a API key and configure in app –

- Goto the website https://platform.openai.com/ and generate a API key
- Configure your API key in application.properties or application.yml:

```
spring.ai.openai.api-key=${OPENAI_API_KEY}
```

Building a "Hello World" App with Spring AI & Open AI

eazy bytes

3. Create the Chat Controller

```
@RestController
@RequestMapping("/api")
public class ChatController {

    private final ChatClient chatClient;

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

    @GetMapping("/chat")
    String chat(@RequestParam("message") String message) {
        return this.chatClient.prompt(message)
                              .call()
                              .content();
    }
}
```


You now have a minimal working Spring AI app that connects to OpenAI and responds to your prompts!

ChatModel

eazy bytes

What is ChatModel?

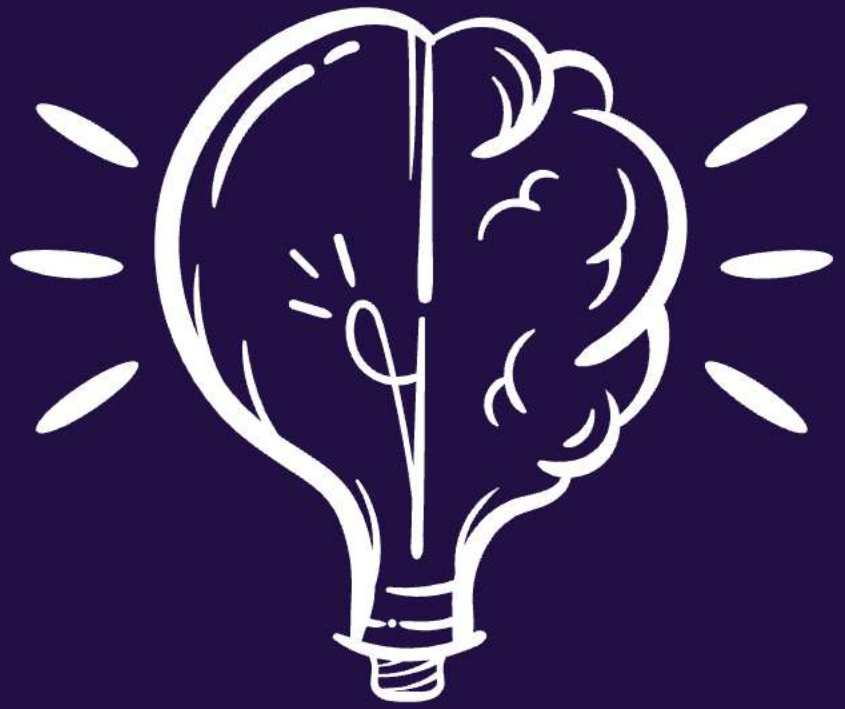
ChatModel is the lower-level abstraction that represents the actual AI model interface. It's the core component that:

- Defines the contract for communicating with different AI providers (OpenAI, Azure OpenAI, Anthropic, etc.)
- Handles the actual API calls to the AI services
- Manages model-specific configurations and parameters
- Provides the foundational layer for AI interactions

Examples of implementations:

- `OpenAiChatModel (for OpenAI)`
- `GeminiChatModel (for Google)`
- `MistralChatModel, etc.`

Purpose: Encapsulates how to interact with a specific underlying AI provider.



ChatClient

eazy bytes

What is ChatClient?

ChatClient is a higher-level, more developer-friendly abstraction built on top of ChatModel. It provides:

- A fluent API for easier interaction with AI models
- Better developer experience with method chaining
- Simplified prompt construction and message handling
- Supports both synchronous and streaming programming models

- ChatClient is the friendly wrapper (or service layer) around a ChatModel.
  It takes care of:
  - `Building a Prompt`
  - `Managing chat history`
  - `Invoking the model`
  - `Extracting the content`
- Purpose: Simplifies working with an AI model and abstracts prompt/message handling.

ChatModel & ChatClient

eazy bytes

The Relationship between ChatModel & ChatClient

Think of it this way:

ChatModel = The engine (does the heavy lifting of actual AI communication)
ChatClient = The steering wheel and dashboard (provides an intuitive interface to control the engine)

ChatClient uses ChatModel internally but wraps it with a more convenient API. When you use ChatClient, it eventually delegates to the underlying ChatModel to make the actual AI service calls.

ChatModel & ChatClient

eazy bytes

How They Work Together

Spring Boot Autoconfiguration: When you add Spring AI dependencies, Spring Boot automatically creates ChatModel beans for configured AI providers

ChatClient.Builder Creation: The framework provides an autoconfigured ChatClient.Builder that's already wired with the appropriate ChatModel

Fluent API Usage: You use ChatClient's fluent methods to build prompts, which internally get converted to the format expected by the ChatModel

Execution: ChatClient delegates to ChatModel to send requests to the AI service and handle responses

Example Flow

```
Your Code → ChatClient (fluent API) → ChatModel (AI service integration) → AI Provider API
```

This design follows the common pattern of having a low-level technical interface (ChatModel) and a high-level user-friendly interface (ChatClient) that makes the framework more accessible while maintaining flexibility for advanced use cases.

Building a “Hello World” App with Spring AI & Ollama

eazy bytes

What is Ollama?

Ollama is a developer-friendly tool that allows you to run large language models (LLMs) locally on your machine — easily and efficiently. Built for privacy, speed, and offline use.

Key features of Ollama:

Local execution: Run AI models entirely on your machine without internet dependency

Simple interface: Easy command-line tool for model management

Model library: Access to popular open-source models through a simple download system

API access: Provides REST API endpoints so you can integrate models into your applications

Cross-platform: Works on macOS, Linux, and Windows

To get started with Ollama, visit https://ollama.com/

Building a "Hello World" App with Spring AI & Ollama

eazy bytes

1. Install model in local using ollama - We can install a model with a simple command like mentioned below,

```
ollama run llama3.2:1b
```

2. Add Required Dependencies - Enables Spring Boot web and ollama integration

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-model-ollama</artifactId>
</dependency>
```


3. configure properties in Spring AI app

Configure below properties in application.properties or application.yml.
API key is not required as the model is available in local.

```
spring.ai.model.chat=ollama
spring.ai.ollama.chat.options.model=llama3.2:1b
```

Building a "Hello World" App with Spring AI & Ollama

eazy bytes

4. Create the Chat Controller

```
@RestController
@RequestMapping("/api")
public class ChatController {

    private final ChatClient chatClient;

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

    @GetMapping("/chat")
    String chat(@RequestParam("message") String message) {
        return this.chatClient.prompt(message)
                              .call()
                              .content();
    }
}
```


You now have a minimal working Spring AI app that connects to ollama and responds to your prompts!

Building a "Hello World" App with Spring AI & Docker

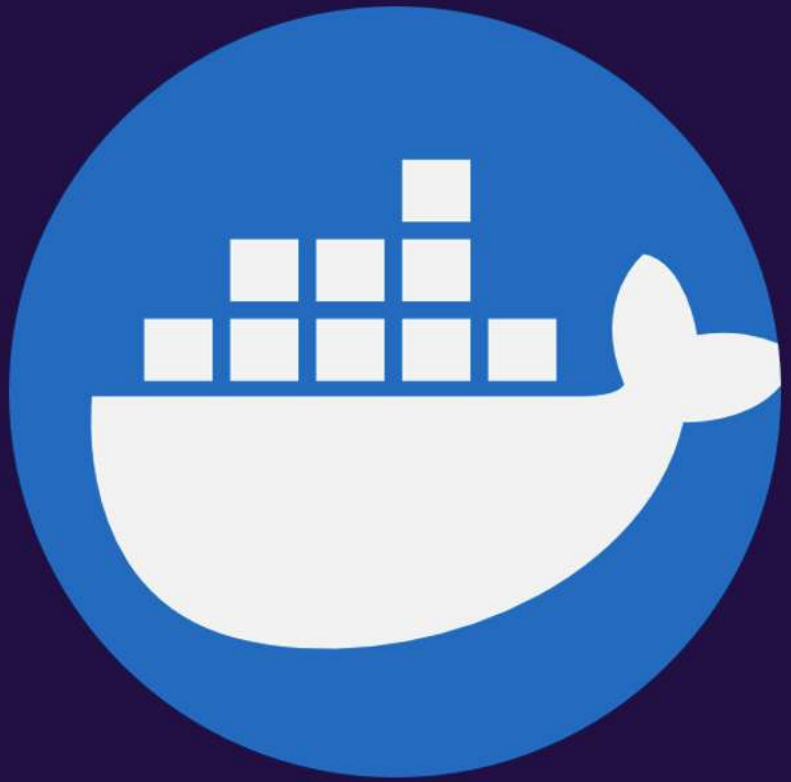
eazy bytes

🐋 What is Docker Model Runner?

Docker Model Runner is a general term referring to the practice of running AI models inside Docker containers, often to provide a local, isolated, and portable environment for serving LLMs or other machine learning models.

It's commonly used to:

- Run open-source LLMs (e.g., LLaMA2, Mistral, Falcon)
- Deploy tools like Ollama, LM Studio, LocalAI, Text Generation Web UI
- Host custom or fine-tuned models in a containerized setup



For more details, refer below links,

- https://www.docker.com/blog/run-llms-locally/
- https://www.docker.com/blog/introducing-docker-model-runner/

Building a "Hello World" App with Spring AI & Docker

eazy bytes

1. Install model in local using Docker - We need to run a model locally using the below docker command,

```
docker model run ai/gemma3
```

This will Loads the model and exposes a REST API which accepts prompt requests and returns completions

2. Add Required Dependencies - Enables Spring Boot web and Docker integration

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-openai</artifactId>
</dependency>
```


3. configure properties in Spring AI app

Configure below properties in application.properties or application.yml. You need to mention a dummy API key

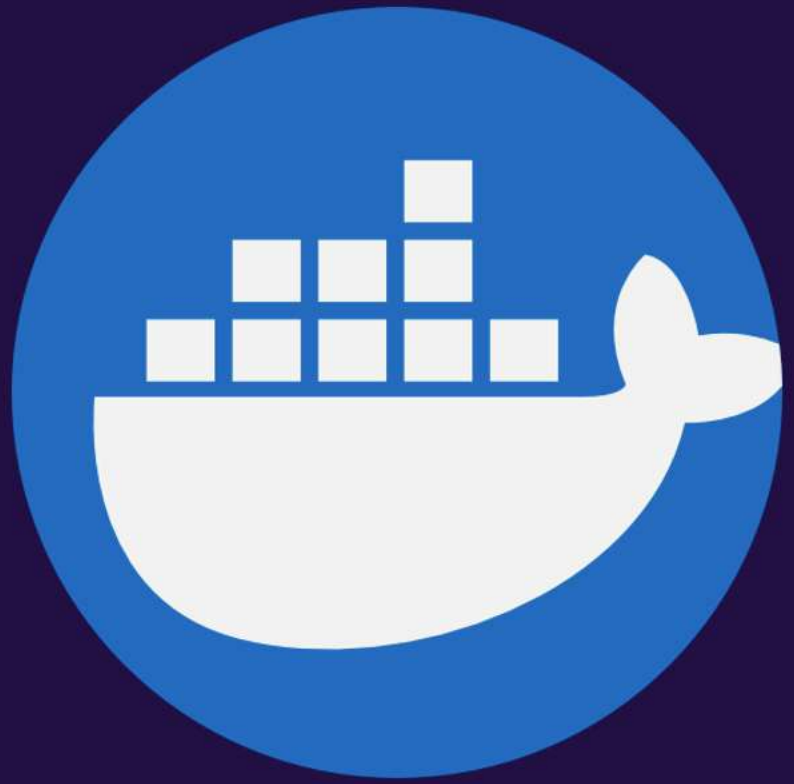
```
spring.ai.openai.api-key=test
spring.ai.openai.base-url=http://localhost:12434/engines
spring.ai.openai.chat.options.model=ai/gemma3
```

Building a "Hello World" App with Spring AI & Docker

eazy bytes

4. Create the Chat Controller

```
@RestController
@RequestMapping("/api")
public class ChatController {

    private final ChatClient chatClient;

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

    @GetMapping("/chat")
    String chat(@RequestParam("message") String message) {
        return this.chatClient.prompt(message)
                              .call()
                              .content();
    }
}
```

You now have a minimal working Spring AI app that connects to Docker model and responds to your prompts!

Building a “Hello World” App with Spring AI & AWS Bedrock

eazy bytes

☁️ What is AWS Bedrock?

Amazon Bedrock is a fully managed service offered by Amazon Web Services (AWS) that simplifies building and deploying generative AI applications.

🚀 Key Highlights

- No infrastructure to manage
- Access to multiple foundation models (FMs) from multiple vendors
- Use fine-tuning and RAG (Retrieval Augmented Generation) with your own data.
- Fully managed and scalable. Runs on AWS — serverless, secure, and enterprise-ready.

Similar offering is available in Azure(Azure OpenAI Service) and GCP(Google Vertex AI)

Building a "Hello World" App with Spring AI & AWS Bedrock

eazy bytes

1. Enable and Setup model, API keys in AWS

You need to select a model and request access to it. Once the access is granted, configure access keys for an IAM user

2. Add Required Dependencies - Enables Spring Boot web and Bedrock integration

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-bedrock-converse</artifactId>
</dependency>
```


Building a "Hello World" App with Spring AI & AWS Bedrock

eazy bytes

3. configure properties in Spring AI app

Configure below properties in application.properties or application.yml. You need to mention a dummy API key

```
spring.ai.bedrock.aws.region=ap-south-1
spring.ai.bedrock.aws.access-key=AKIASNL2HASEBZSZDYFB
spring.ai.bedrock.aws.secret-key=XAgiI1dObcn6/egFe7Meu9/SQOcuyYFeg46LKK74
spring.ai.bedrock.aws.timeout=10m
spring.ai.model.chat=bedrock-converse
spring.ai.bedrock.converse.chat.options.model=apac.anthropic.claude-sonnet-4-20250514-v1:0
```


Building a "Hello World" App with Spring AI & AWS Bedrock

eazy bytes

4. Create the Chat Controller

```
@RestController
@RequestMapping("/api")
public class ChatController {

    private final ChatClient chatClient;

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

    @GetMapping("/chat")
    String chat(@RequestParam("message") String message) {
        return this.chatClient.prompt(message)
                              .call()
                              .content();
    }
}
```


You now have a minimal working Spring AI app that connects to Bedrock and responds to your prompts!

Working with Multiple Chat Models in Spring AI

eazy bytes

In real-world applications, it's common to integrate multiple chat models for better flexibility, performance, and user experience. Here are some practical scenarios:

Why Use Multiple Chat Models?

✅ Task-Based Model Selection
Use a powerful model for complex reasoning and a lightweight model for basic queries.

🔁 Fallback Strategy
Automatically switch to another model if the primary one is unavailable.

🧪 A/B Testing
Test different models or configurations to compare accuracy, latency, and cost.

💁 User Preference
Allow users to choose their preferred model for interaction.

🔧 Specialized Models
Combine models with different strengths (e.g., one for code, another for creative writing).

Working with Multiple Chat Models in Spring AI

eazy bytes

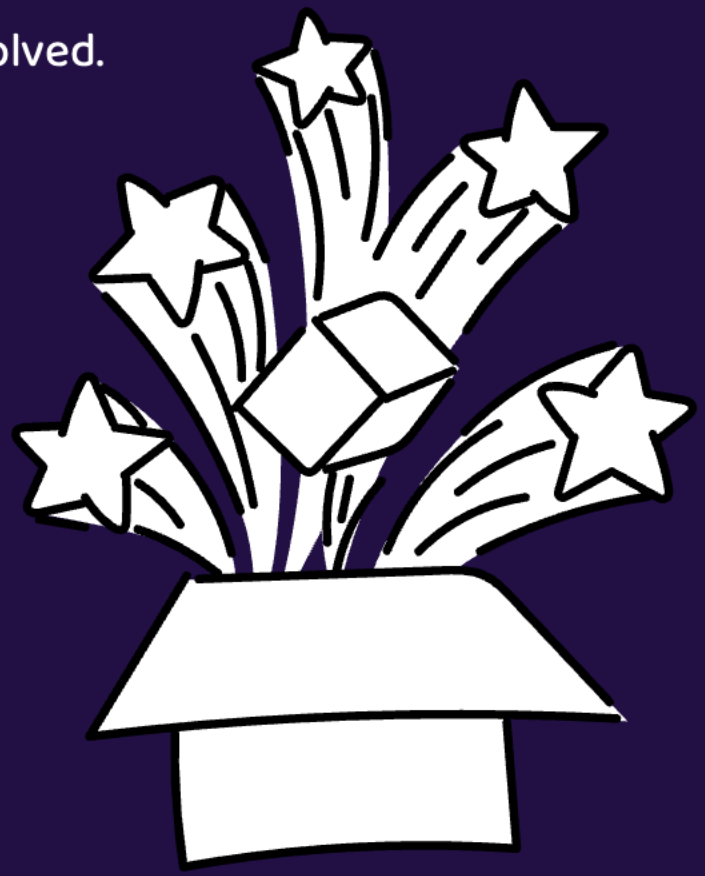
Spring AI Default Behavior

Spring AI auto-configures a single ChatClient.Builder bean.
This is suitable for simple use cases but not sufficient when multiple models are involved.

What You'll Need to Do

Manually configure multiple ChatClient.Builder beans for each model.
Inject the desired builder where needed based on your business requirements.

You need to disable the ChatClient.Builder autoconfiguration by setting the property `spring.ai.chat.client.enabled=false`

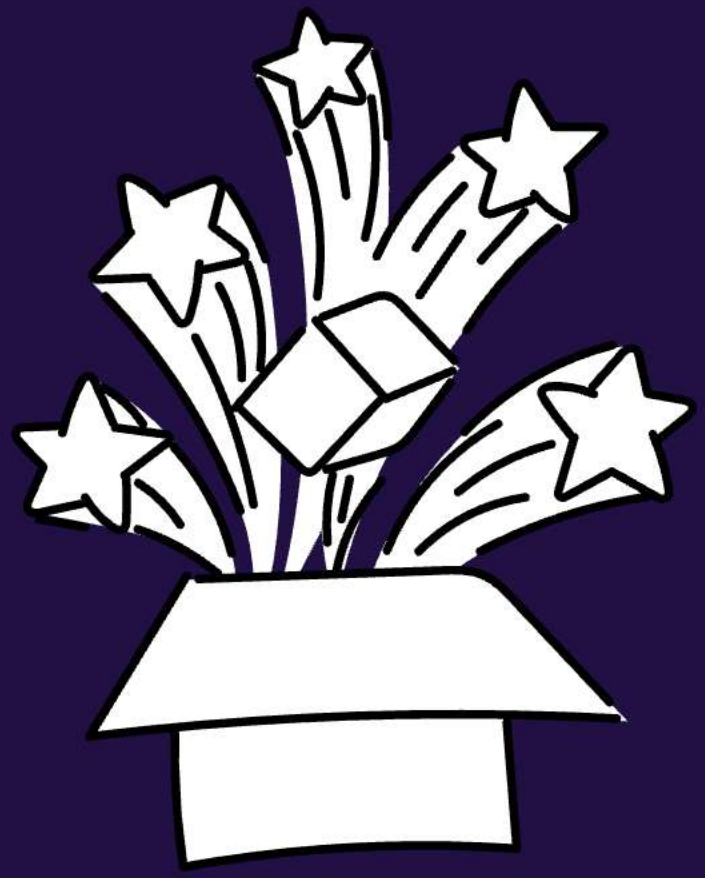


Working with Multiple Chat Models in Spring AI

eazy bytes

Create multiple ChatClient instances manually for each model. The same client instances can be autowired in the required controller classes.

```
@Configuration
public class ChatClientConfig {

    @Bean
    public ChatClient openAiChatClient(OpenAiChatModel chatModel) {
        return ChatClient.create(chatModel);
    }

    @Bean
    public ChatClient ollamaChatClient(OllamaChatModel chatModel) {
        ChatClient.Builder chatClientBulder =
                ChatClient.builder(ollamaChatModel);
        return chatClientBulder.build();
    }



}
```

Unlocking the Power of Spring AI

eazy bytes

What You've Built So Far

- A simple Spring AI app that sends prompts and receives responses
- Great start—but it only scratches the surface!

🚀 What's Coming Up

Spring AI can do much more:

- Message roles
- Advisors
- Prompt Templates
- Chat History Management
- RAG (Retrieval-Augmented Generation)
- Function Calling (Tools Integration)
- Building MCP clients & Server

📚 Get Ready!

You're about to unlock the full potential of Generative AI with Spring AI!

Understanding Message Roles in LLMs

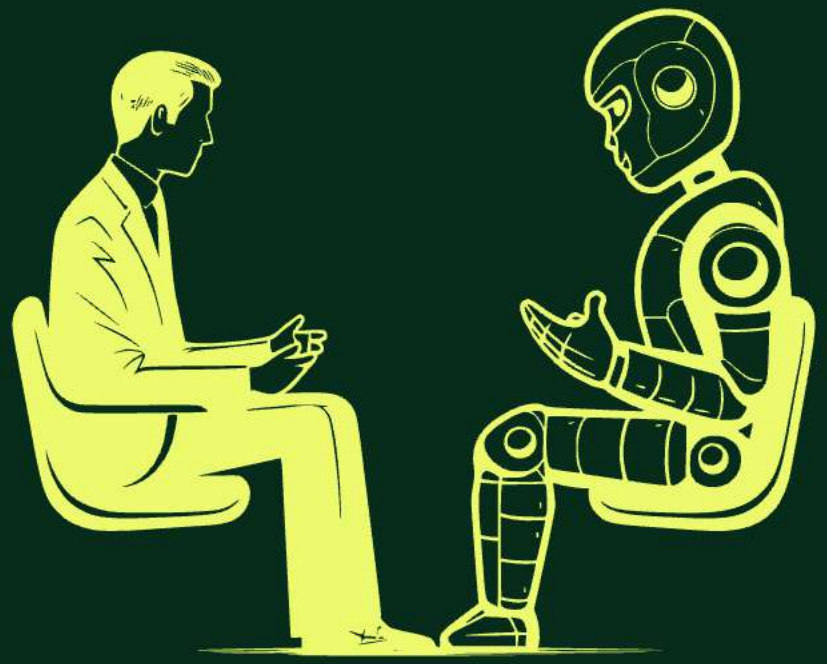
eazy bytes

What Are Message Roles in LLMs?

When interacting with Large Language Models (LLMs), we send prompts made up of messages. Each message has a role that helps the LLM understand the context and how to respond.

Common Roles (Used by Most APIs)

Role	Description
👱 User	What the user says or asks. Example: "Tell me a joke."
🛠️ System	Instructions for how the LLM should behave. Example: "Be formal."
🤖 Assistant	The LLM's response. Example: "Sure! Why did the chicken..."
⚙️ Function/Tool	Special instructions to run a function or fetch data



Example Prompt Breakdown:

```
System: "You are a friendly tour guide."
User: "What are the top 3 places to visit in Rome?"
Assistant: "Sure! The top 3 are..."
```

Understanding Message Roles in LLMs

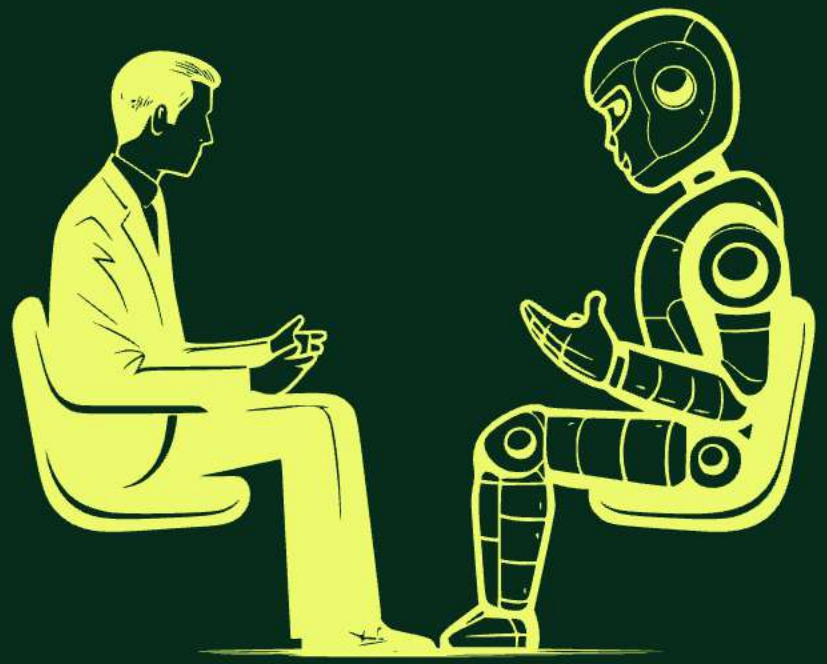
eazy bytes

💡 Analogy: It's Like a Stage Play 🎭

System: The director giving stage instructions (e.g., "Act like a professional chef")
User: The audience asking a question (e.g., "How do I cook pasta?")
Assistant: The actor replying (e.g., "Boil water and add pasta for 8 minutes.")
Function: A backstage helper fetching ingredients or recipes

Not All LLM model providers Support All Roles

Provider	Supported Roles
OpenAI	User, System, Assistant, Function
Anthropic	User, System, Assistant
Mistral AI	User, System, Assistant
Google Gemini	No System/Function roles. Only: User, Model (like Assistant)



Spring AI Handling Tip

If the System role is not supported (like in Gemini), Spring AI combines the System message into the User message behind the scenes.

Understanding Message Roles in LLMs

eazy bytes

Creating the ChatClient with a Default System Message

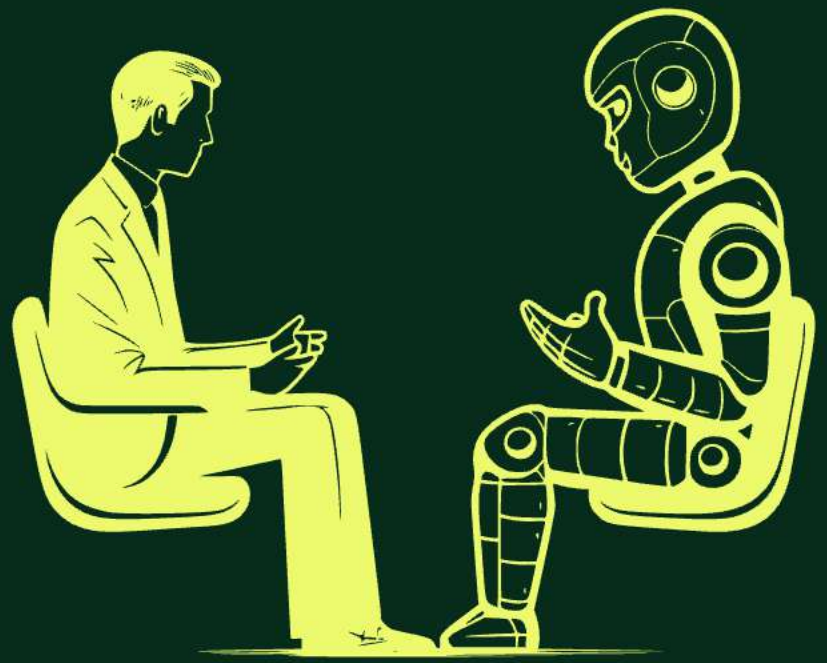
```
this.chatClient = chatClientBuilder
    .defaultSystem("""
        You are an internal HR assistant...
      """).build();
```

- Sets the default behavior/personality of the AI assistant.
- Used for general fallback behavior across requests.

Overriding System Message (Optional) in Runtime

```
chatClient.prompt()
    .system("""
        You are an internal IT helpdesk assistant...
    """)
```

- Overrides the default system message.
- Useful for changing the assistant's role dynamically (e.g., HR instead of IT).



Understanding Message Roles in LLMs

eazy bytes

Sending a User Message

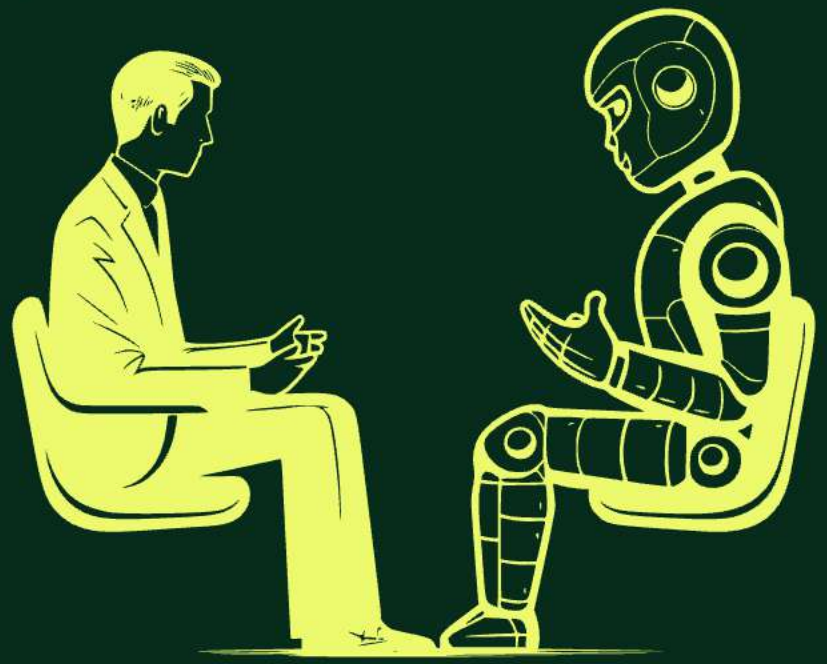
```
chatClient.prompt()
    .user(message)
```

- This is the actual input from the user (e.g., "Can I apply for leave during my probation?")

Return Assistant's Reply

```
return this.chatClient.prompt()
                .user(message)
                .call().content();
```

- Extracts and returns the LLM's generated response as plain text



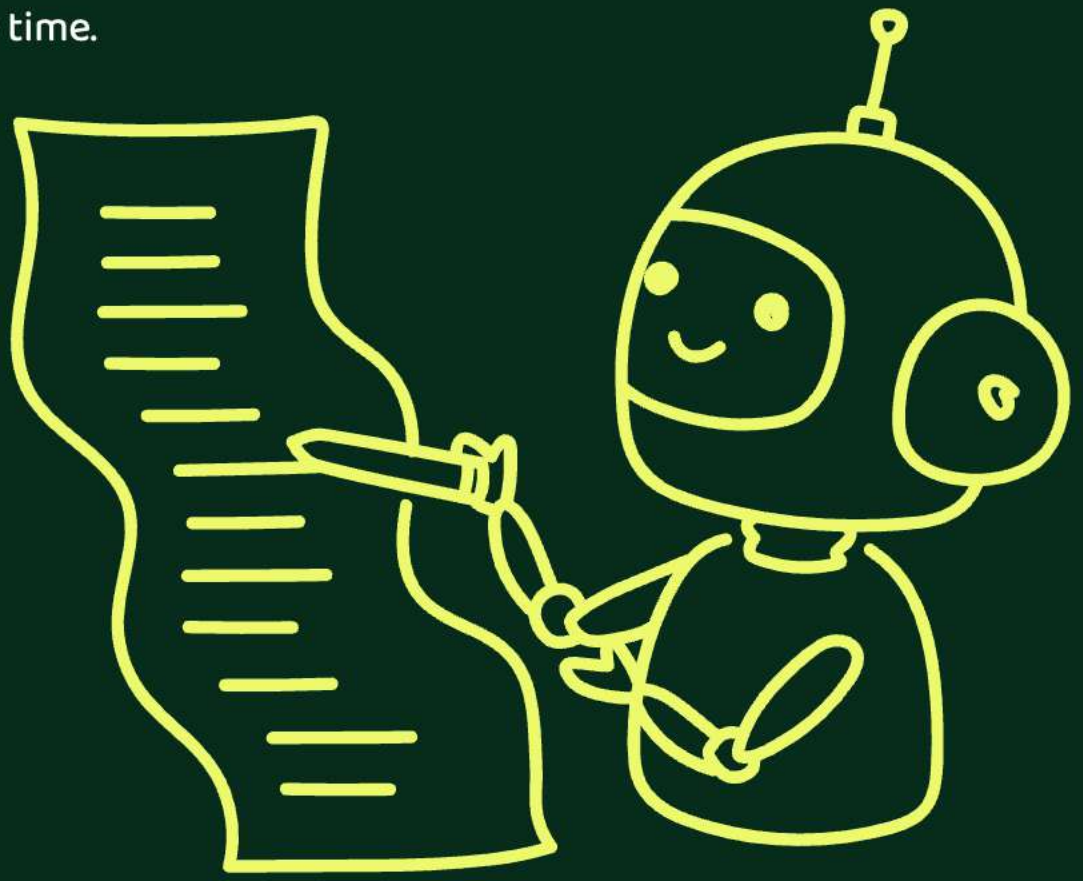
What Are Defaults in Spring AI?

eazy bytes

In Spring AI, defaults refer to preconfigured values or behaviors that are applied automatically to each request made through the ChatClient — unless overridden.

These defaults are useful when:

- You want to avoid repeating the same system message or advisor every time.
- You have a consistent role or tone for your assistant.
- You want consistent logging or tool behavior across calls.



Common Defaults You Can Set

defaultSystem(...) - Defines a default system message — which sets the behavior or role of the AI assistant.

```
chatClientBuilder.defaultSystem("""
    You are a helpful assistant that answers
    HR-related queries.
""")
```

What Are Defaults in Spring AI?

eazy bytes

defaultAdvisors(...) - Registers advisors like logging or custom filters that apply to each interaction.

```
chatClientBuilder.defaultAdvisors(new SimpleLoggerAdvisor())
```

defaultTools(...)- Registers tools/functions available to the LLM by default.

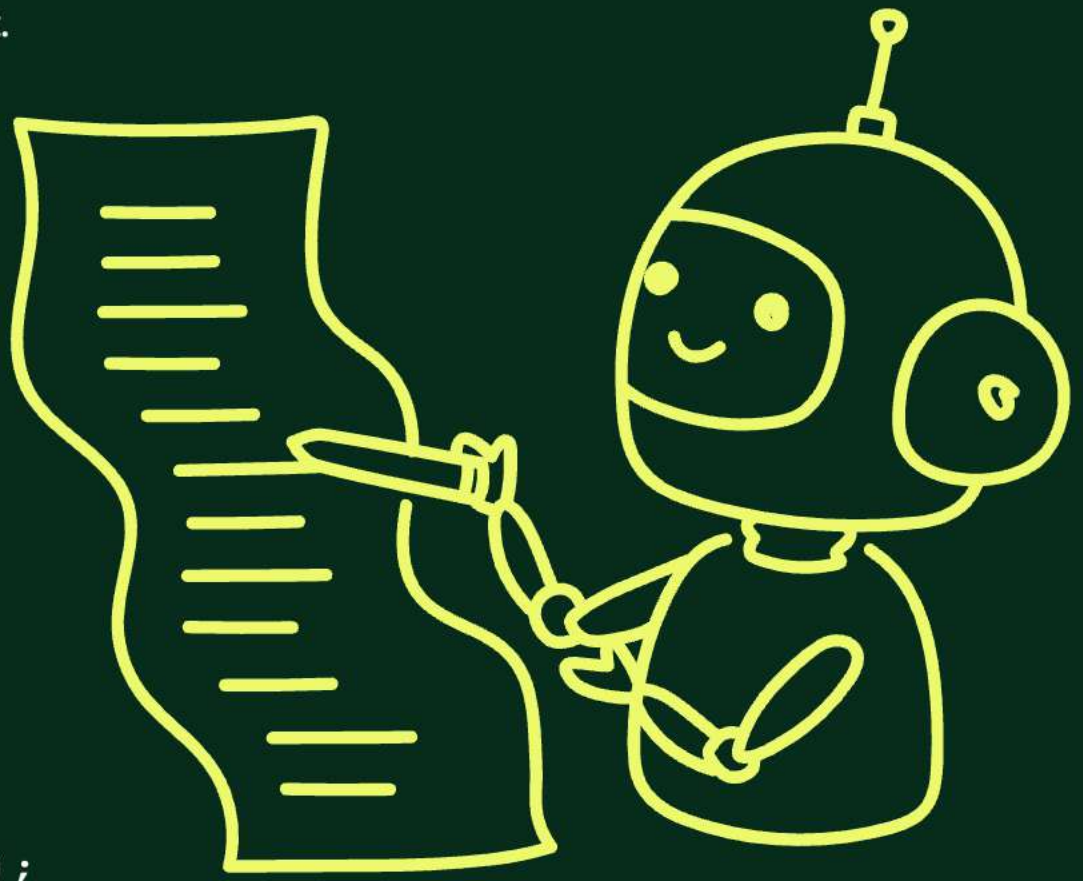
```
chatClientBuilder.defaultTools(
            List.of(myToolClassInstance))
```

defaultOptions(...) - Allows you to set default configuration options for the model request (e.g., temperature, maxTokens, etc.)

```
chatClientBuilder
    .defaultOptions(ChatOptions.builder().
       temperature(0.3).maxTokens(300).build())
```

defaultUser(...) - Sets a default user message that will be included in every prompt unless overridden.

```
chatClientBuilder.defaultUser("How can you help me ?");
```

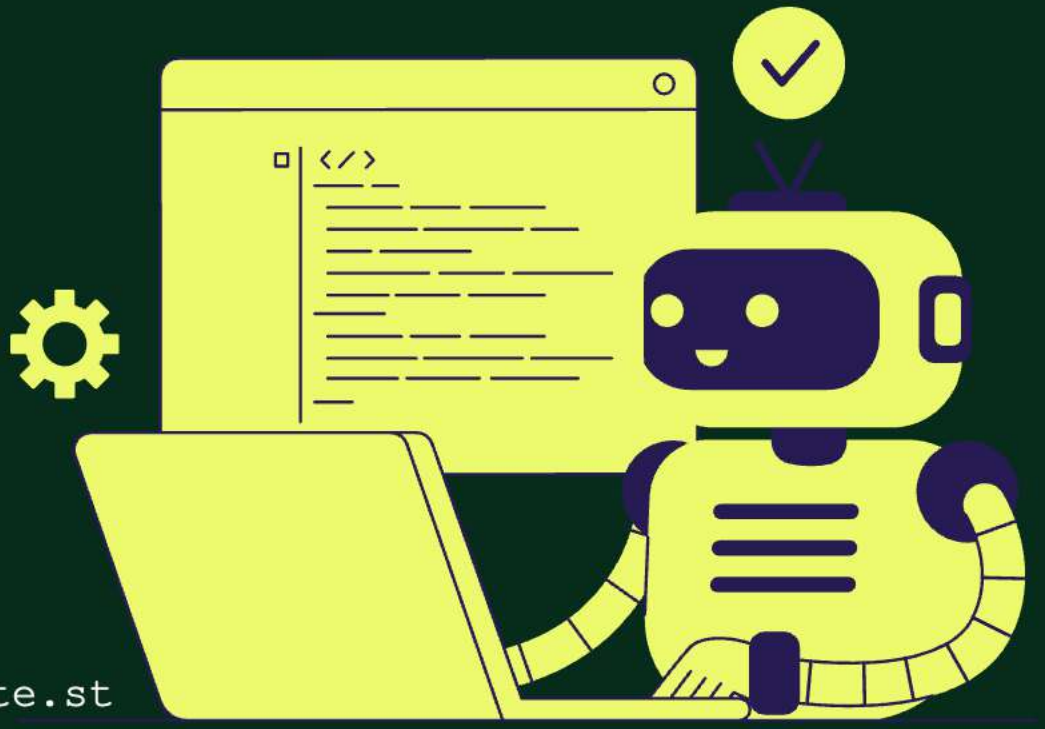


Using Prompt Templates in Spring AI

eazy bytes

Why Use Prompt Templates?

- Simplifies prompt construction
- Makes prompts reusable and maintainable
- Keeps logic and text cleanly separated
- Supports parameterized placeholders (like {customerName})



Where can Prompt Templates Live ?

You can store them in:

```
src/main/resources/promptTemplates/userPromptTemplate.st
File extension: .st (StringTemplate)
```

How to Use in Code

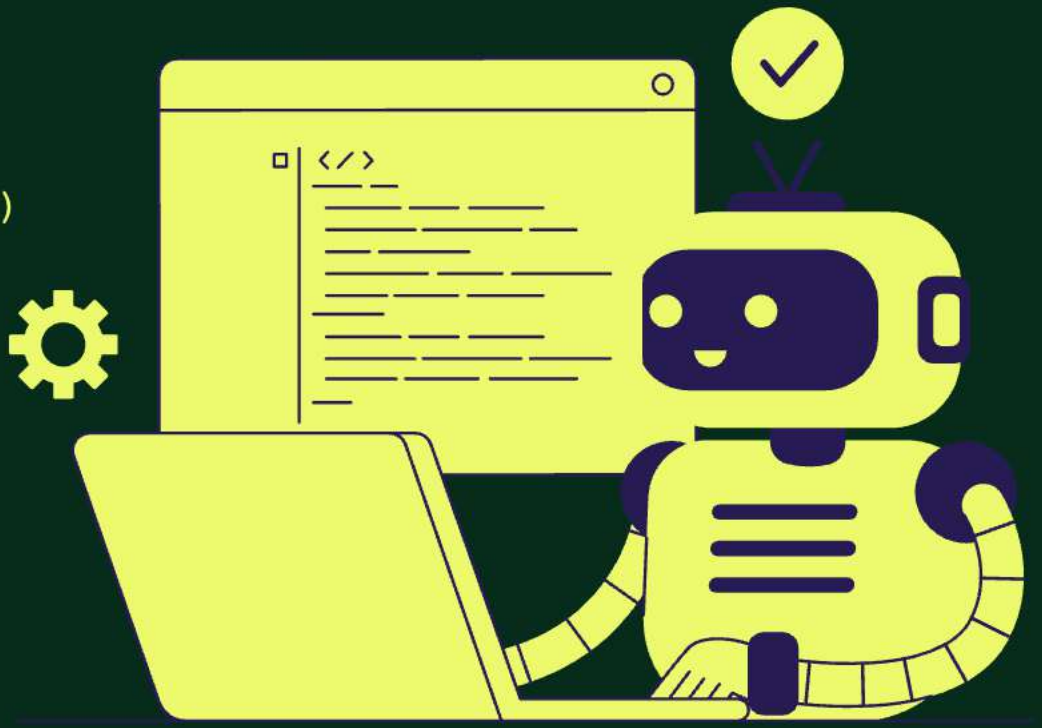
```
@Value("classpath:/promptTemplates/userPromptTemplate.st")
Resource userPromptTemplate;
```

Using Prompt Templates in Spring AI

eazy bytes

Use with .user() Prompt Builder

```
chatClient.prompt()
    .user(promptUserSpec ->
        promptUserSpec.text(userPromptTemplate)
            .param("customerName", customerName)
            .param("customerMessage", customerMessage))
    .call().content();
```

Prompt Stuffing

eazy bytes

Prompt Stuffing = Giving the LLM an open book before answering a question.

- You include contextual data or reference text along with the user's question.
- The LLM uses this extra content to answer the question accurately — even if it was not pre-trained on the topic.
- This technique is also known as in-context learning or retrieval-augmented prompting (when done programmatically).

Example: Internal company policy lookup

Prompt (Stuffed):

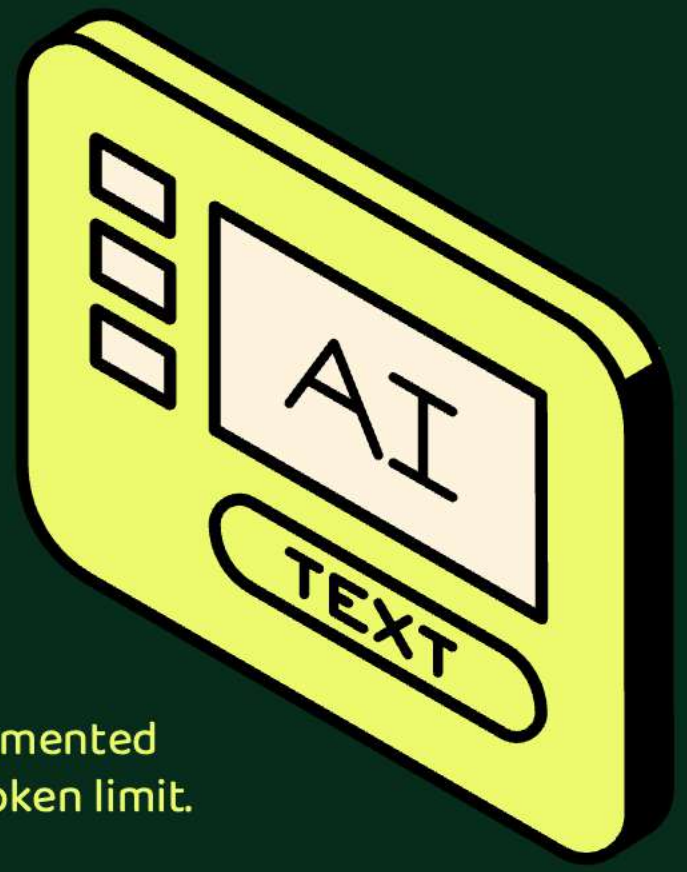
```
"According to the company's HR policy, employees are
eligible for 18 days of paid leave annually. Unused leave
can be carried over to the next year."
```

Question:

```
How many paid leaves do employees get each year?
```

LLM Answer:

```
Employees are eligible for 18 days of paid leave annually.
```

In the coming sections, we will learn how to apply a technique called Retrieval Augmented Generation (RAG) to provide relevant context in prompts without exceeding the token limit.

Configuring Advisors

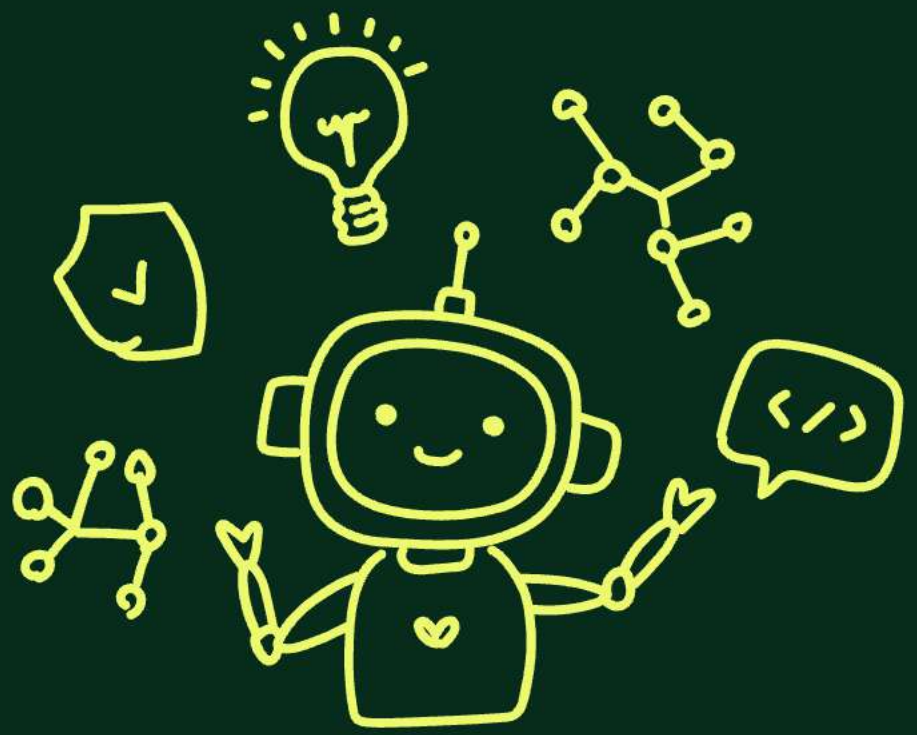
eazy bytes

What Are Advisors?

In Spring AI, advisors are like interceptors or middleware for your prompt flow.

They allow you to:

- Pre-process or post-process prompt data
- Add custom logging or auditing
- Inject additional behavior without modifying core logic
- Chain multiple behaviors cleanly



Where Advisors Fit

```
User → ChatClient → [Advisors] → LLM → Response → [Advisors] → User
```

Best Practices

- Keep advisors stateless or request-scoped
- Chain multiple advisors if needed
- Avoid altering the meaning of prompts unless intentional
- Use advisors for cross-cutting concerns, not core logic

Configuring Advisors

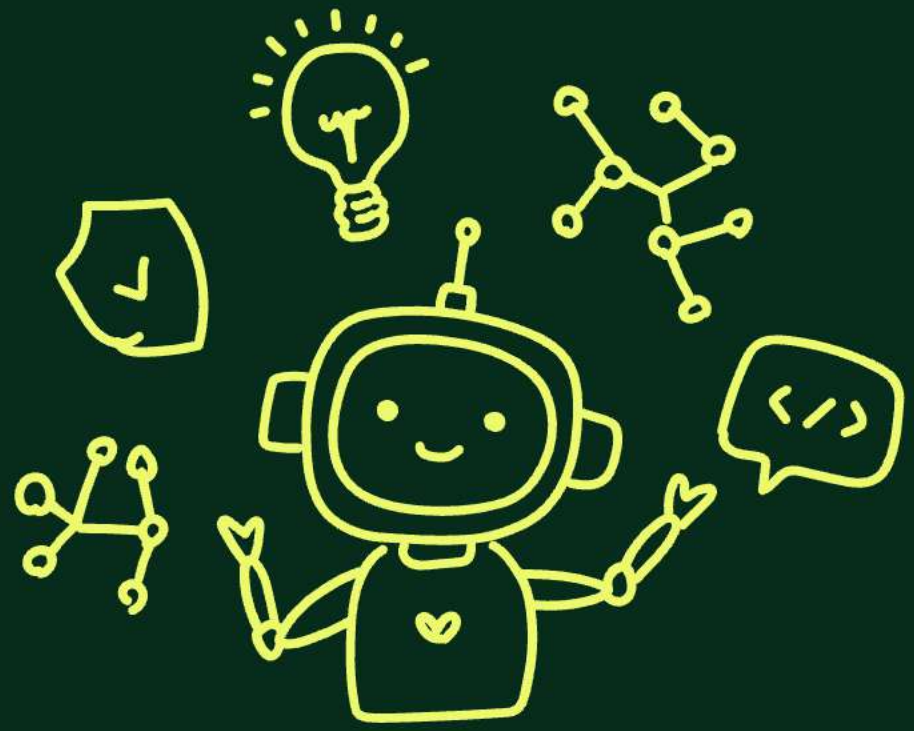
eazy bytes

Spring AI provides some built-in advisors, and you can create your own.

Built-in Example: SimpleLoggerAdvisor, SafeGuardAdvisor, PromptChatMemoryAdvisor etc.

How can advisors configured ?

```
chatClientBuilder
    .defaultAdvisors(new SimpleLoggerAdvisor())
    .build();

chatClient
     .prompt()
    .advisors(new SimpleLoggerAdvisor())
    .user(message)
    .call().content();
```

We can implement our own advisor by implementing the CallAdvisor, StreamAdvisor interfaces

Understanding ChatOptions in Spring AI

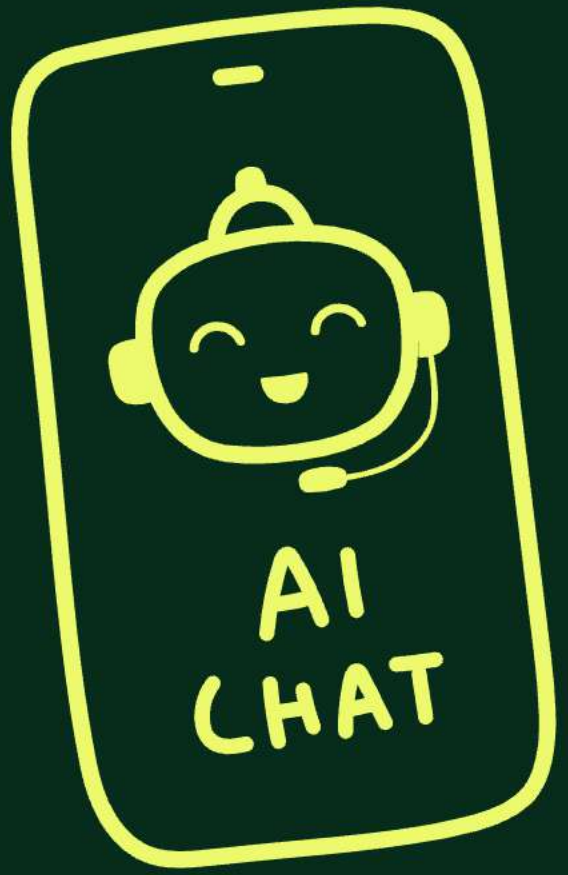
eazy bytes

What is ChatOptions?

ChatOptions is a configuration in Spring AI that allows you to customize how a language model behaves during chat/completion calls. Think of it like a "tuning panel" for your AI model — you can set limits, adjust creativity, randomness, verbosity, control response length, and more.

Key Chat Options

Option	Meaning
model	Which LLM model to use (e.g., gpt-4, gpt-3.5-turbo, etc.)
frequencyPenalty	Reduces repetition. Higher = less repetition
presencePenalty	Encourages mentioning new topics
temparature	Controls creativity. 0 = focused, 1 = random
topP	Controls randomness (nucleus sampling)
stopSequences	Stop generating when specific phrases are found
maxTokens	Maximum number of tokens (words/chars) in the reply
topK	controls how many top choices are considered

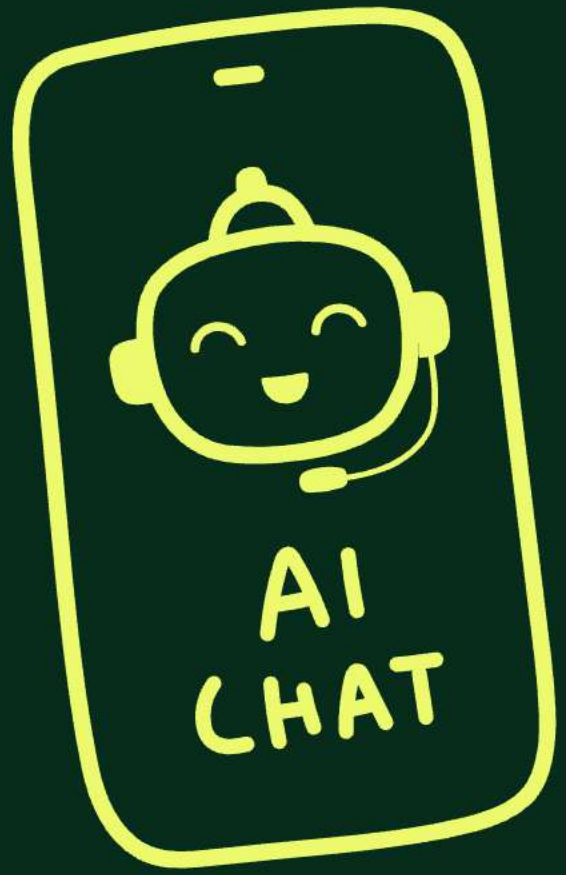


Understanding ChatOptions in Spring AI

eazy bytes

Example: How to Use in Builder Style

```
ChatOptions options = ChatOptions.builder()
    .model("gpt-3.5-turbo").temperature(0.3).maxTokens(200)
    .presencePenalty(0.6).stopSequences(List.of("END")).build();

chatClientBuilder
      .defaultOptions(options).build();

chatClient
      .prompt().options(options)
      .user(message).call().content();
```

ChatOptions is a generic interface in Spring AI. It defines a common set of configurable parameters that apply across different LLM providers (like OpenAI, HuggingFace, etc.). Helps make the code portable and consistent when switching providers.

OpenAiChatOptions is a concrete implementation of ChatOptions. It provides OpenAI-specific capabilities on top of the generic ones. Used when interacting with OpenAI models via Spring AI.

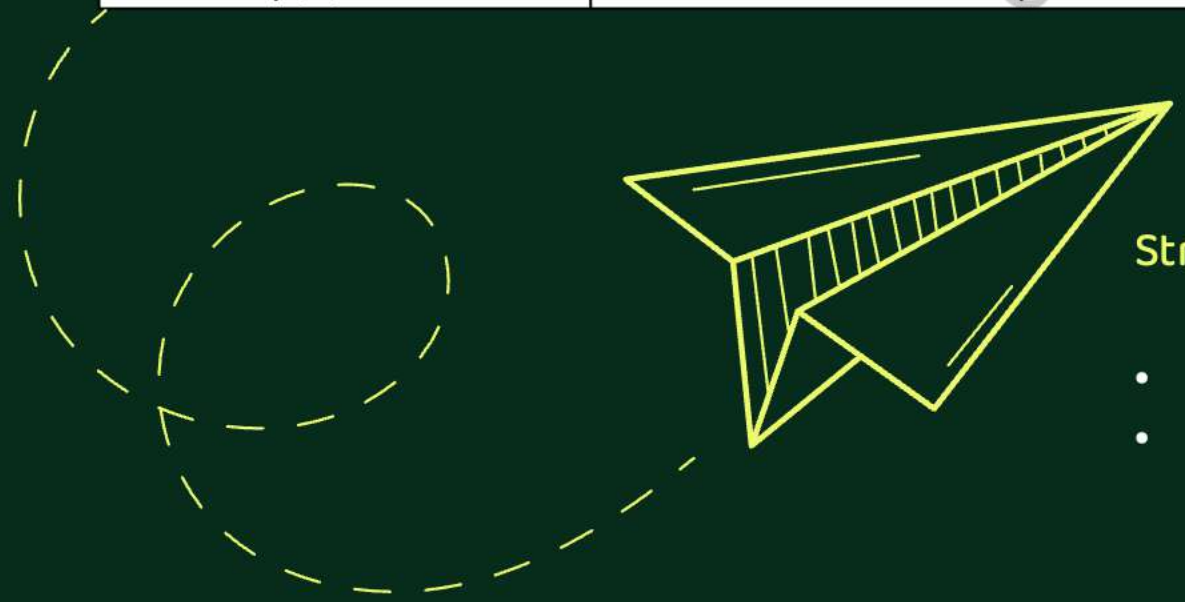
Spring AI ChatClient – Response Types Explained

eazy bytes

When you invoke .call() on a ChatClient, there are different ways to get the result, depending on what you want to do with it.

Basic options

Method	What It Returns	Use Case
content()	Just the response as a String	Simple use case - display or print reply
chatResponse()	A ChatResponse object	Get full details like token usage
chatClientResponse()	A ChatClientResponse object	Useful in RAG - includes context & metadata
entity(...) methods	Converts response to POJOs	Getting Java Objects (Structured Output)



Streaming Responses

- To stream reponses, we can use .stream() instead of .call()
- Good for real-time or chunked responses (like streaming output to UI)

Structured Output Converter in Spring AI

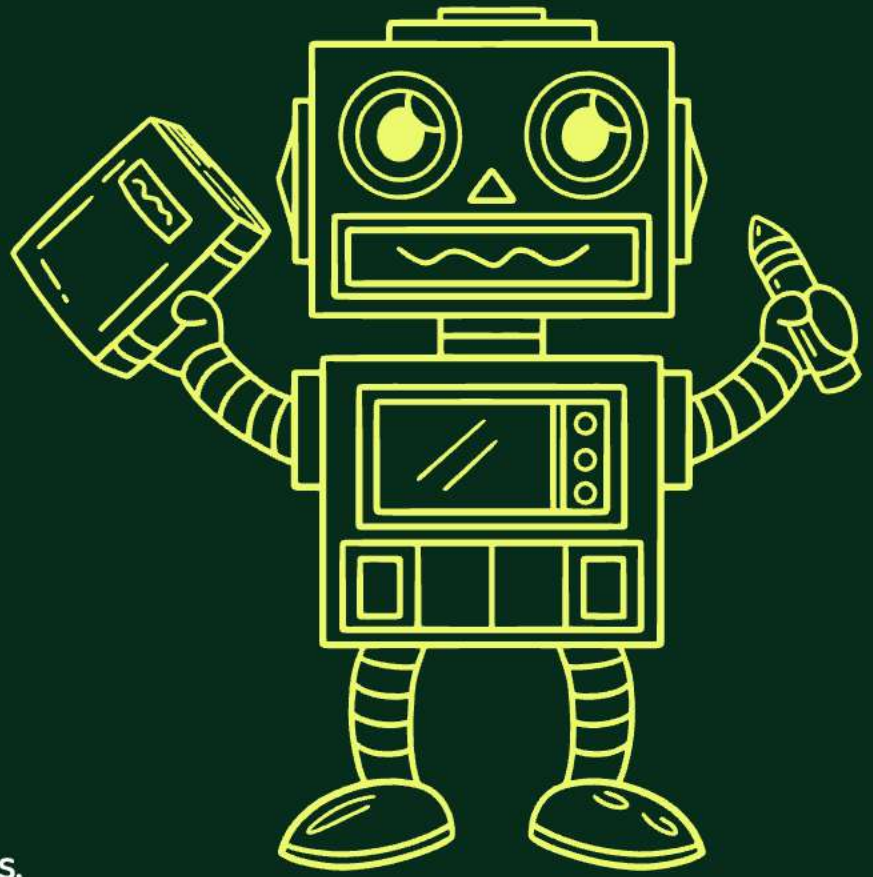
eazy bytes

Why Do We Need Structured Output?

LLMs typically return plain text responses. But in real-world apps, we often need structured data like,

- JSON
- XML
- Java Classes (POJOs)

Structured data is easier to parse, use, and integrate in applications.



What is a Structured Output Converter?

A helper that does two key things:

Before sending prompt to LLM:

➤ Adds formatting instructions to guide the model.

➤ Ensures it replies in a parseable format.

After getting response from LLM:

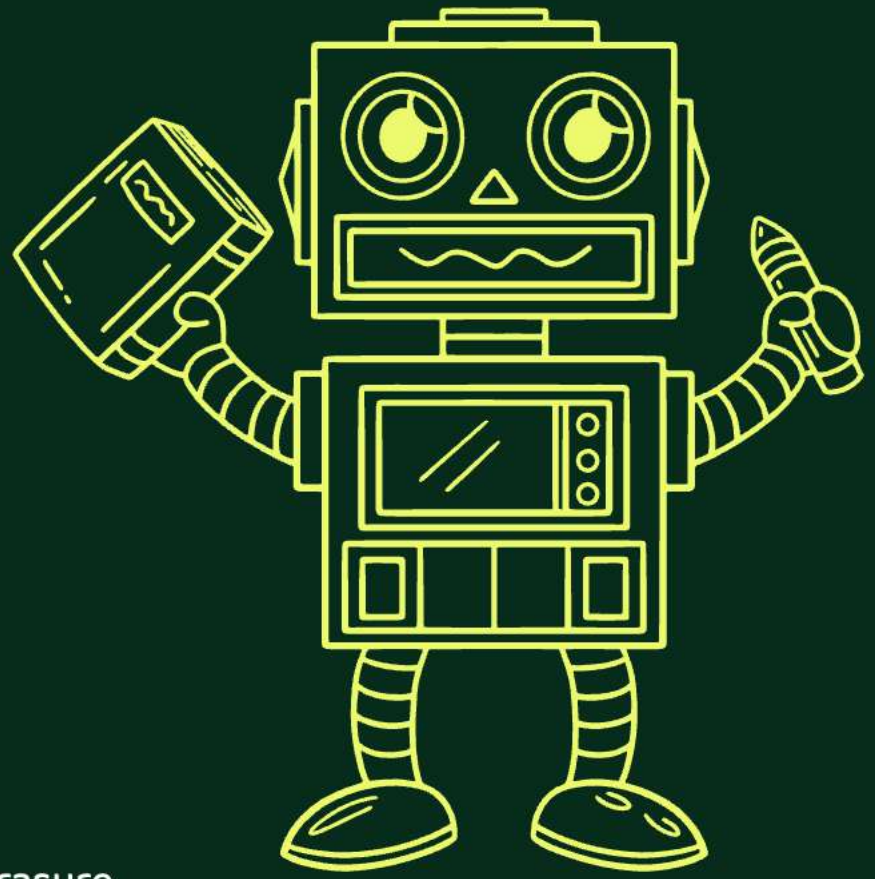
➤ Converts raw text into a Java object, like Map, List, or custom class.

Structured Output Converter in Spring AI

eazy bytes

Example: Convert reponse to a Java object

```
CountryCities countryCities = this.chatClient.prompt()
                .user(question)
                .call()
                .entity(CountryCities.class);
```



Example: Convert reponse to a list of Java objects

```
List<CountryCities> countryCitiesList =
                  this.chatClient.prompt()
                .user(question).call()
                .entity(new ParameterizedTypeReference
                        <List<CountryCities>>() {});
```

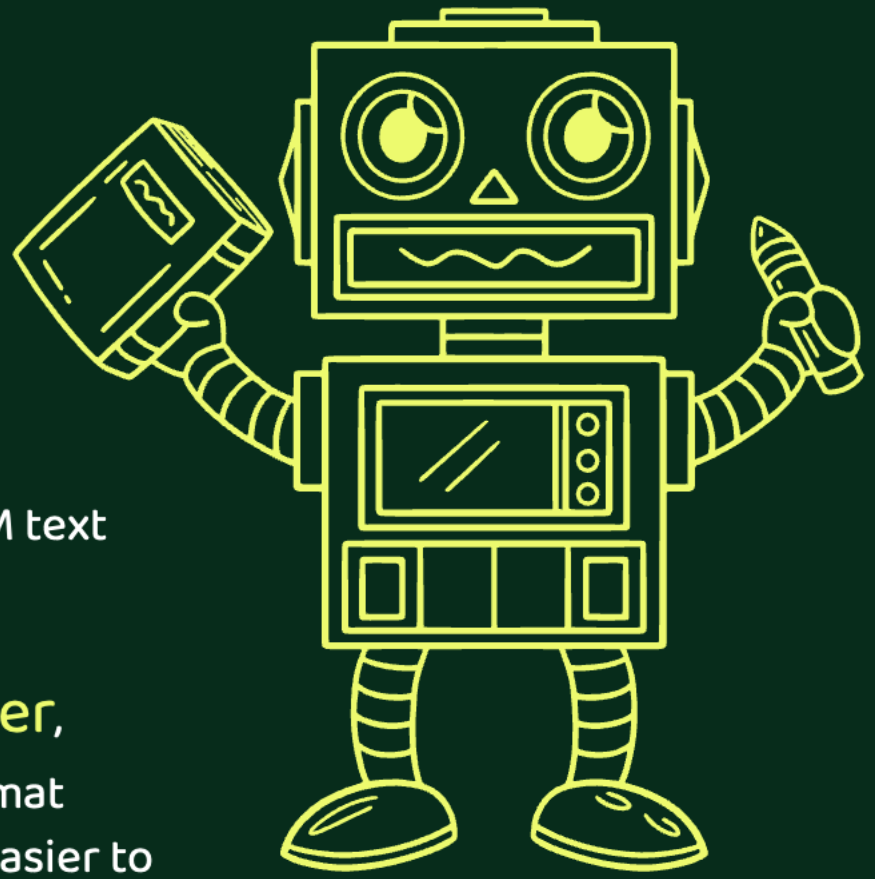

We use ParameterizedTypeReference<List<CountryCities>> because Java's type erasure removes generic type information like List<CountryCities> at runtime, making it impossible to use List<CountryCities>.class. ParameterizedTypeReference preserves this type info so Spring can correctly deserialize the LLM response into a list of CountryCities objects.

Structured Output Converter in Spring AI

eazy bytes

Example: Convert reponse to a Map

```
Map<String, List<String>> countryMapList =
      this.chatClient.prompt()
      .user(question).call()
      .entity(new ParameterizedTypeReference
            <Map<String, List<String>>>() {});
```



- Spring AI's StructuredOutputConverter helps convert raw LLM text responses into structured Java data like objects, lists, or maps.
- We can use it's implementations as well like BeanOutputConverter, ListOutputConverter, and MapOutputConverter, add format instructions before the AI call and parse the response after, making it easier to work with typed data in your applications.

Journey to Generative AI

eazy bytes

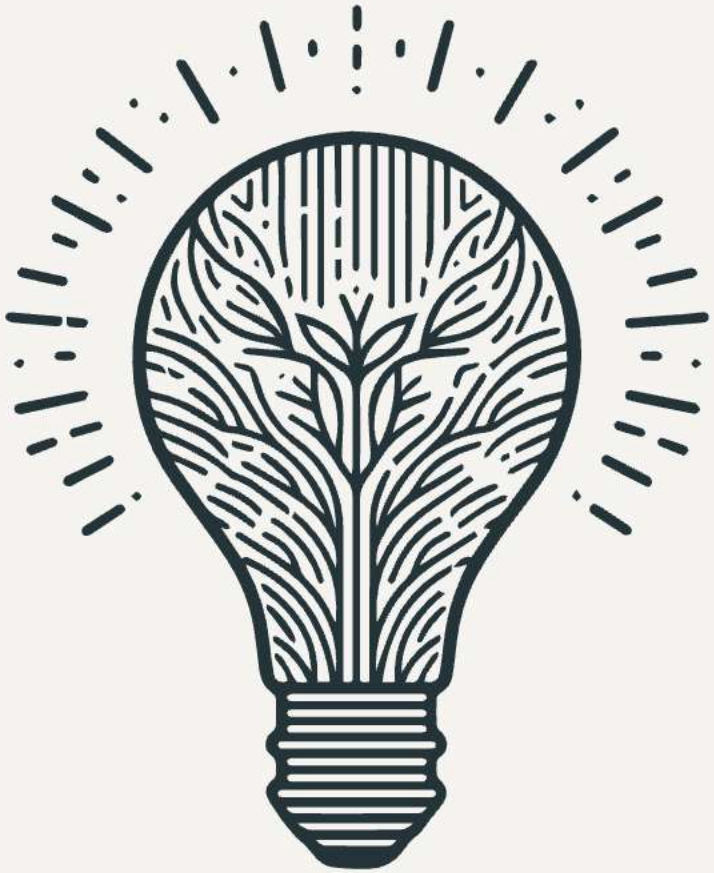
What is GenAI?

GenAI (Generative AI) is like a super-smart artist or writer that uses computers to create new things—like stories, pictures, music, or even ideas. Imagine a robot that can draw a picture or write a poem all by itself—that's what GenAI does!

The GenAI Family Tree

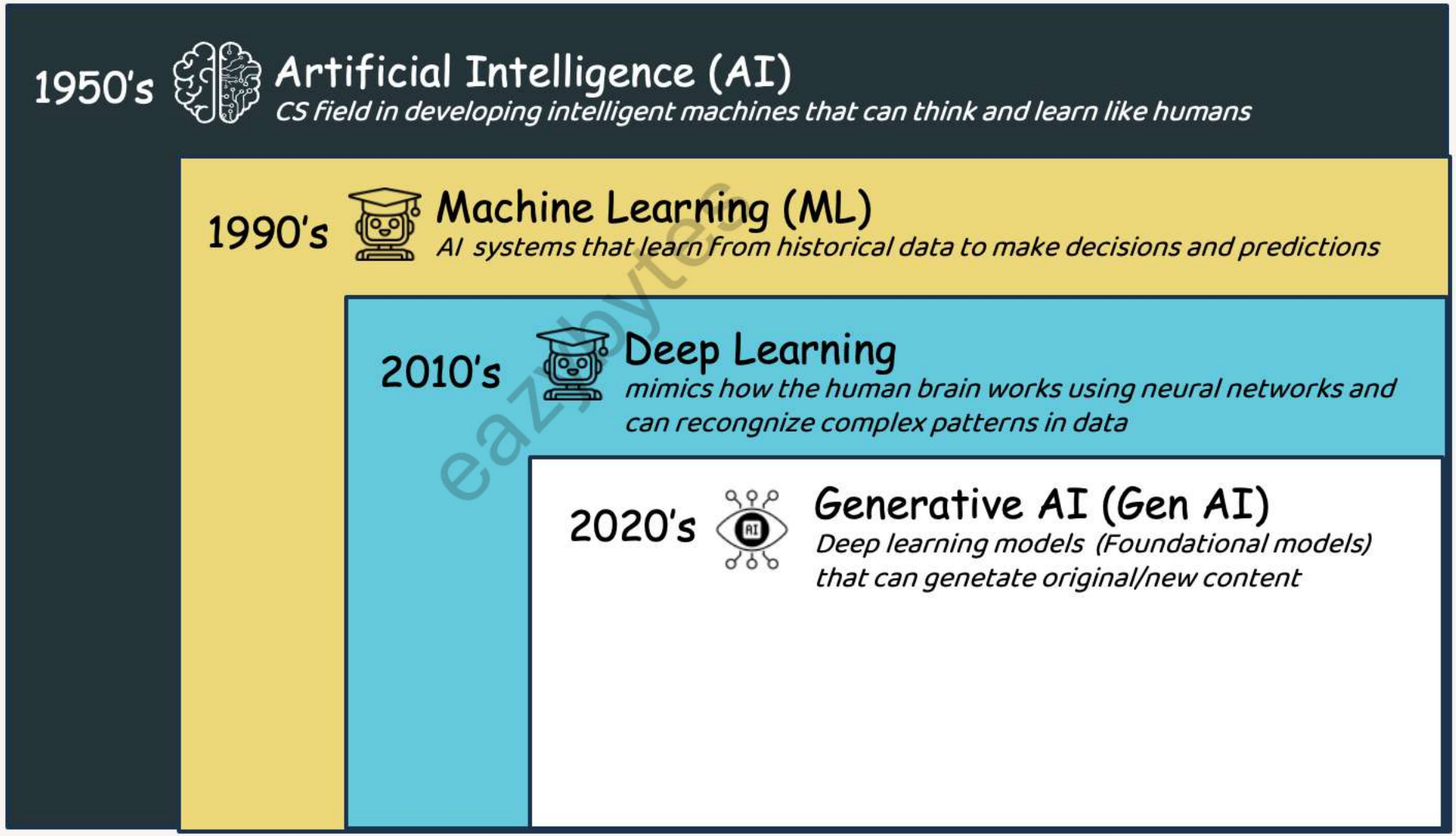
GenAI is part of a bigger family of technologies that help computers think and learn.

- Artificial Intelligence (AI)
- Machine Learning (ML)
- Deep Learning (DL)
- Neural Networks (NN)
- NLP (Natural Language Processing)
- GenAI (Generative AI)
- LLMs (Large Language Models)



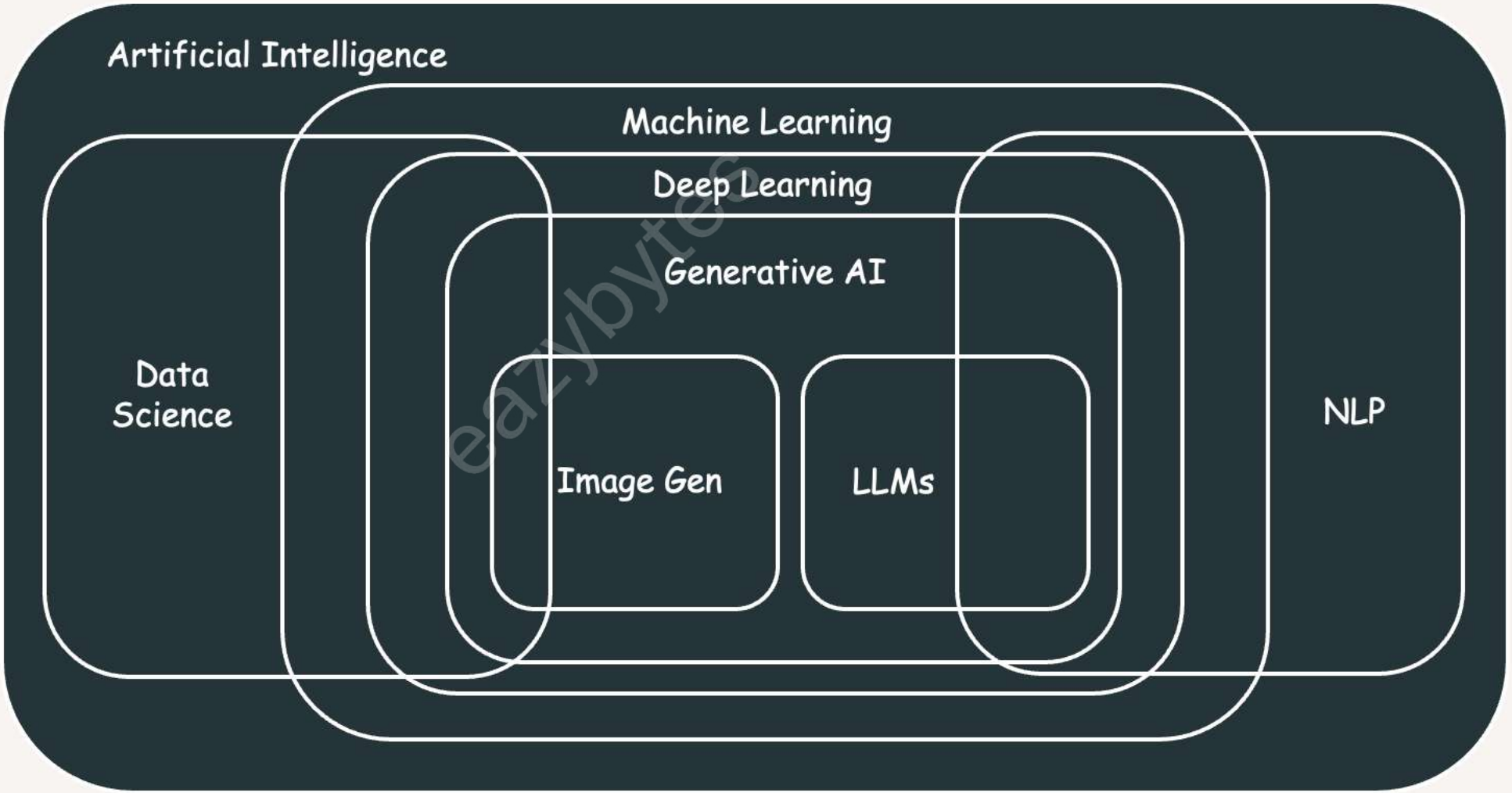
Journey to Generative AI

eazy bytes



Journey to Generative AI

eazy bytes



The Big Picture - What is Artificial Intelligence?

eazy bytes

🔍 What is Artificial Intelligence (AI)?

AI means making computers behave like humans — thinking, learning, solving problems, or making decisions.

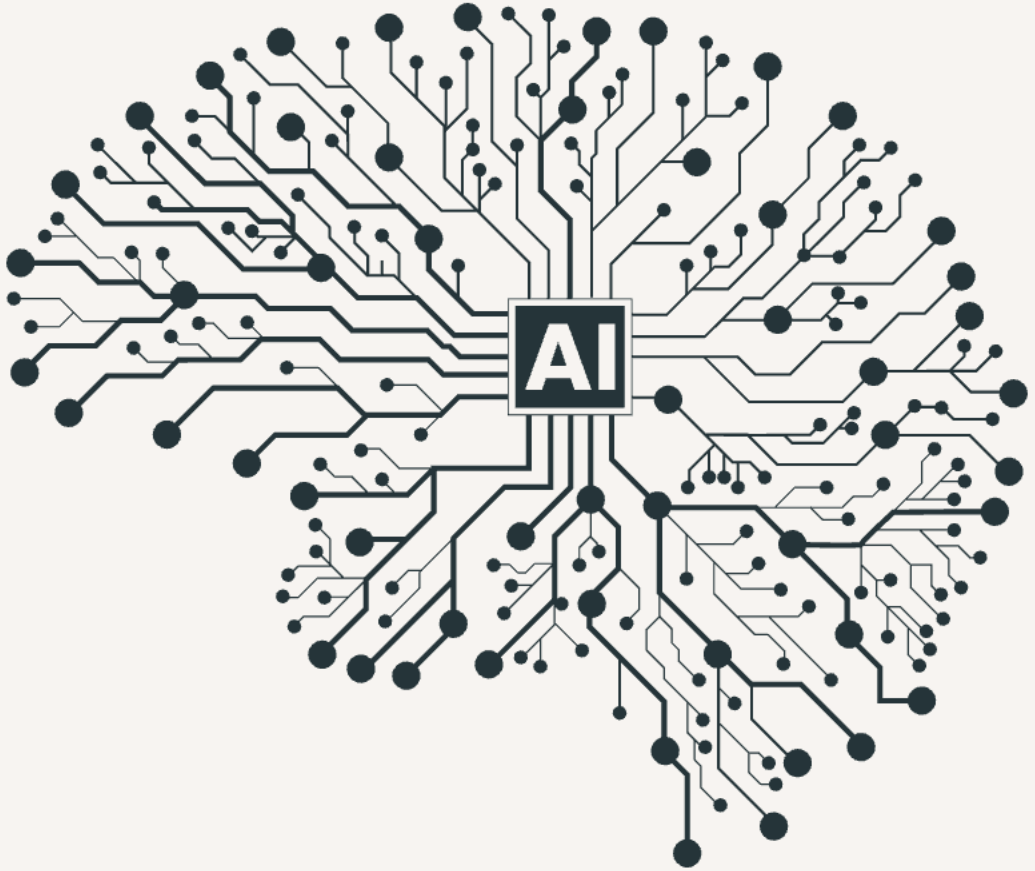
It's like giving machines a "brain" that can learn, reason, and make decisions

Examples you use daily:

Siri, Google Assistant, A self-driving car using sensors and rules to drive safely.

Key Point:

AI is the umbrella term for all smart computer systems



🤖 What is Machine Learning (ML)?

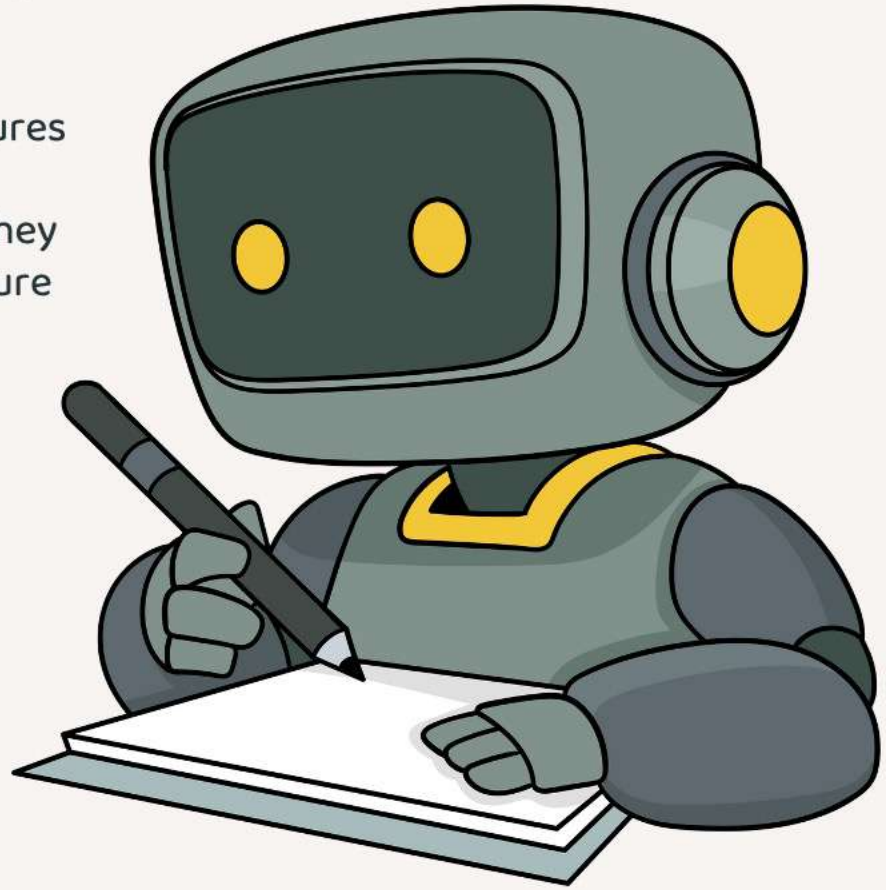
ML is a smaller part of AI. It's like teaching a computer to learn from examples, not by giving it strict rules.

Imagine teaching a child to recognize cats by showing them thousands of cat pictures

Machine Learning works the same way - we show computers lots of examples so they can learn patterns. Instead of programming specific rules, we let the computer figure out the patterns on its own

Real-world Examples:

- Email spam detection (learns what spam looks like)
- Online shopping recommendations (learns your preferences)
- Voice recognition (learns to understand different accents)



Key Point:

ML is how computers learn from experience, just like humans do

Deep Learning - The Brain-inspired Approach

eazy bytes

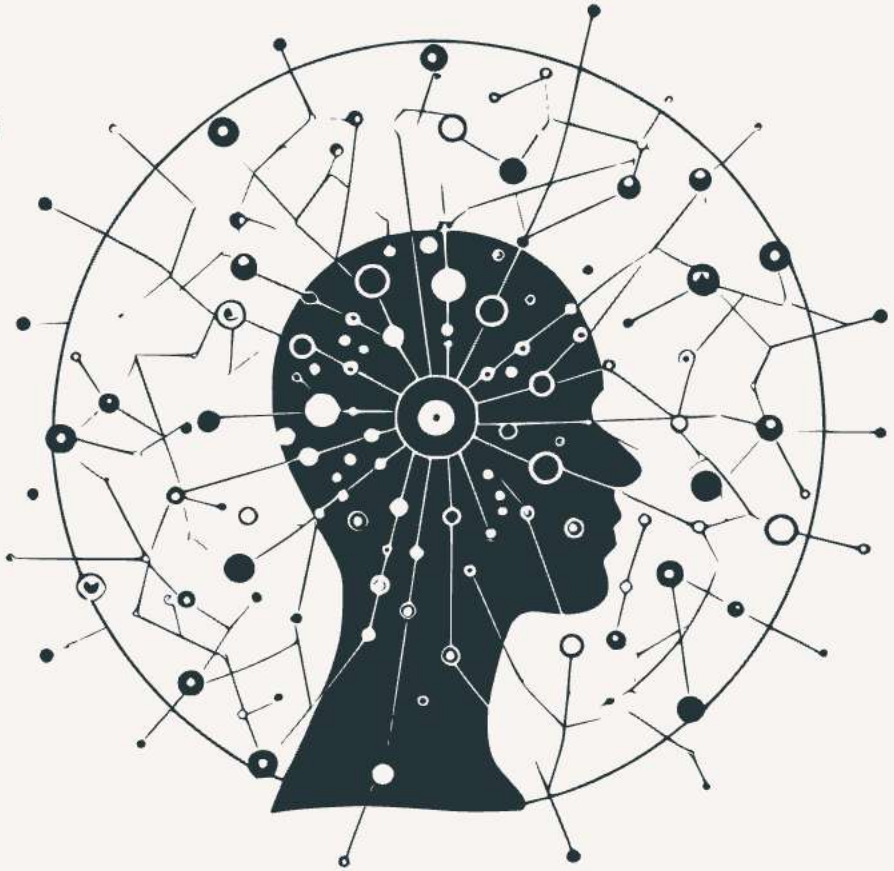
🧠 What is Deep Learning (DL)?

DL is a more advanced form of machine learning that mimics how the human brain works using structures called neural networks.

- Inspired by how our brain works with interconnected neurons
- Uses artificial "neural networks" with many layers (that's why it's called "deep")
- Each layer learns different features - like edges, shapes, then complete objects

Real-world Examples:

- Image recognition (Google Photos organizing your pictures)
- Language translation (Google Translate)
- Self-driving cars (recognizing traffic signs and pedestrians)



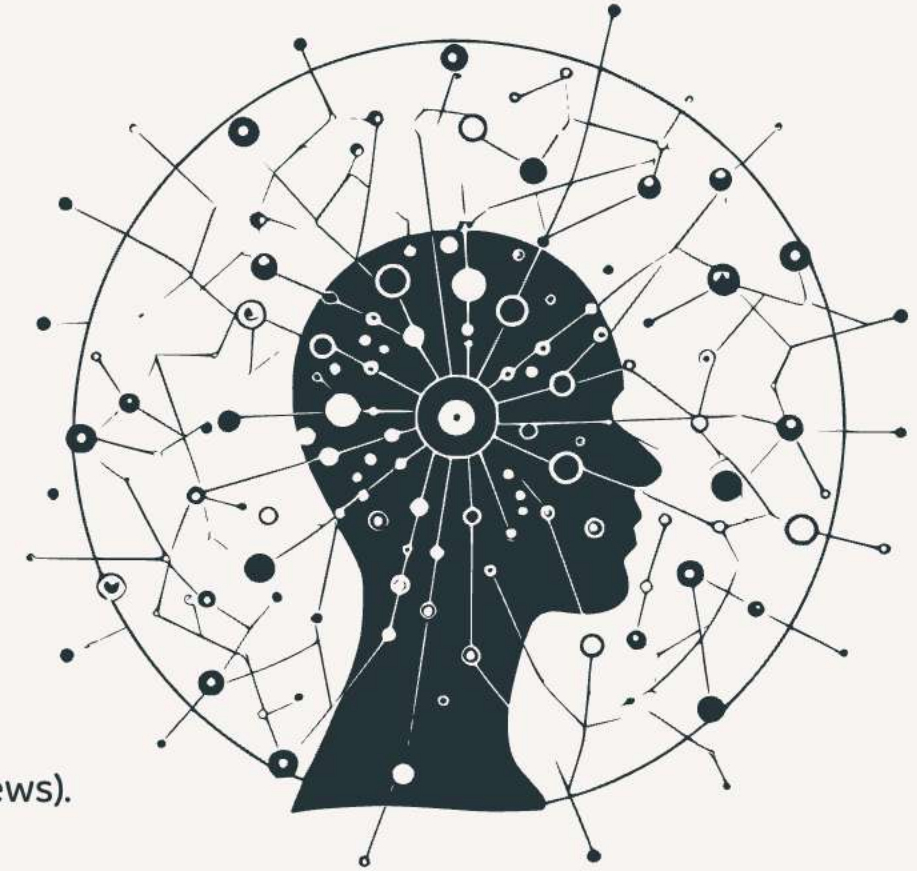
Deep Learning - The Brain-inspired Approach

eazy bytes

Simple Analogy on how Deep Learning helps learning:

Layer 1: Recognizes lines and edges
Layer 2: Combines lines to see shapes
Layer 3: Combines shapes to recognize objects
Final Layer: "This is a cat!"

Just like it learns edges → shapes → objects in images, Deep Learning learns letters → words → meanings → intent in text.



Step-by-step Analogy for Text:

Lower layers learn grammar (e.g., "I am" vs. "He is").
Middle layers learn word relationships (e.g., "bank" = river bank or money bank?).
Higher layers learn meaning and intent (e.g., sentiment: happy/sad, or topic: sports/news).

Key Point:

Just like our brain understands not just the words but the meaning behind them, deep learning models learn how language works – context, emotions, structure – all automatically

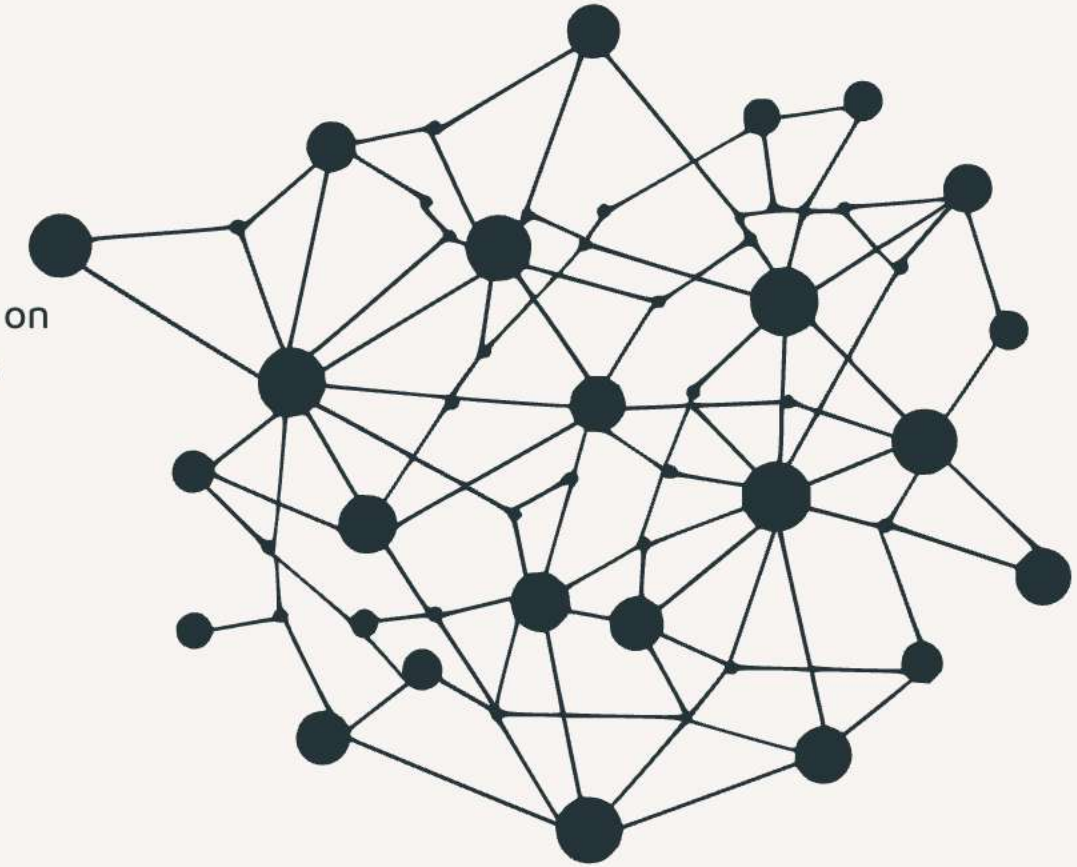
Neural Networks - The Building Blocks

eazy bytes

🧩 How Neural Networks (NN) Fit In

Neural networks are the building blocks of deep learning.

- Computer systems designed to work like brain cells (neurons)
- Each "artificial neuron" receives information, processes it, and passes it on
- Connected in networks where each connection has different strengths



Why They Matter:

- They're the foundation that makes Deep Learning possible
- They can find patterns humans might miss
- They get better with more data and practice

Natural Language Processing - Understanding Human Language

eazy bytes

💬 What is NLP (Natural Language Processing)?

NLP is a branch of AI that helps machines understand and work with human language — reading, writing, and talking.

- The technology that helps computers understand and work with human language
- Bridges the gap between how humans communicate and how computers process information
- Makes it possible for us to talk to computers naturally

Key Capabilities:

- Understanding: What does this sentence mean?
- Generation: How do I express this idea in words?
- Translation: How do I say this in another language?
- Sentiment: Is this comment positive or negative?

Daily Examples:

- Voice assistants understanding your questions
- Email auto-complete suggestions
- Language translation apps
- Chatbots on websites`

Generative AI - The Creative Engine

eazy bytes

🎨 What is GenAI (Generative AI)?

- Generative AI (GenAI) is like a big box that can create brand new things.
  It's like a super creative artist! This box can draw new pictures, write new songs, or even make up new stories.

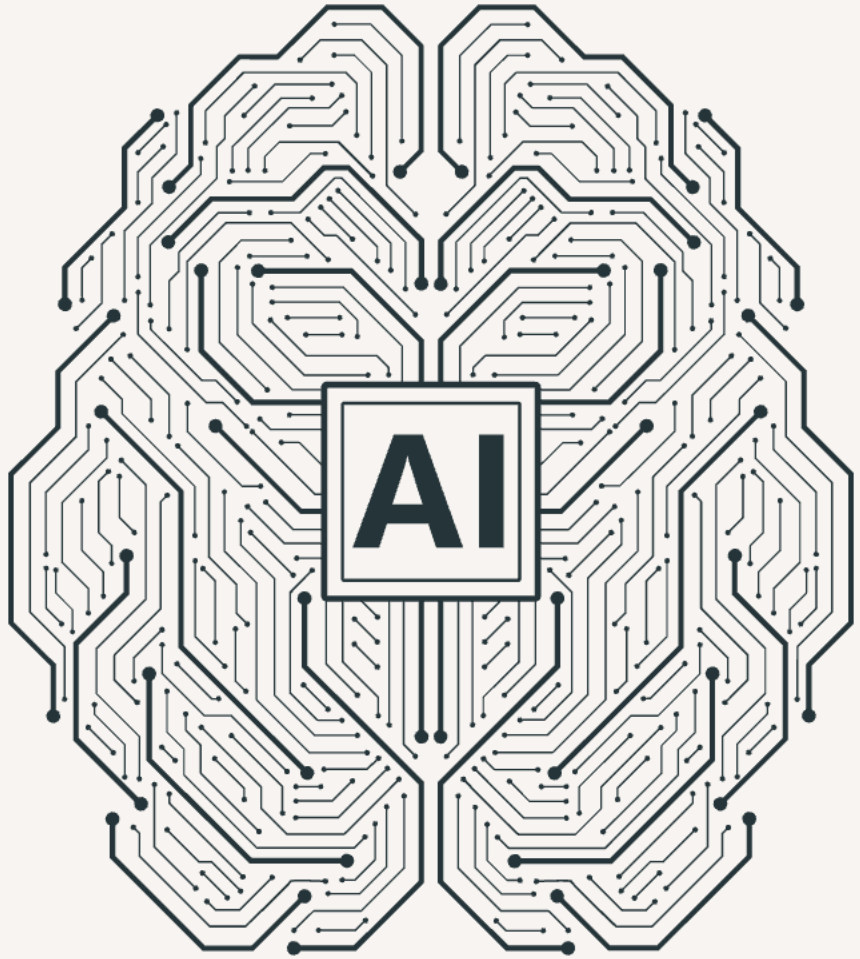
- It learns from lots of examples (like seeing many drawings, hearing many songs, or reading many stories) and then uses what it learned to invent something totally original.

GenAI (Big category of creative AI)

- LLM (Text)
- Image Generators (like DALL·E)
- Music Generators
- Video Generators

Popular Examples:

- DALL-E, Midjourney (image creation)
- ChatGPT, Claude (text generation)
- GitHub Copilot (code generation)
- Runway, Sora (video generation)



Real-World Analogy

eazy bytes

Imagine building a robot chef:

AI: The robot's ability to cook a meal.

ML: The robot learning recipes by watching cooking videos.

DL: The robot using a complex brain-like system to perfect its cooking.

NNs: The "wiring" in the robot's brain that processes ingredients and steps.

NLP: The robot understanding your spoken order ("Make pasta!").

GenAI: The robot inventing a new recipe.

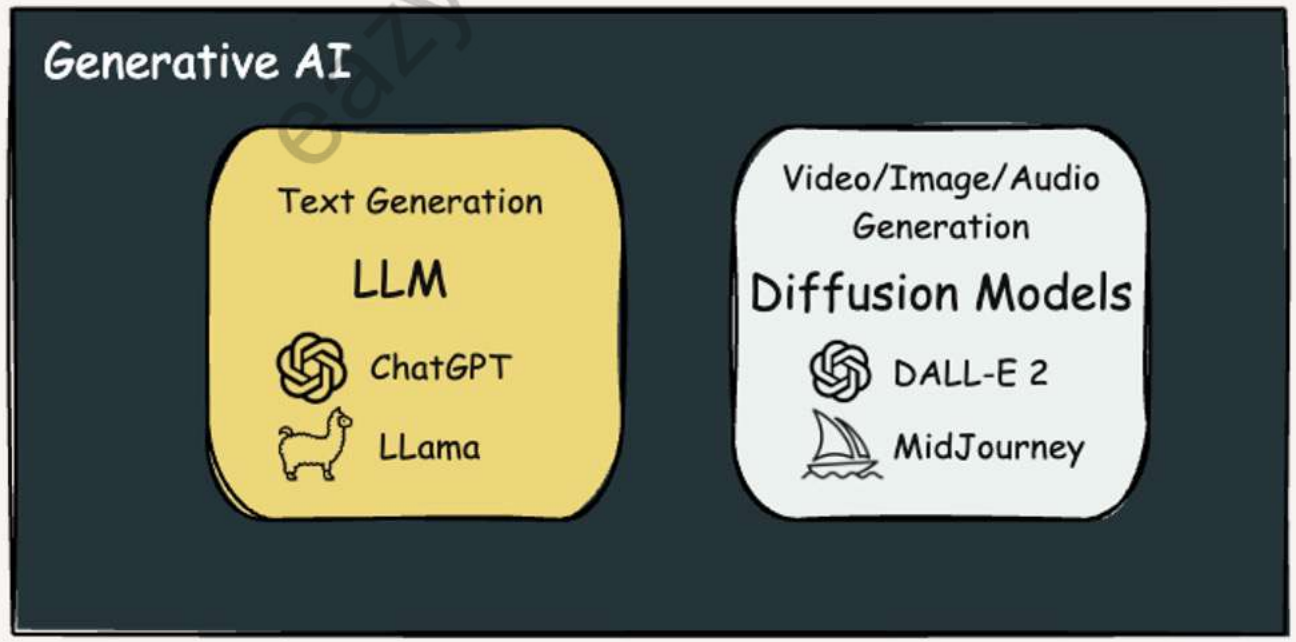
LLMs: The robot writing a cookbook or explaining the recipe in detail.

Types of Generative AI Models

LLMs and **Diffusion** models are two key technologies in generative AI.

For text generation, OpenAI's ChatGPT—trained on vast datasets like books, websites, and forums—has transformed how machines understand and generate language.

Diffusion models, used mainly for image and audio generation, learn to reverse added noise to produce high-quality outputs. In models like DALL·E 2, this process is guided by text prompts to create images that match natural language descriptions.



Large Language Models - The Text Experts

eazy bytes

🧠 What is a Large Language Model (LLM)?

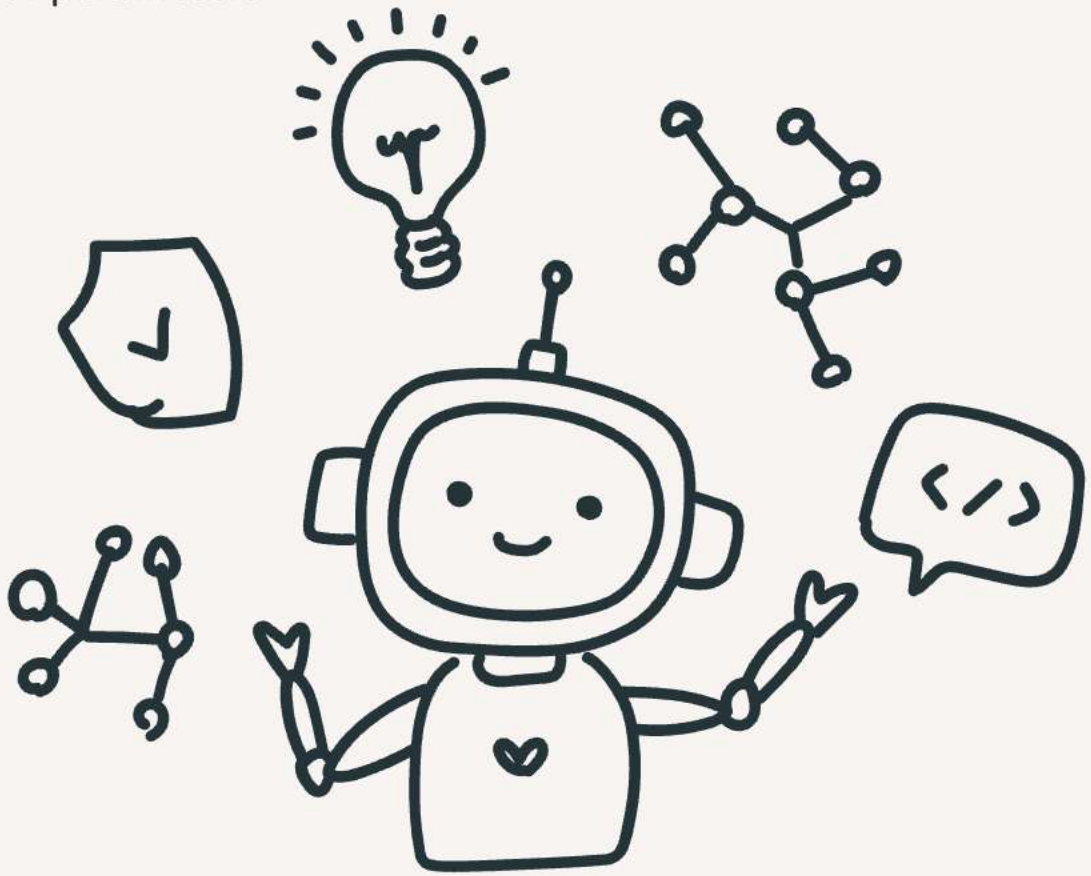
- LLMs are special AI systems trained on massive amounts of text from books, websites, and articles
- They understand and generate human-like text
- "Large" because they're trained on billions of words and have billions of parameters

📚 How Do LLMs Learn?

- Trained using unsupervised learning
- Learn patterns from billions of words and phrases
- No manual labels needed — models learn by observing language

⚙️ Why Do LLMs Need So Much Power?

- Continuously calculate probabilities and patterns in text
- Require massive computational power, especially during training
- Use GPUs for fast parallel processing (ideal for deep learning)



How big are these LLMs ?

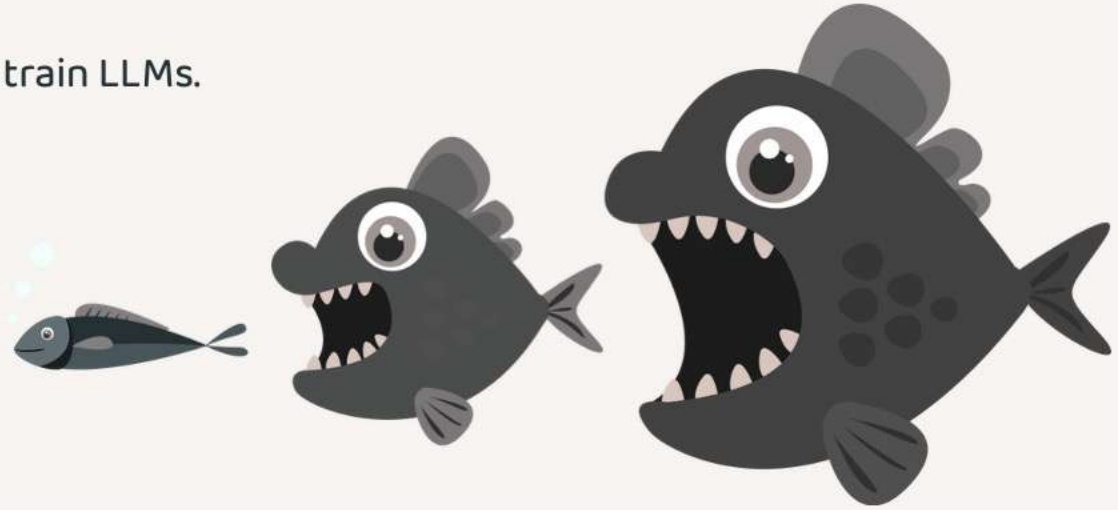
eazy bytes

GPT-4 is estimated to be trained on:

- 1 to 2 trillion tokens (roughly 300–500 billion words)
- Equivalent to reading:
  - ✓ Over 1 million books
  - ✓ Nearly the entire internet's text content
  - ✓ we're at risk of running out of publicly available internet data to train LLMs.

Few facts about LLMs training

- GPT-4 may have used 10,000 to 25,000 GPUs (NVIDIA A100 or H100)
- Training cost estimates range between $50M–$100M
- Training involved tens of thousands of petaflop-days of compute
- GPT-4 likely took several weeks to a few months to train.
- A human would take ~114 years reading 24/7 to consume the same amout of data as GPT-4



Why Are GPUs Used Instead of CPUs for LLMs ?

- CPUs have a few powerful cores (usually 4–16), optimized for general-purpose tasks.
- GPUs have thousands of smaller cores, perfect for doing many simple math operations at once.
- AI training requires massive matrix and vector operations — such as multiplying large vectors and matrices.
- GPUs can process these in parallel, whereas CPUs would do it sequentially and much slower.

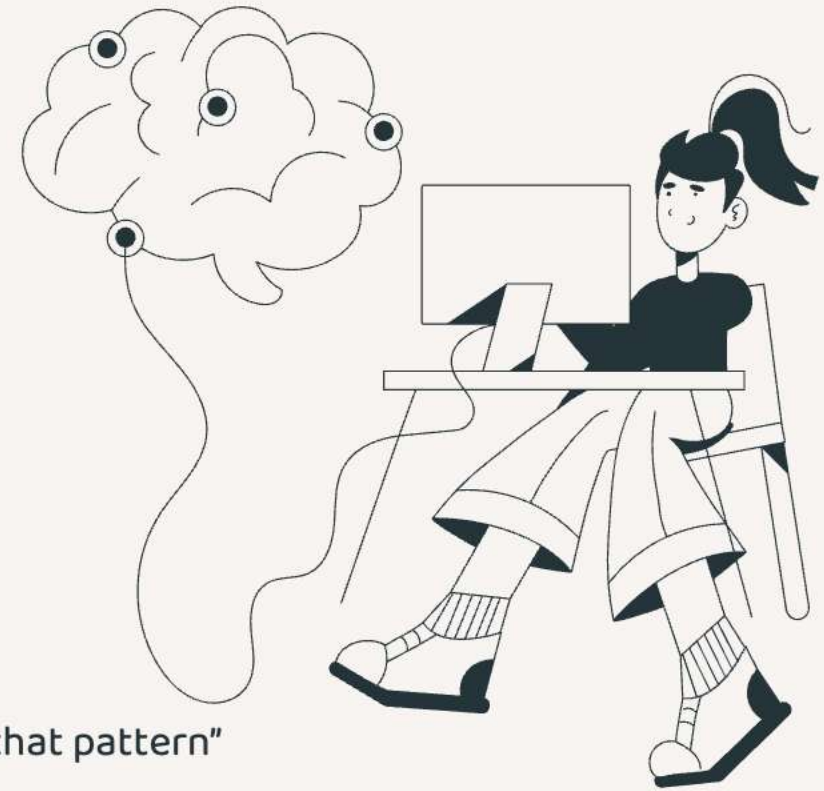
How LLMs Learn Without Supervision

eazy bytes

The Magic of Pattern Recognition

Think of it like learning to talk as a baby: Babies don't have teachers saying "This word comes next"
They just listen to TONS of conversations

Gradually, they notice patterns: "After 'Good' usually comes 'morning'" or
"After 'How' often comes 'are you?'"



How LLMs Do This ?

Imagine reading millions of books, websites, conversations
The LLM sees sentences like: "The cat sat on the..." thousands of times
It starts noticing: "Oh, after 'sat on the' usually comes words like 'mat', 'chair', 'roof'"

The LLM plays a constant guessing game with itself
It reads: "I'm going to the store to buy some..."
It tries to guess: "milk"? "bread"? "apples"?
Then it sees the actual next word and learns: "Oh, it was 'groceries' - let me remember that pattern"

Building Connections

Like a giant web in its "brain", it connects related ideas: "store" connects to "shopping", "money", "items", "Rain" connects to "umbrella", "wet", "clouds"

What is the core job of an LLM ?

eazy bytes

🧠 LLMs (Large Language Models) have one main job: Guessing the next word based on input provided. It's like a super smart autocomplete.

Example: If you type: `Once upon a` ,the LLM might guess: `time`

Then, How Do We Get Full Sentences?

LLM Wrapper (like ChatGPT):

- Makes LLMs user-friendly
- Generates full sentences
- Handles conversations
- Provides the interface you interact with

Analogy: LLM = Engine, Wrapper = Car

The LLM Wrapper uses The "Eating Its Own Output" Trick to get a full response from bare LLM

What is the core job of an LLM ?

eazy bytes

LLM wrappers keep sending the input prompt to LLM to guess the next word Until it hits a stop signal or reaches a limit.

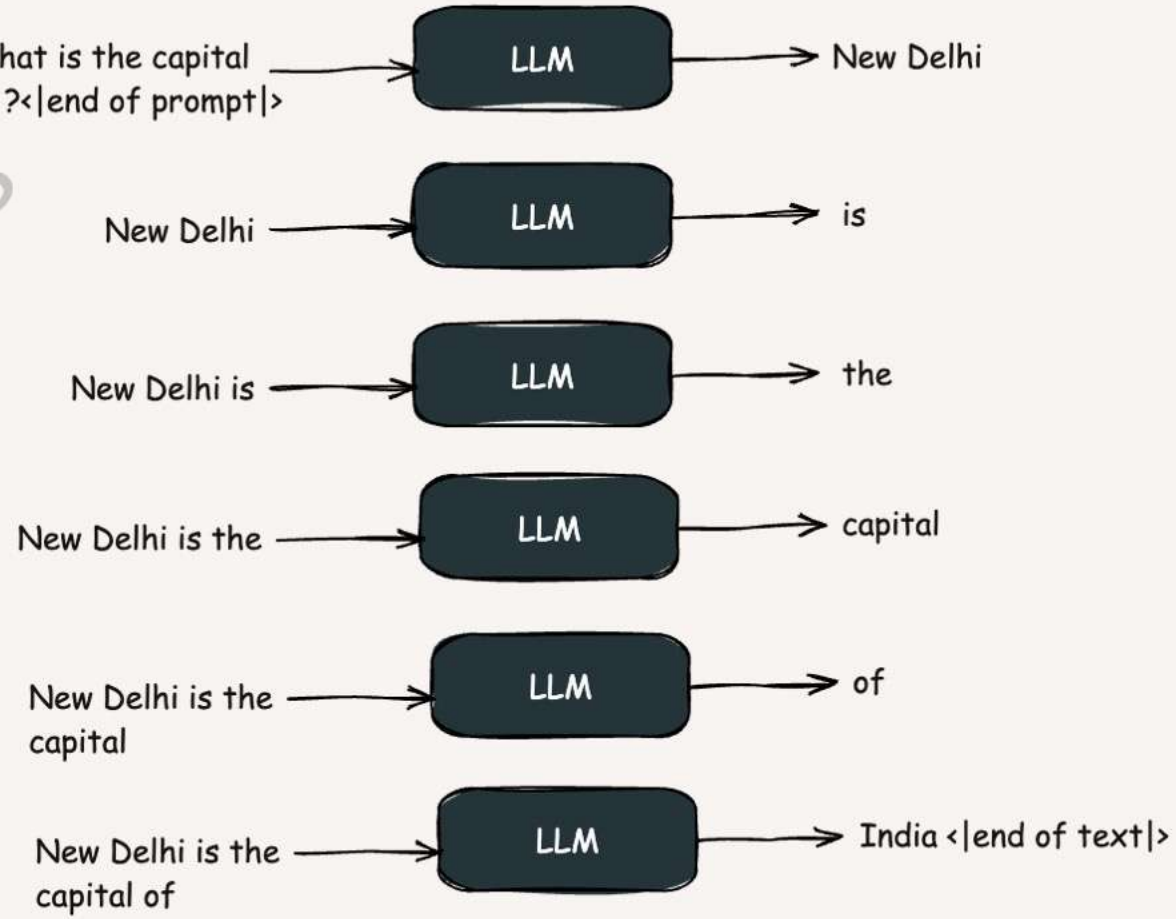
For example, When you say: What is the capital of India ?
The wrapper may send this to the LLM as:
User: What is the capital of India ?<|end of prompt|>

It helps the LLM understand:

- Where the input ends
- That a response is expected
- That this is a question from a user

How does the wrapper know when to stop the loop?
LLMs can also predict special code words like:
<|end of text|> → Means stop generating

Once the LLM outputs <|end of text|>, the wrapper knows it's time to stop the loop.



The concept of Tokens in LLM

eazy bytes

I've said LLMs guess the next word... But that's not quite right. LLMs actually predict the next token — not full words. AI/LLM models can't understand direct text. Everything needs to be converted as Tokens(Numbers).

The process of breaking the original text into tokens is called **tokenization**. For GPT-4, an average token is approximately ¾ the length of a word. So, 100 tokens are approximately 75 words.

So, What's a Token?

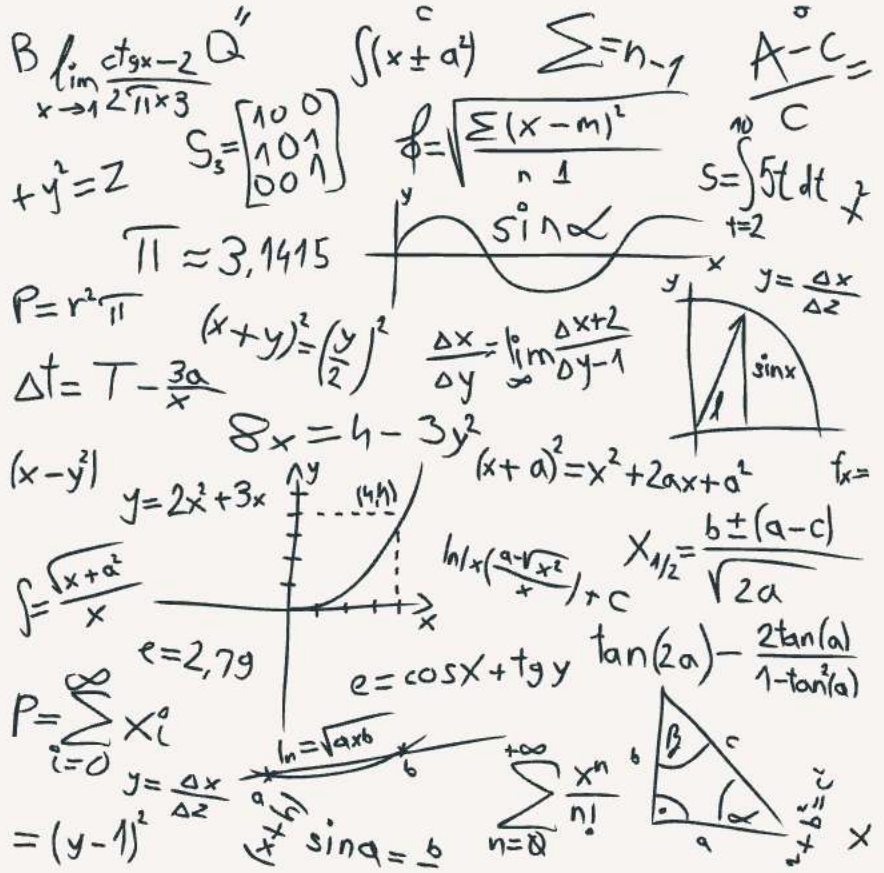
The basic unit of a language model is token. A token can be a character, a word, or a part of a word (like -tion), depending on the model.

It could be:

- A full word → dog
- A word part → ish, play, ing
- A single character → a, b, 1
- A non-English letter → 你, á, æ
- A special keyword → <|end of text|>

📌 Tokens ≠ Words.

They're pieces that models use to build words.



The concept of Tokens in LLM

eazy bytes

Example – "playfulish" . The word playfulish might be split into 3 tokens:

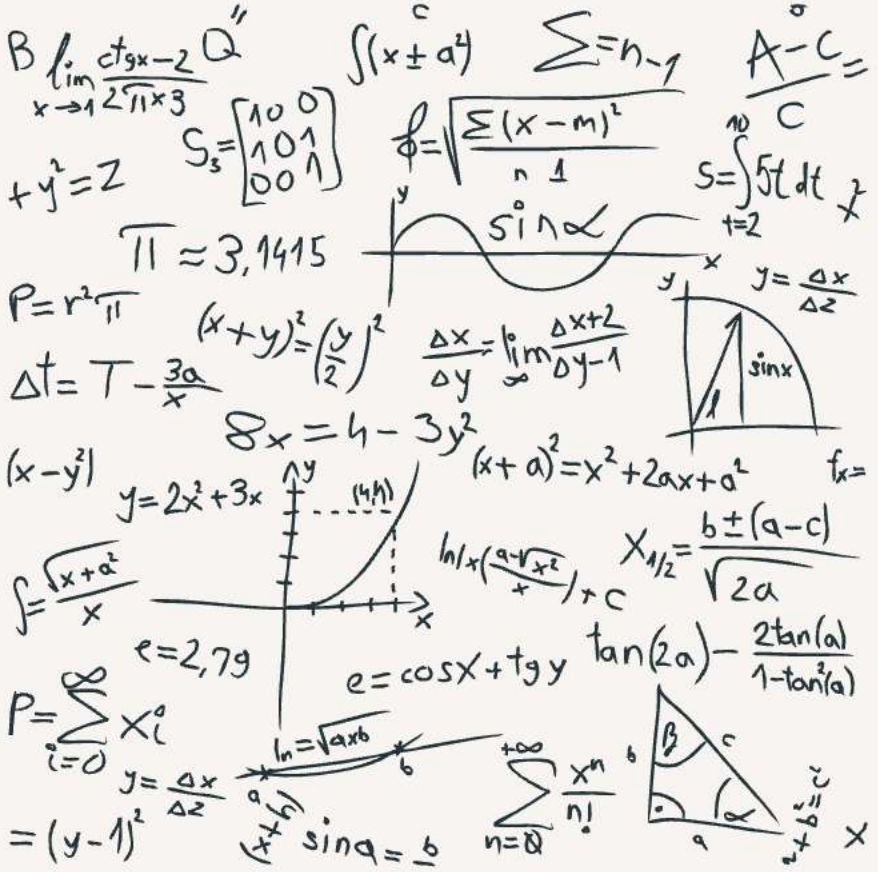
- play
- ful
- ish

Even if the word doesn't exist in a dictionary, LLMs can understand and generate it by combining known tokens.

🤖 This is how LLMs are flexible with new words, typos, or slang.

What exactly happens behind the scenes to understand the text ?

Text → Tokens → Token IDs → Vectors



Model Vocabulary

eazy bytes

Vocabulary Lookup

In the context of large language models (LLMs), a model's vocabulary is the complete list of tokens it can recognize. Each token in this list has a unique ID, so the model can process language as numbers (token IDs), not just text.

Examples of Vocabulary Sizes

Model	Vocabulary Size
GPT-4	~100,000 tokens
BERT	~30,000 tokens
Mixtral	32,000 tokens

The model has a fixed vocabulary – a giant table mapping each token to a unique ID

Token	Token ID
"King"	1342
"Queen"	39
"football"	8121

Model Vocabulary

eazy bytes

What happens if a token not found in vocabulary

Imagine a model with a vocabulary that includes -> "apple", "phone", "table"
But the input word is: "tabletop" -> "tabletop" is not in the vocabulary

How Does the Model Handle This?

The model uses subword tokenization algorithms. These break down unknown words into known smaller pieces (tokens) that are in the vocabulary.

"tabletop" -> Tokenized → ["table", "top"]

If "top" also isn't in the vocabulary, it could go even further → ["tab", "le", "to", "p"]

Each of these will likely exist in the vocabulary.

🧠 The Core Idea:

If a whole word is unknown, the tokenizer splits it into the smallest number of known subword tokens.
So even if the vocabulary doesn't contain "multicloudready", it might tokenize it as -> ["multi", "cloud", "ready"]

Model Vocabulary

eazy bytes

Why language models use tokens instead of just words or characters ?

🧩 Why Not Just Use Words?

Because language is too complex and unpredictable:

- Some words are rare ("unconstitutional").
- Some are misspelled ("gud" instead of "good").
- Some are new (like "iPhone24" or made-up names).
- Some languages (like Chinese) don't have spaces between words!

🔠 Why Not Just Use Characters?

Because characters are too small. To understand "unbelievable", the model would need to process 12 separate steps:
"u", "n", "b", "e", "l", "i", "e", "v", "a", "b", "l", "e"

This makes training slower and understanding harder — like reading one letter at a time instead of chunks.

So LLMs Use Tokens — A Sweet Spot in Between

Embeddings: A way to represent meaning

eazy bytes

What Are Embeddings?

🤖 AI's biggest challenge: How do we teach a model to understand meaning, not just letters? Embeddings help solve this!

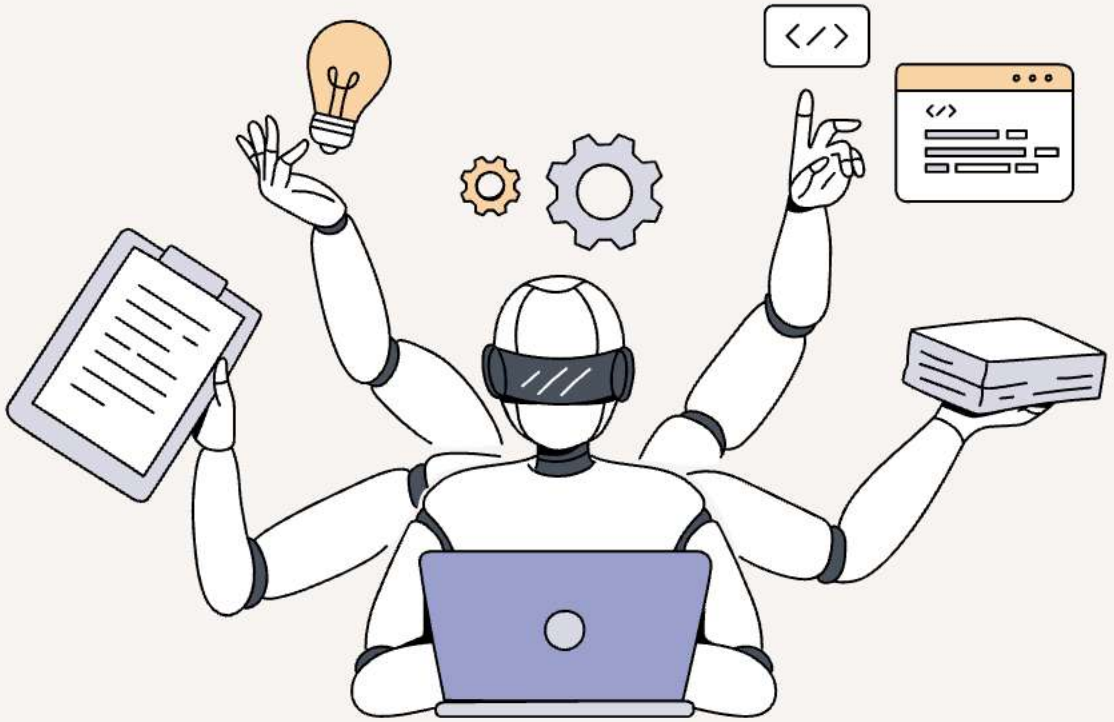
📦 An embedding is:

- A list of numbers (aka a vector)
- Each number tells us how much a word relates to a concept

A vector is just a list of numbers that represents something
In Math Terms: A vector is like a sequence of numbers

$$V = [v_1, v_2, v_3, v_4, v_5]$$

Each v_i is a number representing how much the word matches a concept



Meaning Beyond Spelling

🔤 Traditional AI: Thinks of "King" as just the letters K-I-N-G

💡 But we want the model to "understand" that:

King = 👑 ruler, male, royalty, powerful, leader
So we turn the word into a meaningful vector of numbers.

Embeddings: A way to represent meaning

eazy bytes

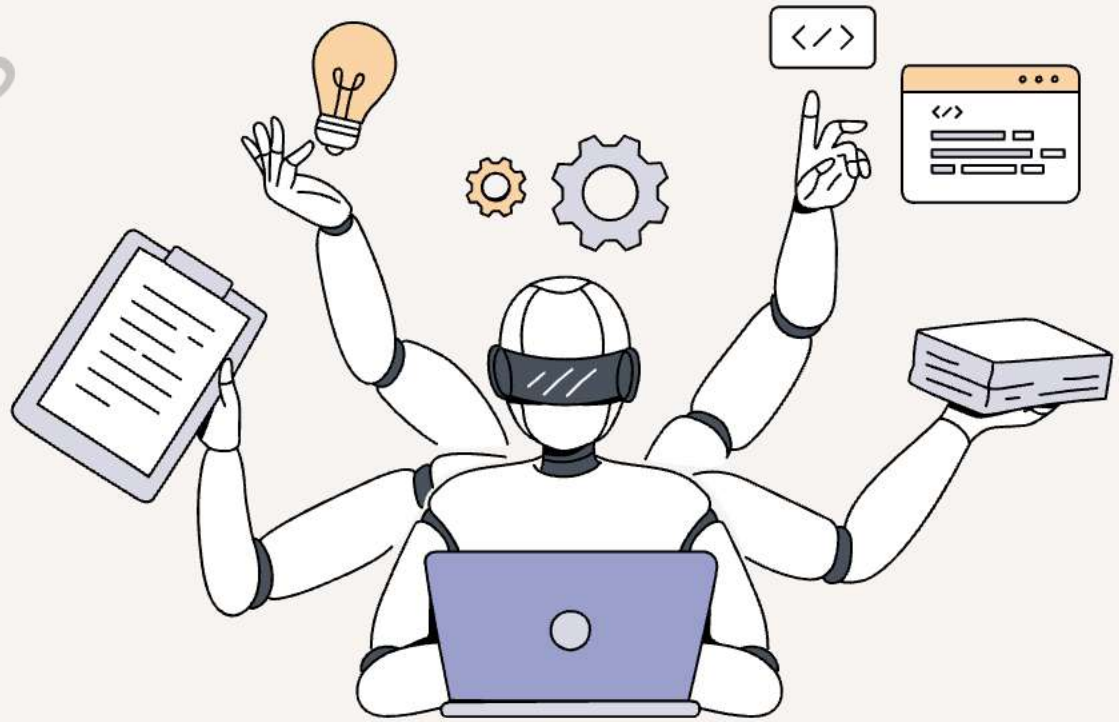
A Simple Imaginary Embedding for "King" (Vector Style)
Let's imagine an embedding vector for the word "King" with 5 topics

🧠 Topics: [Royalty | Female | Leadership | Strength | Fictional]

📦 Embedding Vector for **"King"**:

[9.5 , -8.0 , 7.5 , 6.0 , 2.0]

📝 This means:
9.5 → Strongly related to *Royalty*
-8.0 → Opposite of *Female*
7.5 → Highly related to *Leadership*
6.0 → Quite *Strong*
2.0 → Slightly *Fictional*



📦 Embedding Vector for **"Queen"**:

[9.5 , 8.0 , 7.0 , 5.5 , 2.0]

📝 Interpretation:

Close to "King" in Royalty, Leadership, and Strength but differs significantly in Female value

Embeddings: A way to represent meaning

eazy bytes

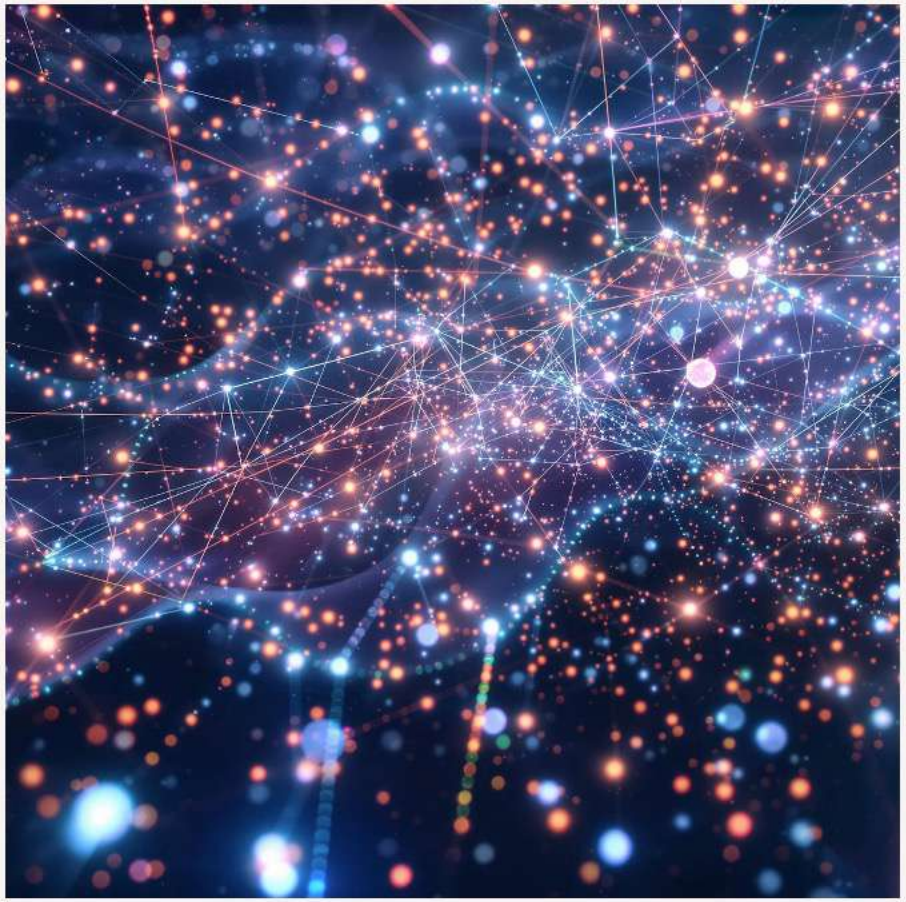
Real Models Use BIG Vectors

Real LLMs don't use 5 topics — they use thousands!

Model	Embedding Size
GPT-3	12,288
LLaMA 3 (small)	4,096
LLaMA 3 (large)	16,384

Each token (like "King") becomes a huge list of numbers, capturing thousands of subtle meanings and patterns.

Since embedding vectors are very long, they exist in a high-dimensional space where similar tokens tend to cluster together — even if we can't visualize it.



How Embedding Vectors calculated ?

eazy bytes

When a token like "king" is converted to a vector:

"king" → [9.5 , -8.0 , 7.5 , 6.0 , ..., 2.0]

This vector might have thousands of numbers. Each of the number represents a characteristic of the token or word

Let's say the model sees thousands of sentences like:

- "The king and queen rule the land."
- "The lion is the king of the jungle."
- "A rich king lived in a big palace."
- "The queen wears a crown."

Through training, the model notices patterns:

- "king" appears in contexts of power, royalty, male gender.
- "queen" appears in similar contexts, but more female.
- "lion" appears in powerful or animal contexts.

So it starts adjusting the vector numbers so that:

- Similar words get similar embeddings
- Differences like gender or species are captured in certain directions

The model learns those concepts implicitly during training, even if it never names the dimensions.

How Embedding Vectors calculated ?

eazy bytes

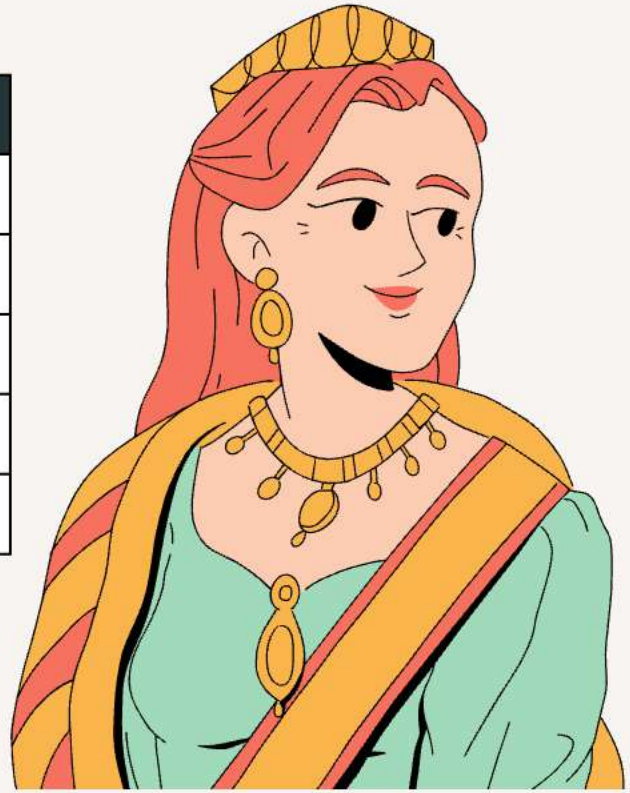
Famous Example: Vector Arithmetic

embedding("king") - embedding("man") + embedding("woman") ≈ embedding("queen")

How is this possible?

Because somewhere in that high-dimensional space, there's a "gender axis" and a "royalty axis", not labeled, but learned automatically. So, dimensions may implicitly capture

Axis (Not named!)	What it captures
Gender	Male ↔ Female
Royalty	Commoner ↔ Royalty
Living being	Object ↔ Animal
Richness	Poor ↔ Rich
Strength	Weak ↔ Powerful



semantic relationships are captured geometrically

Word Embedding Vector Magic – King, Queen example

eazy bytes

embedding("king") - embedding("man") + embedding("woman") ≈ embedding("queen")

The Math Behind the Magic

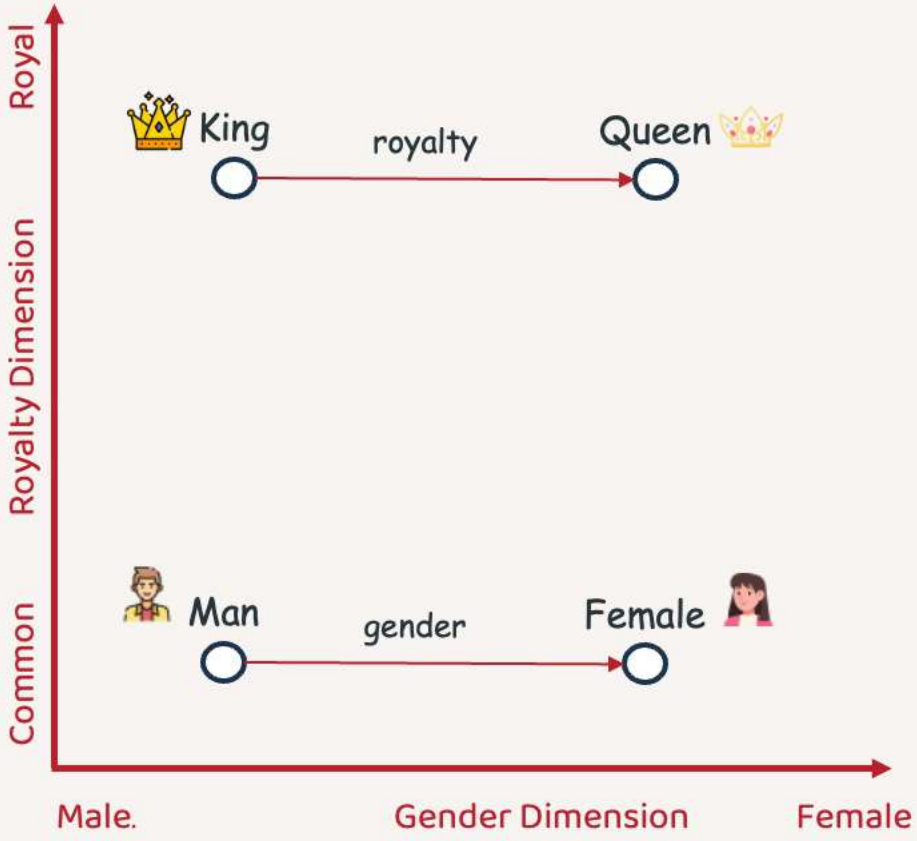
Step 1: King - Man = ?
Royalty Vector (removes "maleness", keeps "royalty")

Step 2: Woman + Royalty Vector = ?
Queen! (adds "royalty" to "femaleness")

Simplified 2D Vectors:

King = [0.1, 0.8] (low gender, high royalty)
Man = [0.1, 0.2] (low gender, low royalty)
Woman = [0.9, 0.2] (high gender, low royalty)
Queen = [0.9, 0.8] (high gender, high royalty)

King - Man + Woman:
[0.1, 0.8] - [0.1, 0.2] + [0.9, 0.2] = [0.9, 0.8] ≈ Queen ✓



Words that are conceptually similar are close in vector space, and conceptual relationships become mathematical operations. This is why LLMs can understand analogies, comparisons, and complex relationships - they're literally doing geometry with meaning!

Word Embedding Vector Magic – France, Paris example

eazy bytes

embedding("capital") + embedding("France") ≈ embedding("Paris")

Step 1: During training, the AI processes millions of sentences like:

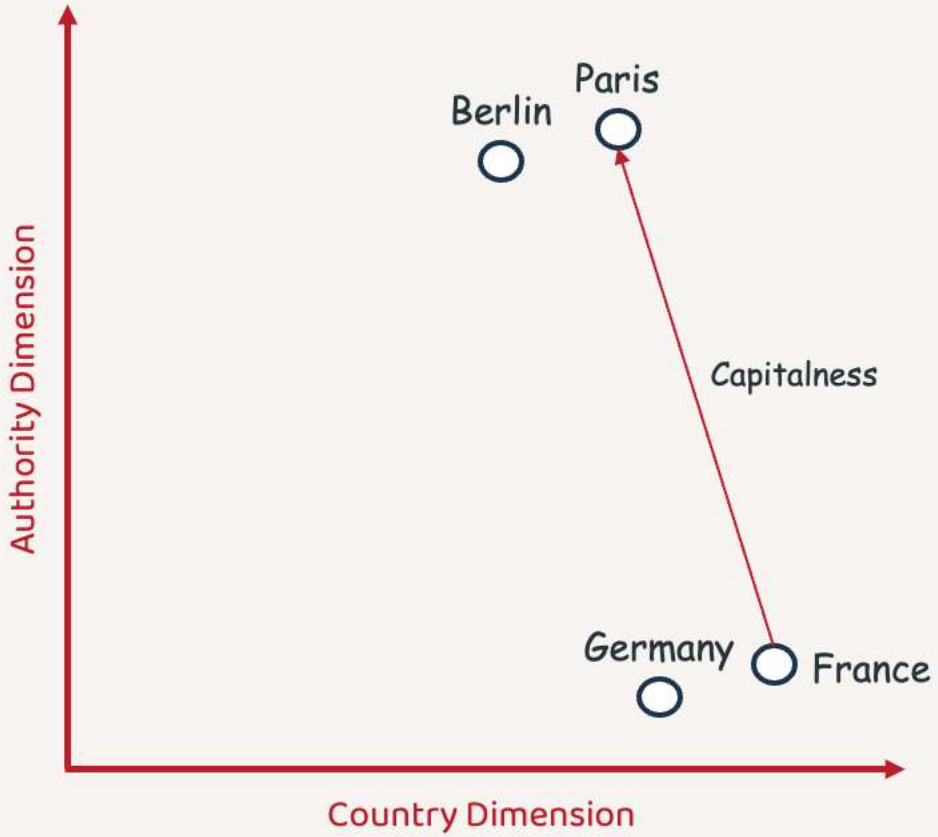
"Paris is the capital of France"
"France's capital city is Paris"
"The capital of France is Paris"

From this, it learns to create a Geography Vector that captures the "capital-of" relationship.

Step 2: Each word gets converted to a vector in high-dimensional space.
In our simplified 2D example:

Capital = [0.2, 0.9] (low country-ness, high authority)
France = [0.9, 0.3] (high country-ness, low authority)
Paris = [0.8, 0.8] (high country-ness, high authority)

Step 3: When asked "What is the capital of France?", the AI:
Recognizes the pattern as Capital + France
Capital + France = [0.2, 0.9] + [0.9, 0.3] = [1.1, 1.2] ≈ Paris
Finds the closest match in vector space = Paris (92% confidence)



How Embedding Vectors calculated ?

eazy bytes

What value does "king" vector have at the start of training?

When training just begins, the vector for "king" (or any token) is: A random list of numbers.

"king" → [0.04, -0.92, 0.15, ..., 0.66] // Randomly initialized or all zeros in some models

These initial vectors have no meaning. During training the sentences like will be feed:

- The king ruled the kingdom.
- The queen wore a crown.
- The king sat on the throne.

The model tries to:

- Predict the next word
- Fill in missing words
- Match the right outputs for inputs

If it's wrong, the model calculates loss, and then:

- Updates the vector for "king"
- Updates the vectors of surrounding words (context)
- After millions of training examples, "king"'s vector becomes meaningful, and settles somewhere near words like "queen", "prince", "royal", etc.

Why Are Embeddings Useful?

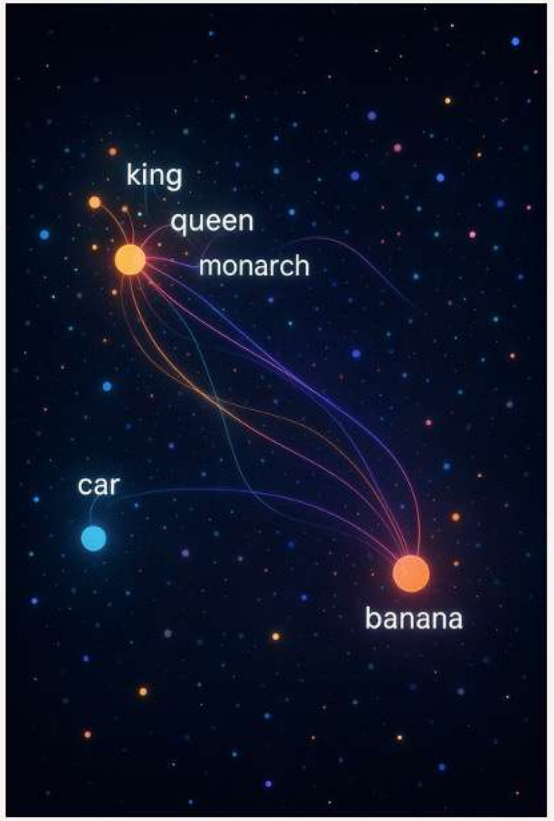
eazy bytes

Embeddings are not just numbers — They help us compare and extract meaning between words.

We can compare how similar two words are by using their embedding vectors.

📌 Example:

King and Queen → Very close in meaning → Vectors are close
King and Banana → Distant in meaning → Vectors are far apart



How Do LLM Measure "Closeness"?

They use a simple math trick called the dot product.

🔶 Multiply each number in one vector by the matching number in the other vector
🔶 Add all the results
💡 Result tells us:

- Positive → Meanings are related
- Zero → Unrelated
- Negative → Opposite meanings (e.g., large vs. small)

Why Are Embeddings Useful?

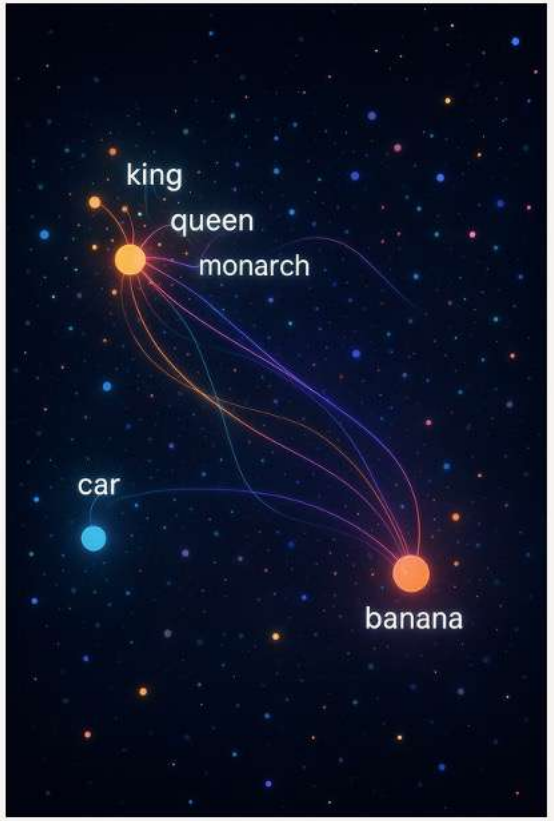
eazy bytes

Let's say we compare these two embeddings:

Vector for "King": [9, -5, 6]

Vector for "Queen": [8, -4, 6]

🧱 Dot product:
(9×8) + (-5×-4) + (6×6) = 72 + 20 + 36 = 128 ✅ → Strong similarity



The full vector of "King" has thousands of dimensions — but maybe you only care about King traits like:

["Royalty", "Gender", "Human"]

This means:

You want to squash the vector into a smaller one that only keeps these values.

What Is This Squashing Called? It's called a projection.

📉 Projection =
Flattening a huge space (e.g. 12,288D) into a smaller one (e.g. 100D) that focuses on specific topics.

Why Are Embeddings Useful?

eazy bytes

Here's what really happens when we ask a question to LLM:

- Your input (question) is first tokenized and converted into high-dimensional embedding vectors.
- These embeddings then pass through multiple layers of the neural network (attention layers, linear projections, etc.).
- At every layer, the model learns which parts of the vector space matter for the task — it performs dynamic, contextual projections.
- By the end, the model produces an output based on those "filtered" or "focused" representations.

Think of it like this, let's say you ask:

"What sound does a dog make?"

The model doesn't need to access all 12,000+ dimensions of every token's meaning equally.

Instead, it focuses on dimensions related to:

- Animals
- Sound
- Pets

...and disregards things like "currency", "architecture", or "sports".
This attention + projection happens internally — like smart compression.

Static Embeddings

Each token ID now gets converted into a vector of fixed size, like a mathematical meaning of the token.
These vectors are called static embeddings because:

- They are pre-trained
- They are fixed (the vector for "King" is always the same)
- They do not depend on surrounding words

Token	Embedding (Vector) Example (simplified)
"King"	[9.5 , -8.0 , 7.5 , 6.0 , ..., 2.0]
"Queen"	[9.5 , 8.0 , 7.0 , 6.0 , ..., 2.0]
"football"	[7.5 , 2.6 , 1.2 , 1.4 , ..., 4.0]

📌 These embeddings are stored in a lookup table. Each token ID indexes its corresponding vector.

The word "bank" has one static embedding regardless of context:

- "He went to the bank to deposit money"
- "She sat by the bank of the river"

Static embedding for "bank" is the same, even though meanings differ. That's why later layers (like attention) are needed to understand context.

Positional Embedding

eazy bytes

Positional Embedding → Add information about position of each token

Why Do We Need Positional Embeddings? -> Unlike humans, transformers don't naturally understand word order.

The sentence "The cat sat on the mat" and "The mat sat on the cat" have different meanings — but the model wouldn't know that without positional info. Transformers are order-agnostic — they look at all tokens at once (self-attention) and don't know the sequence unless we tell them.

So we use positional embeddings to fix that.

Final Input to Transformer:

Add token embedding + positional embedding element-wise:

FinalVector[i] = TokenEmbedding[i] + PositionalEmbedding[i]

So now, each token is aware of both:

- Its meaning (from static embedding)
- Its position (from positional embedding)

Attention Layer in GenAI

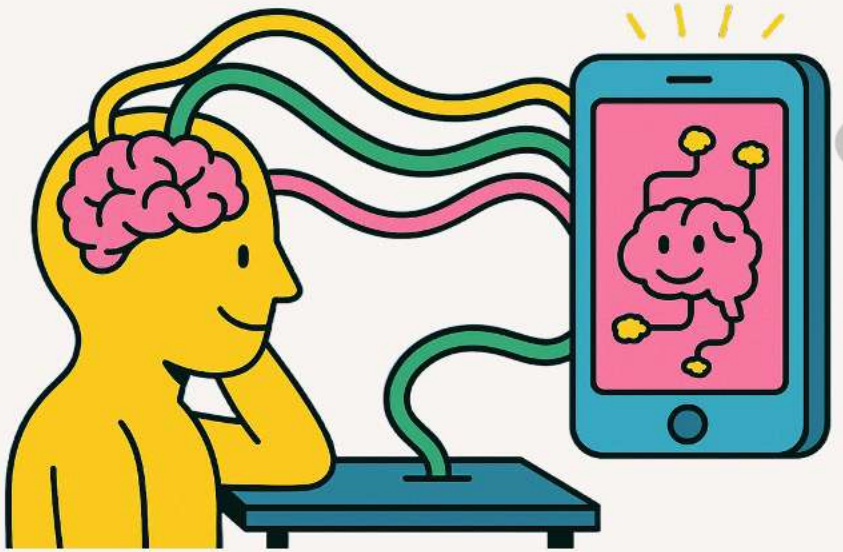
eazy bytes

What is Attention in an LLM?

Think of attention like a smart highlighter 🖍️ the model uses to decide:

"Which words in the sentence should I focus on more when trying to understand or predict a particular word?"
It's like the model asking: "To understand the word I'm looking at now, which other words matter most?"

It's especially useful when a word has multiple meanings — like "bank."



Example 1:
"He went to the bank to deposit a check."

Now the model looks at the word "bank". It needs to decide — is this a money bank or a river bank? So it checks the context (other words in the sentence):

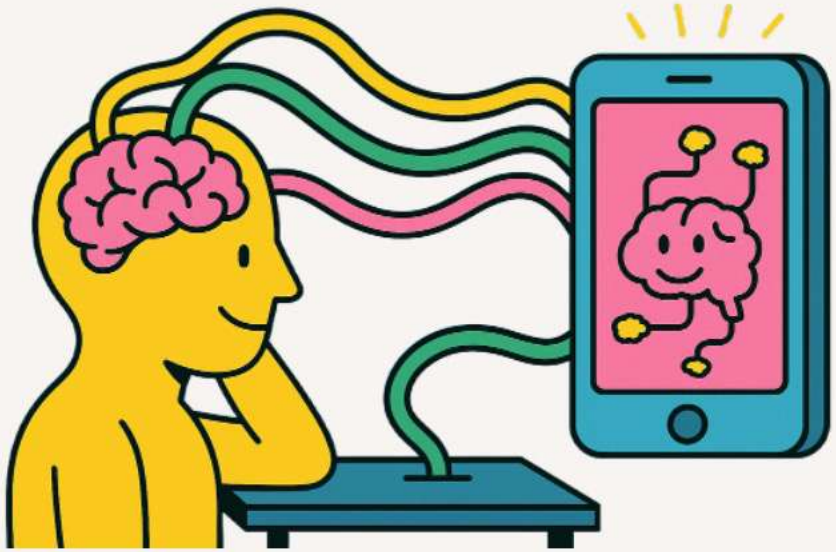
"deposit" → strong signal for a financial bank 💰
"check" → confirms it's about money 🧾
"went to the..." → also fits the banking scenario

So, the model gives attention scores like:

deposit → 50%, check → 30%, went → 15%, he → 5%

Result: The model concludes "bank" = financial institution

Example 2:
"She sat by the bank of the river and watched the sunset." Now it sees "bank" again — but in a different sentence.



Let's look at the new context:

"river" → strong clue this is about nature 🌊
"sat by" → suggests a location, not a transaction 🪑
"sunset" → fits a peaceful outdoor scene 🌇

Attention scores might look like:

river → 60% , sat → 25%, sunset → 10% , she → 5%

Result: The model understands "bank" = river edge

LLMs use contextual attention to:

- Dynamically decide which words to focus on
- Change the meaning of a word based on those context clues

This is only possible because of attention layers — static embeddings alone cannot do this!

Chat Memory using Spring AI

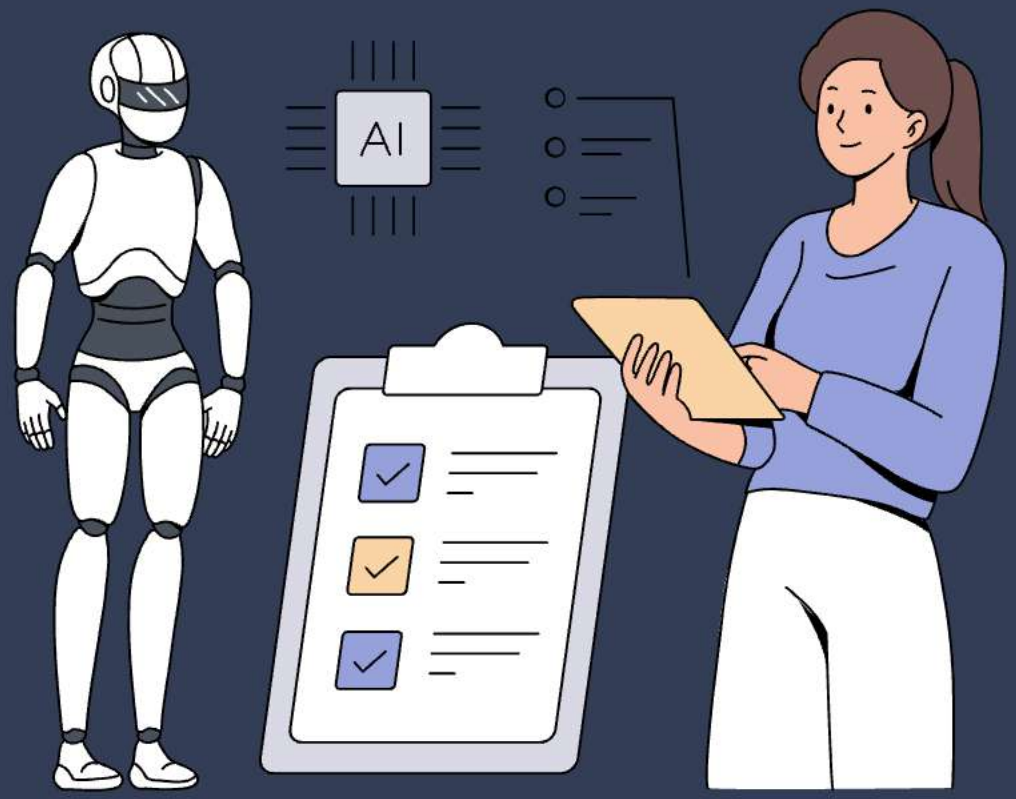
eazy bytes

🤖 LLMs Are Stateless – But Spring AI Adds Memory!

- LLMs don't remember past chats. Each interaction is like a fresh start — no memory!
- This can be a problem if you want to maintain conversation context across multiple interactions.

🧠 Enter Spring AI's Chat Memory

- Spring AI gives us the **ChatMemory** abstraction to store and manage conversation history.
- Messages are saved using a ChatMemoryRepository — think of it as the storage engine behind the scenes.



🔁 Flexible Memory Strategies

We can choose what to remember:

- Keep only the last N messages
- Store messages for a specific time window
- Retain messages until a token limit is reached

💡 With ChatMemory, LLMs can behave more like a human — remembering the past to improve the response!

Chat Memory using Spring AI

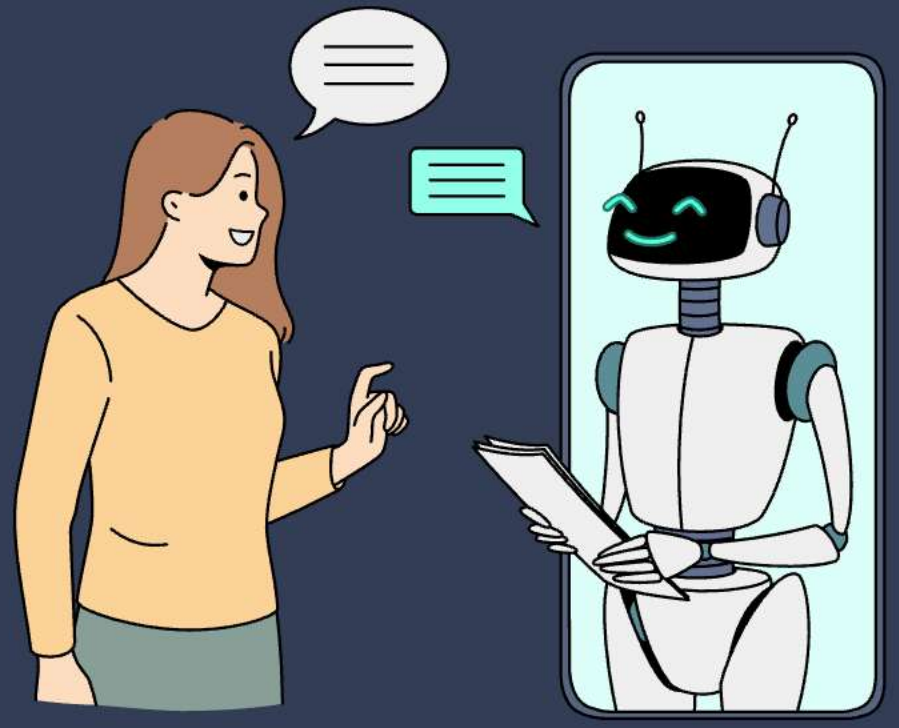
eazy bytes

🧠 How Chat Memory Works

ChatMemory: Defines what to store (e.g., last N messages)
Sample implementation class MessageWindowChatMemory

ChatMemoryRepository: Defines contract for storing and retrieving chat messages.

- Using InMemoryChatMemoryRepository we can store the chat memory inside In-Memory (default)
- Using JdbcChatMemoryRepository we can store the chat memory in a DB like H2, MySLQ, Postgres etc.



Chat Memory using Spring AI

eazy bytes

Making ChatClient "Remember" with Advisors

When using Spring AI's ChatClient, you can add memory to your LLM conversations with the help of Advisors.
These advisors manage how memory is stored and reused across multiple interactions.

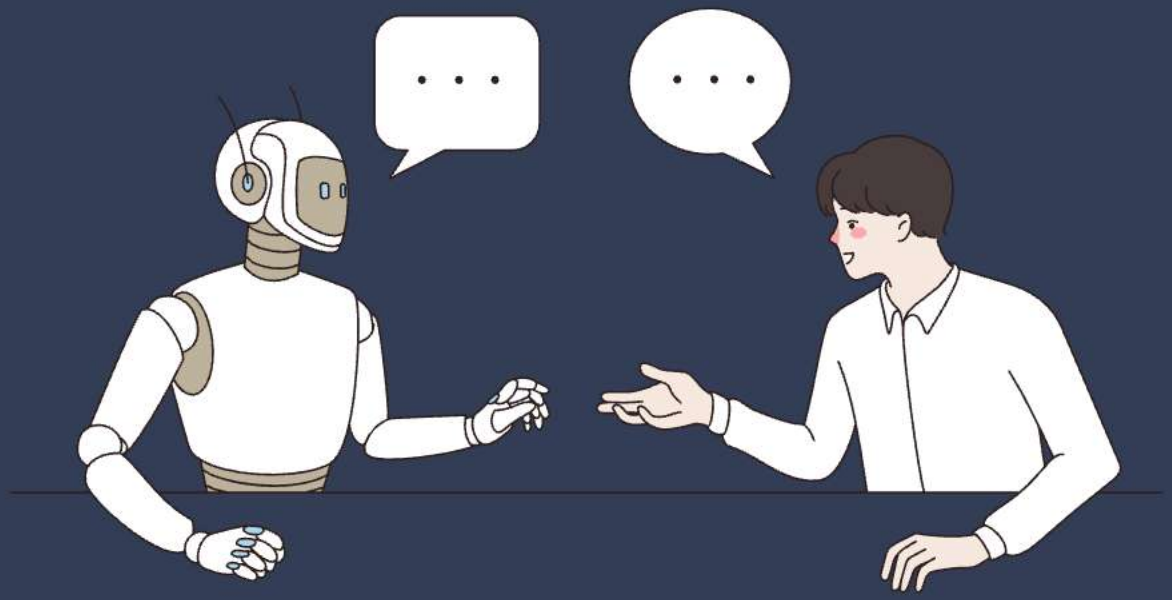
⚙️ Built-in Advisors in Spring AI

1️⃣ MessageChatMemoryAdvisor

💬 Stores chat as a list of structured messages
📃 Injects past messages directly into the prompt
✅ Best when you want the LLM to "see" the full chat history like a real chat log

2️⃣ PromptChatMemoryAdvisor

🪵 Converts memory into plain text format
🧠 Appends it to the system prompt (like a summary)
✅ Good for simple LLMs or when token budget is limited

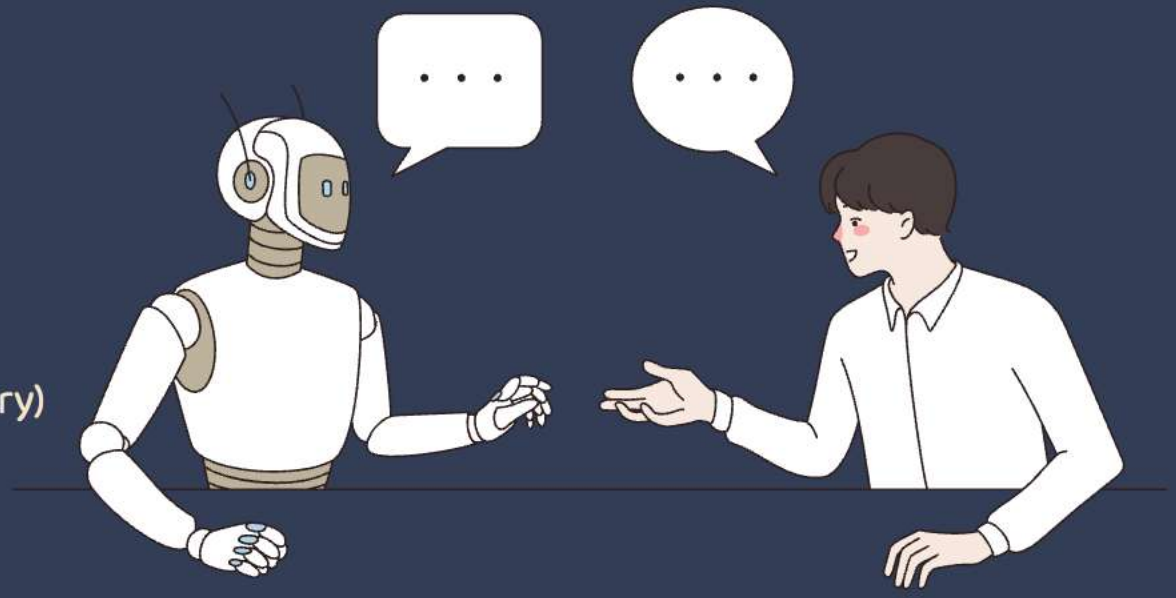


Chat Memory using Spring AI

eazy bytes

Making ChatClient "Remember" with Advisors

3️⃣ VectorStoreChatMemoryAdvisor



📂 Stores memory in a vector database (e.g., Qdrant, Pinecone)
🔍 Retrieves the most relevant past messages using embeddings
✅ Ideal for long or knowledge-based conversations (semantic memory)

When to Use What?

Advisor Type	Format	Use Case
MessageChatMemoryAdvisor	Structured Msg	Real-time chat memory
PromptChatMemoryAdvisor	Plain Text	Token-optimized conversations
VectorStoreAdvisor	Semantic Match	Long-term or knowledge-based chats

Chat Memory using Spring AI

eazy bytes

💾 Default Memory Setup

Spring AI auto-configures a ChatMemory bean that you can use directly in your application. By default, it uses an in-memory repository to store messages (InMemoryChatMemoryRepository) and a MessageWindowChatMemory implementation to manage the conversation history. The same can be injected into the ChatClient as shown below,

```
@Bean
ChatClient chatClient(ChatClient.Builder builder, ChatMemory chatMemory) {
    return builder
        .defaultAdvisors(MessageChatMemoryAdvisor.builder(chatMemory).build()).build();
}
```



```
@GetMapping("/chat-memory")
public ResponseEntity<String> chatMemory(@RequestHeader("username") String username,
            @RequestParam("message") String message) {
   String conversationId = username;
   return ResponseEntity.ok(chatClient.prompt().user(message)
        .advisors(advisorSpec -> advisorSpec.param(CONVERSATION_ID,conversationId))
        .call().content());
}
```

Chat Memory using Spring AI

eazy bytes

🔌 Custom Persistent Memory (JDBC)

```
<!-- Add JDBC support -->
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-chat-memory-repository-jdbc</artifactId>
</dependency>

<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```


JdbcChatMemoryRepository is a built-in implementation that uses JDBC to store messages in a relational database. It supports multiple databases out-of-the-box and is suitable for applications that require persistent storage of chat memory.

Spring AI also supports storing chat memory using Neo4j (**Neo4jChatMemoryRepository**) and Cassandra (**CassandraChatMemoryRepository**)

Chat Memory using Spring AI

eazy bytes

Custom Setup (H2 DB + Memory Enabled ChatClient)

```
@Bean
ChatMemory jdbcChatMemory(JdbcChatMemoryRepository chatMemoryRepository) {
    return MessageWindowChatMemory.builder()
            .chatMemoryRepository(chatMemoryRepository)
            .maxMessages(10)
            .build();
}

@Bean
ChatClient chatClient(ChatClient.Builder builder, ChatMemory chatMemory) {
    return builder
        .defaultAdvisors(MessageChatMemoryAdvisor.builder(chatMemory).build()).build();
}

@GetMapping("/chat-memory")
public ResponseEntity<String> chatMemory(@RequestHeader("username") String username,
            @RequestParam("message") String message) {
   String conversationId = username;
   return ResponseEntity.ok(chatClient.prompt().user(message)
        .advisors(advisorSpec -> advisorSpec.param(CONVERSATION_ID,conversationId))
        .call().content());
}
```

Chat Memory using Spring AI

eazy bytes

Backing DB: H2 Configuration

```
spring.ai.chat.memory.repository.jdbc.initialize-schema=always
spring.ai.chat.memory.repository.jdbc.schema=classpath:/schema/schema-h2db.sql
spring.datasource.url=jdbc:h2:file:~/chatmemory;AUTO_SERVER=true
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=madan
spring.datasource.password=12345
```

Consider Context Window when using Chat Memory

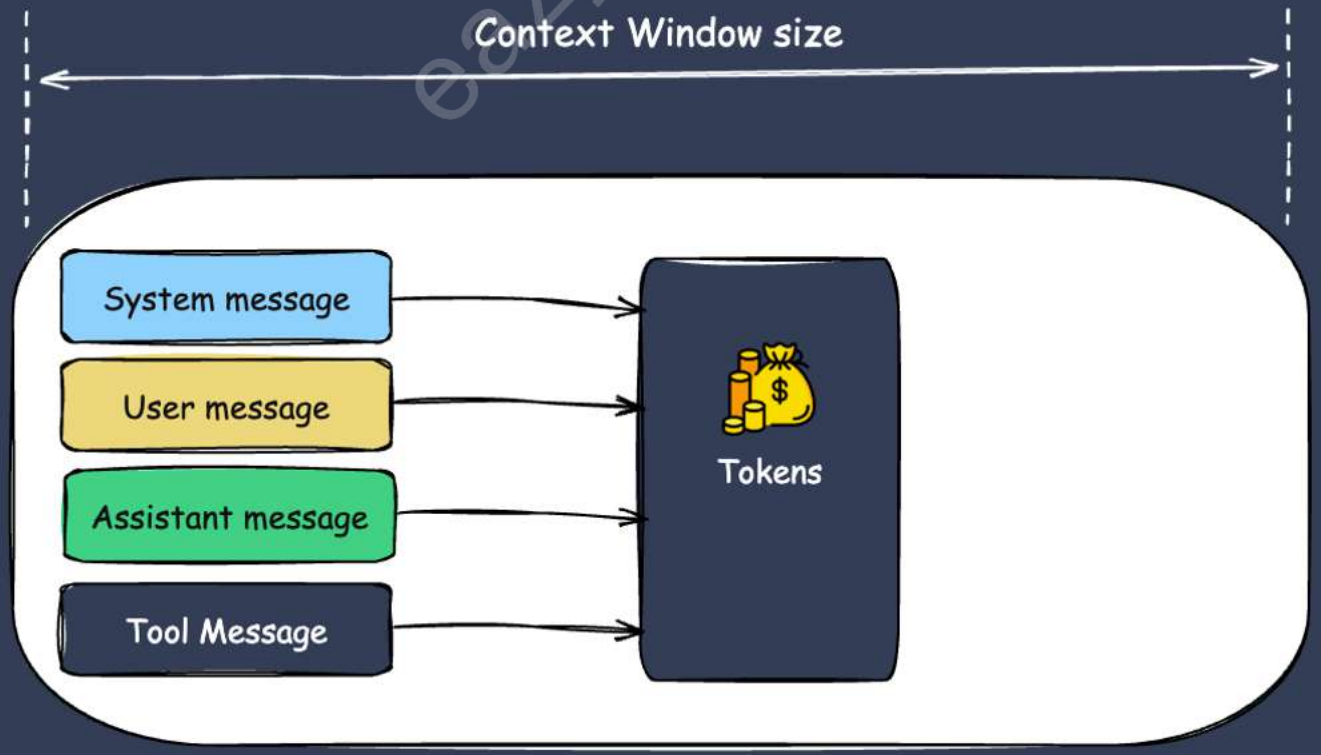
eazy bytes

The context window is like the memory span of a Large Language Model (LLM). It tells the model how much text (tokens) it can "see" at one time while generating responses.

Real-World Analogy

Imagine a whiteboard that fits only so much text. Once full, you have to erase the older parts to make space.

Similarly, if the total token count exceeds the context window size, older messages get dropped. The LLM "forgets" what gets dropped.



Consider Context Window when using Chat Memory

eazy bytes

How to Be Careful with Context Window when using Chat Memory

- Explore the Model's limit and choose a model based on how long your conversations are expected to be.
- For large chat histories, prefer VectorStoreChatMemoryAdvisor to select only relevant past messages instead of the entire history.
- Limit the Number of Stored Messages. Spring AI by default keeps last 20 messages
- Log or Monitor Token Usage

Simple Prompting — Just the LLM

eazy bytes

Limitations of Relying on LLM Alone

With just LLM, you give a prompt → The LLM generates a response based on its pre-trained knowledge.

🧩 But here's the problem...

❌ LLM doesn't know everything! It can't fetch real-time or private company data

❌ It offers little or no personalization

💬 "It's like asking someone to answer questions without letting them Google or look at their notes."

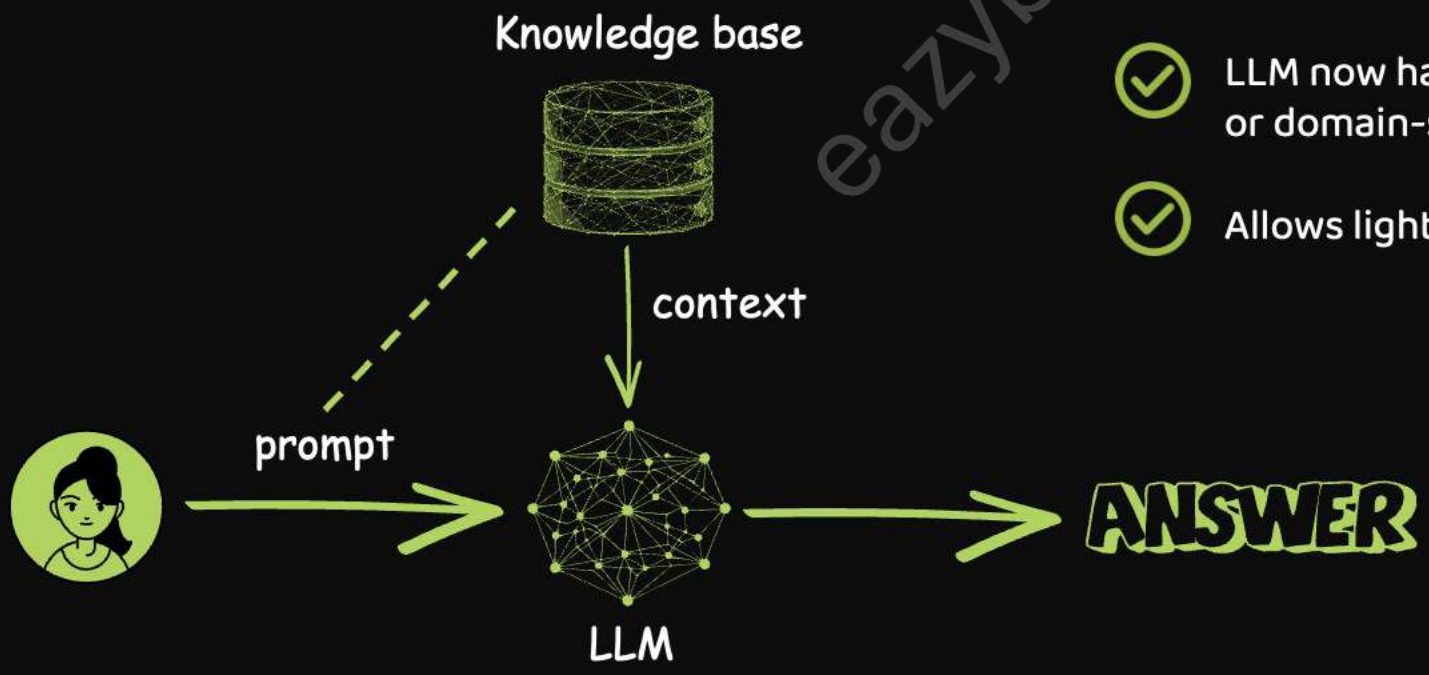
Retrieval-Augmented Generation — Smart + Informed

eazy bytes

RAG: Giving LLMs Superpowers with External Knowledge

With RAG implemented,

- You give a prompt
- RAG searches a knowledge base for relevant documents or information
- These fetched data/documents are added as context to the LLM
- Then the LLM generates a more accurate and grounded answer



- LLM now has access to the right info — even private, dynamic, or domain-specific!
- Allows light personalization based on user or context

Why RAG ?

eazy bytes

The Challenge with Prompt Stuffing

When answering questions using LLMs, we often try to include helpful context directly in the prompt. This is called "prompt stuffing"—adding all relevant information to help the model answer accurately.

It works well when the documents are small. But as the number or size of documents increases:

❌ The prompt size becomes too large which will leads to more tokens consumption
❌ Only a tiny portion of the context may be relevant to the question

Enter RAG (Retrieval Augmented Generation)

RAG solves the inefficiency of prompt stuffing. It breaks large documents into smaller chunks. During each user query:

✅ Only the most relevant chunks (retrieved using similarity search) are added to the prompt.
✅ This keeps the prompt compact and focused.
✅ Improves accuracy and reduces token waste.

What is RAG ?

eazy bytes

RAG = Retrieval + Augmentation + Generation

A smart way to give language models access to external knowledge so they can give better, more accurate answers.

User: "Tell me about a product XXXXXX"

R Retriever: Searches company docs, PDFs, or a vector database

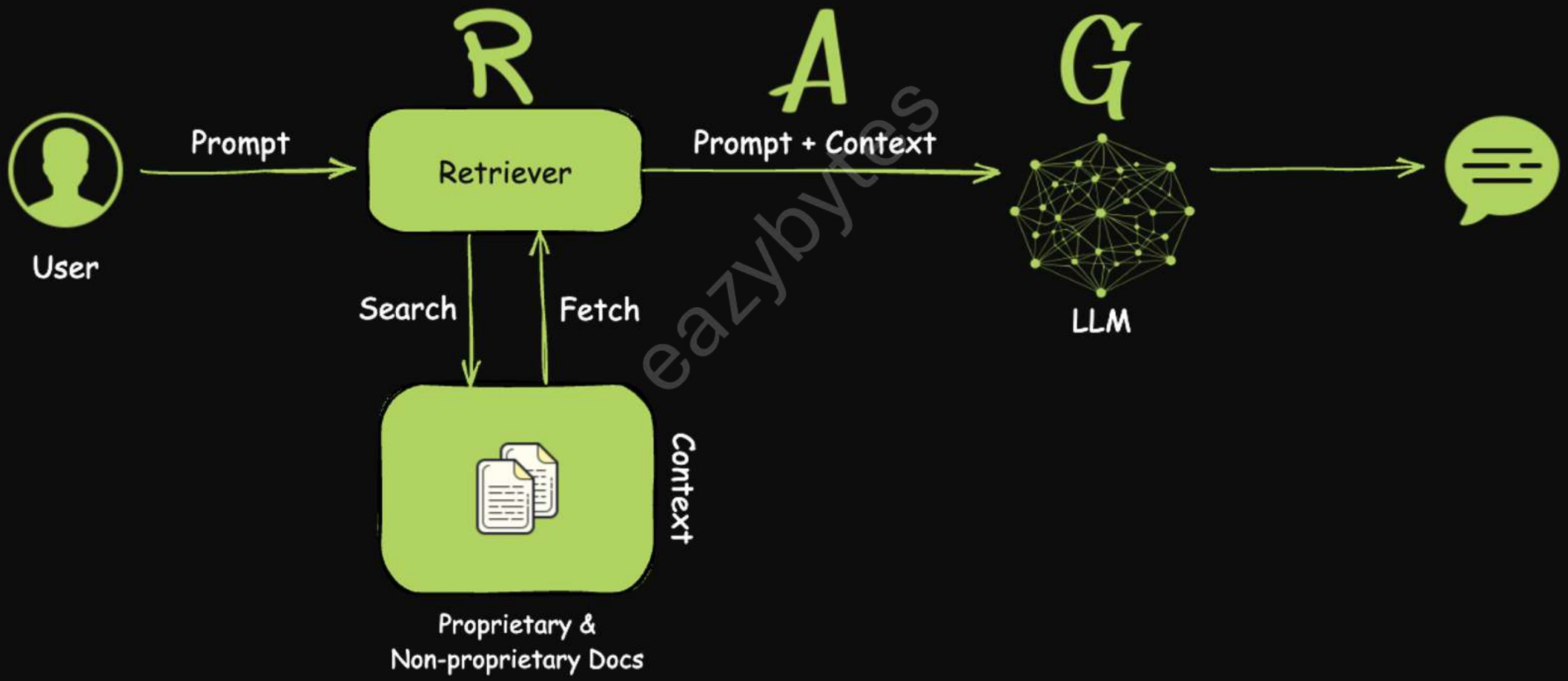
A Augmentor: Picks the most relevant chunks of text

G Generator (LLM): Writes an answer using that retrieved knowledge

✅ More accurate, up-to-date, and personalized response!

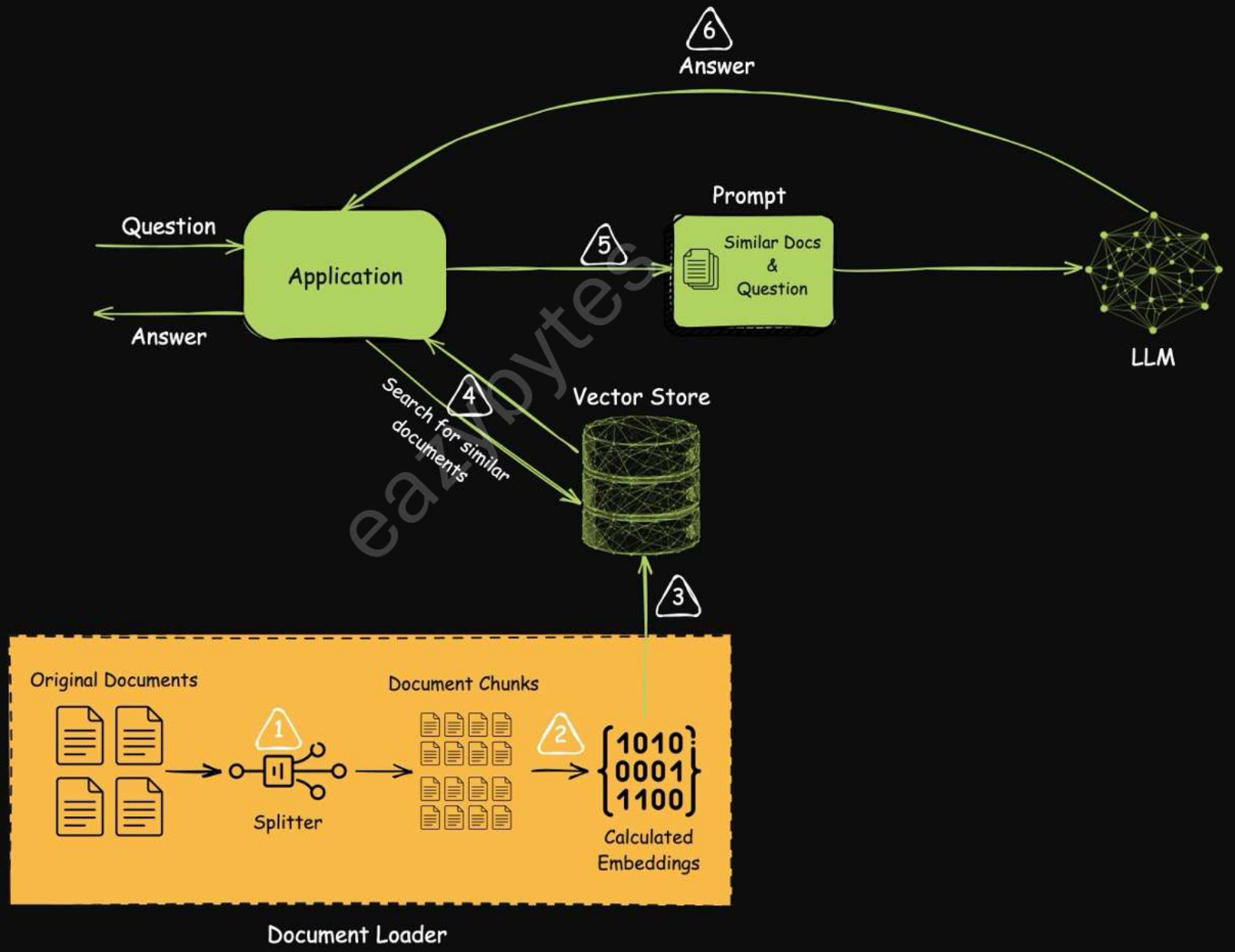
RAG flow

eazy bytes



RAG flow

eazy bytes



Setting up a vector store

eazy bytes

What is a Vector Store (or Vector Database)?

A Vector Store is a special type of database designed to store and search data in the form of vectors (high-dimensional numerical representations).

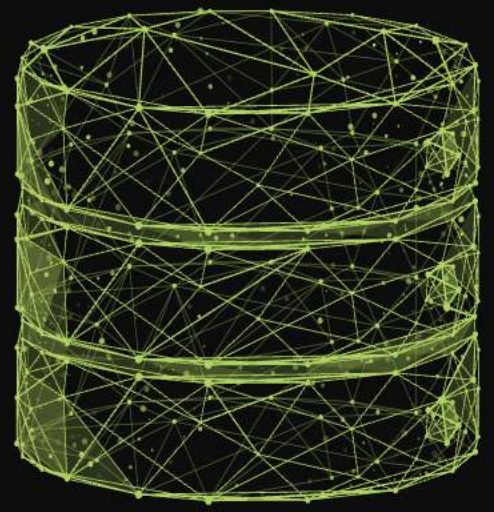
These vectors represent semantic meaning of text, images, or audio, often generated using AI models (like embeddings from OpenAI).

Why Use Vectors ?

Traditional keyword search is limited (e.g., only finds exact or similar words).

Vector search understands context and meaning, enabling semantic search.

Example: Searching "How to fix a laptop screen" will retrieve content related to "repairing a broken display" even if exact words don't match.



Popular Vector Store Tools

Azure AI Search, Cassandra, Elasticsearch, MongoDB, Neo4j, Pinecone , PostgreSQL with pgvector extension, Qdrant, Redis with RediSearch module

Setting up a vector store

eazy bytes

How Vector Stores Enable RAG

1. Indexing the Knowledge

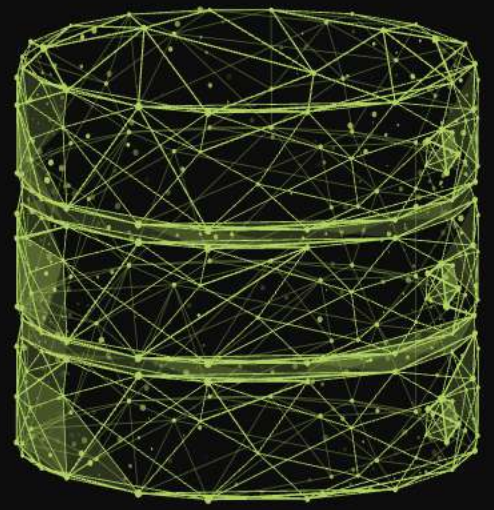
- Convert documents into vector embeddings
- Store them in a vector store

2. Semantic Retrieval

- When a user asks a question:
- Convert it into a query vector
- Use the vector store to retrieve most similar document chunks

3. Augment the Prompt

- Inject the retrieved chunks into the LLM prompt
- The model uses this relevant context to generate accurate answers



Vector Store vs Traditional DB

eazy bytes

Feature	Traditional DB	Vector Store
Data Format	Structured (tables, rows)	Unstructured (text/images → vectors)
Search Type	Keyword or SQL-based	Semantic (meaning-based)
Use Case	CRUD operations, reports	AI, NLP, recommendation, search
Similarity Matching	No	Yes (e.g., cosine similarity)

Setting Up Qdrant Vector Store in Spring AI

eazy bytes

1 Define Qdrant Service in compose.yml

```
services:
  qdrant:
    image: 'qdrant/qdrant:latest'
    ports:
      - '6333:6333' # REST API
      - '6334:6334' # gRPC API
```

- ✓ This will creates a Qdrant container locally via Docker during the startup of the app
- ✓ Port 6334 is used by Spring AI for communication (gRPC)

2 Add Required Dependencies in pom.xml

```
<!-- Enables Docker Compose support -->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-docker-compose</artifactId>
 <scope>runtime</scope>
</dependency>

<!-- Core Spring AI components -->
<dependency>
 <groupId>org.springframework.ai</groupId>
 <artifactId>spring-ai-rag</artifactId>
</dependency>
```

```
<!-- Advisor support for vector store memory -->
<dependency>
 <groupId>org.springframework.ai</groupId>
 <artifactId>spring-ai-advisors-vector-store</artifactId>
</dependency>

<!-- Qdrant vector store starter -->
<dependency>
 <groupId>org.springframework.ai</groupId>
 <artifactId>spring-ai-starter-vector-store-qdrant</artifactId>
</dependency>
```

Setting Up Qdrant Vector Store in Spring AI

eazy bytes

3 Configure Application Properties

```
spring.docker.compose.stop.command=down

spring.ai.vectorstore.qdrant.initialize-schema=true
spring.ai.vectorstore.qdrant.host=localhost
spring.ai.vectorstore.qdrant.port=6334
spring.ai.vectorstore.qdrant.collection-name=eazybytes
```


- ✓ Automatically initializes Qdrant schema
- ✓ Uses eazybytes collection for storing vectorized documents
- ✓ Port 6334 is the gRPC entry point used by Spring AI

RAG Demo with Random Data

eazy bytes

1 Load Data into the Vector Store

This ensures the vector store is pre-filled for semantic search during chat.

```
@Component
public class RandomDataLoader {

    @PostConstruct
    public void loadSentencesIntoVectorStore() {
        List<String> sentences = List.of(
            "Java is used for building scalable enterprise applications.",
            "Docker packages applications into lightweight containers.",
            ...
        );
        List<Document> documents = sentences.stream()
            .map(Document::new)
            .toList();
        vectorStore.add(documents);
    }
}
```

RAG Demo with Random Data

eazy bytes

2 Accept User Input via API

Create a REST endpoint to simulate a chat interface and captures the user's query and identity for personalized context.

```
@GetMapping("/random/chat")
public ResponseEntity<String> randomChat(
    @RequestHeader("username") String username,
    @RequestParam("message") String message)
```


3 Perform Similarity Search from Qdrant

Retrieves top 3 semantically similar documents based on the user query.

```
SearchRequest searchRequest = SearchRequest.builder()
    .query(message)
    .topK(3)
    .similarityThreshold(0.5)
    .build();

List<Document> similarDocs = vectorStore.similaritySearch(searchRequest);
```

RAG Demo with Random Data

eazy bytes

4 Format the Retrieved Context

Prepare the retrieved documents as input for the prompt.

```
String similarContext = similarDocs.stream()
    .map(Document::getText)
    .collect(Collectors.joining(System.lineSeparator()));
```

5 Generate Response Using ChatClient

Combines retrieved content with the user query to generate a relevant and grounded response.

```
String answer = chatClient.prompt()
    .system(system -> system.text(promptTemplate).param("documents", similarContext))
    .advisors(a -> a.param(CONVERSATION_ID, username))
    .user(message)
    .call()
    .content();
```

RAG Demo with Documents

eazy bytes

1 Add Spring AI Tika Dependency - Enables reading and extracting text from PDFs, DOCs, etc., using Apache Tika.

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

2 Load Documents into Vector Store

Reads and splits the document into vectorized chunks for semantic retrieval.


```
@Component
public class HRPolicyLoader {

  @Value("classpath:Eazybytes_HR_Policies.pdf")
  Resource policyFile;

  private final VectorStore vectorStore;

  @PostConstruct
  public void loadPdf() throws IOException {
    TikaDocumentReader tikaDocumentReader = new TikaDocumentReader(policyFile);
        List<Document> docs = tikaDocumentReader.get();
        TextSplitter textSplitter =
                    TokenTextSplitter.builder().withChunkSize(100).withMaxNumChunks(400).build();
        vectorStore.add(textSplitter.split(docs));
  }
}
```

RAG Demo with Documents

eazy bytes

3 Query Endpoint Using Vector-Based Document Retrieval

Retrieves similar document chunks and injects them into the LLM prompt for contextual answers.

```
@GetMapping("/document/chat")
public ResponseEntity<String> documentChat( @RequestHeader("username") String username,
          @RequestParam("message") String message) {

  SearchRequest searchRequest = SearchRequest.builder()
      .query(message).topK(3).similarityThreshold(0.5).build();

  List<Document> similarDocs = vectorStore.similaritySearch(searchRequest);

  String similarContext = similarDocs.stream()
      .map(Document::getText)
      .collect(Collectors.joining(System.lineSeparator()));

  String answer = chatClient.prompt()
      .system(system -> system.text(hrSystemTemplate).param("documents", similarContext))
      .advisors(a -> a.param(CONVERSATION_ID, username))
      .user(message)
      .call()
      .content();

  return ResponseEntity.ok(answer);
}
```

Simplifying RAG with RetrievalAugmentationAdvisor

eazy bytes

🎯 What Is the Problem?

Previously, we had to manually perform vector search and pass the results to the LLM prompt. This approach worked but had duplicate logic and manual overhead.

```
SearchRequest searchRequest = SearchRequest.builder()
    .query(message).topK(3).similarityThreshold(0.5).build();

List<Document> similarDocs = vectorStore.similaritySearch(searchRequest);

String context = similarDocs.stream()
    .map(Document::getText).collect(Collectors.joining(System.lineSeparator()));

String answer = chatClient.prompt()
    .system(system -> system.text(template).param("documents", context))
    .user(message).call().content();
```

Simplifying RAG with RetrievalAugmentationAdvisor

eazy bytes

✅ Enter RetrievalAugmentationAdvisor

Spring AI provides a built-in advisor to automatically handle document retrieval from a vector store!

🔧 How You Configured It

1️⃣ Define the Advisor Bean

```
@Bean
RetrievalAugmentationAdvisor retrievalAugmentationAdvisor(VectorStore vectorStore)
{
    return RetrievalAugmentationAdvisor.builder()
        .documentRetriever(
            VectorStoreDocumentRetriever.builder()
                .vectorStore(vectorStore)
                .topK(3)
                .similarityThreshold(0.5)
                .build()).build();
}
```

Uses VectorStoreDocumentRetriever to plug in Qdrant
Retrieves top 3 similar chunks automatically

Simplifying RAG with RetrievalAugmentationAdvisor

eazy bytes

2 Register Advisor in ChatClient

```
@Bean("chatMemoryChatClient")
public ChatClient chatClient(ChatClient.Builder builder, ChatMemory chatMemory,
                             RetrievalAugmentationAdvisor advisor) {

    return builder.defaultAdvisors(List.of(
            new SimpleLoggerAdvisor(),
            new TokenUsageAuditAdvisor(),
            MessageChatMemoryAdvisor.builder(chatMemory).build(),
            advisor))
        .build();
}
```

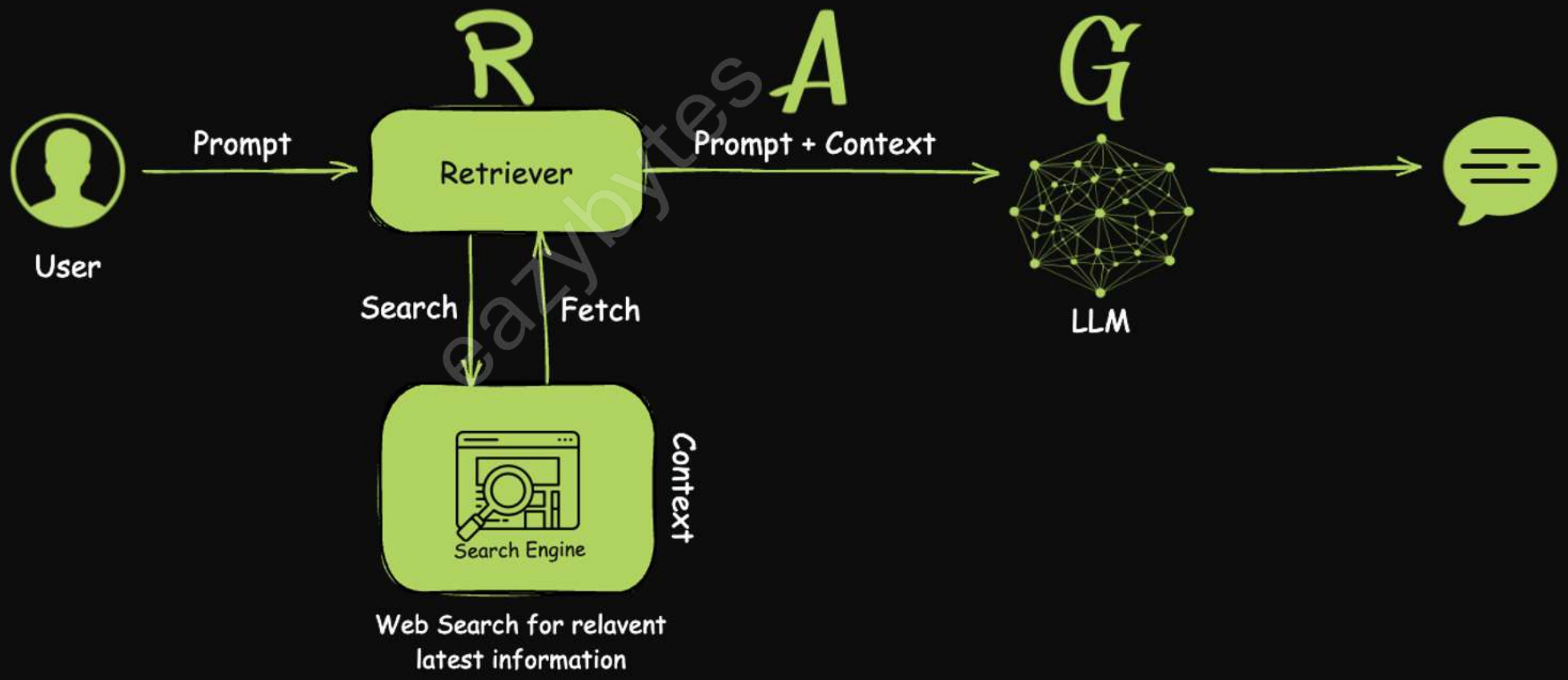

Final API – Clean and Powerful

```
@GetMapping("/document/chat")
public ResponseEntity<String> documentChat(@RequestHeader("username") String username,
    @RequestParam("message") String message) {

    String answer = chatClient.prompt()
        .advisors(a -> a.param(CONVERSATION_ID, username))
        .user(message).call().content();

    return ResponseEntity.ok(answer);
}
```

RAG flow with Web Search

eazy bytes

Enhanced RAG system with live web search integration via Tavily API, mapping results into Spring AI documents to provide accurate, up-to-date knowledge in AI responses

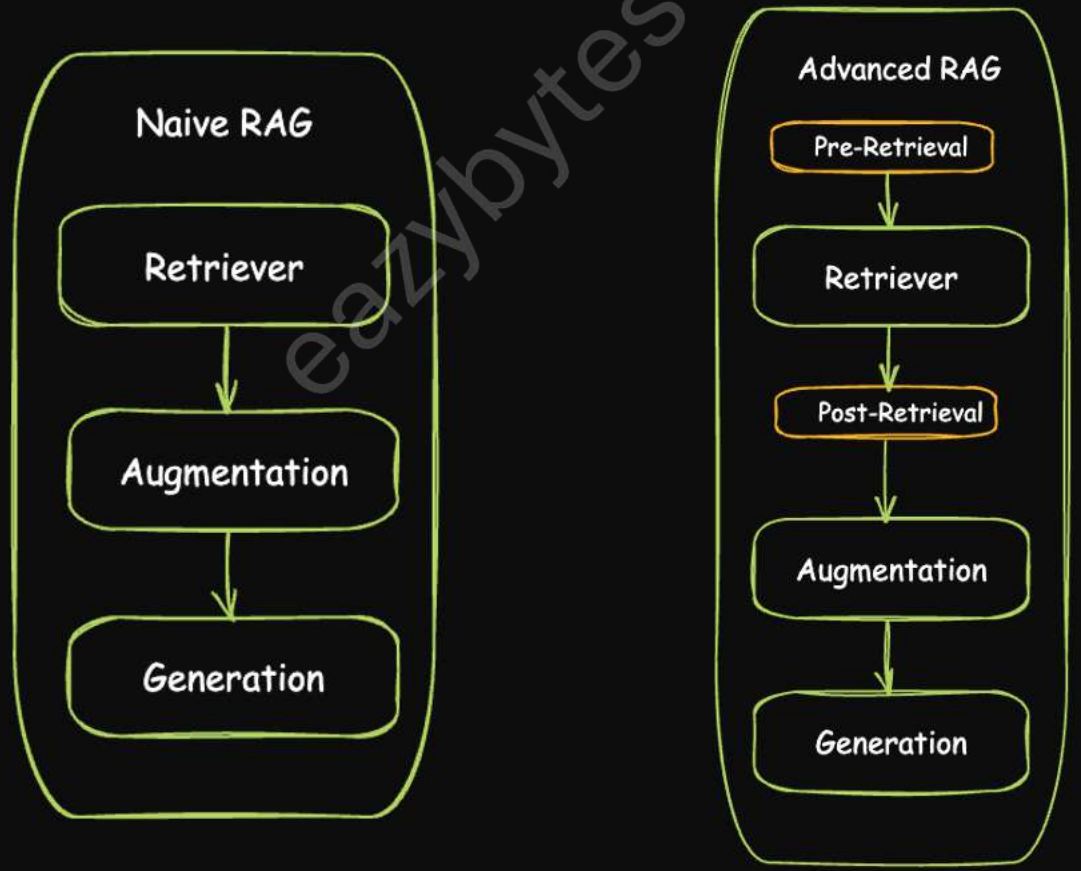


Naive RAG & Advanced RAG

eazy bytes

Naive RAG: A simple Retrieval-Augmented Generation approach where retrieved documents are directly passed to the LLM without optimization, which may lead to irrelevant or redundant context.

Advanced RAG (Pre-Retrieval & Post-Retrieval): An enhanced RAG pipeline that applies techniques like query rewriting, filtering, reranking, and summarization before and after retrieval to deliver more accurate, relevant, and concise results.

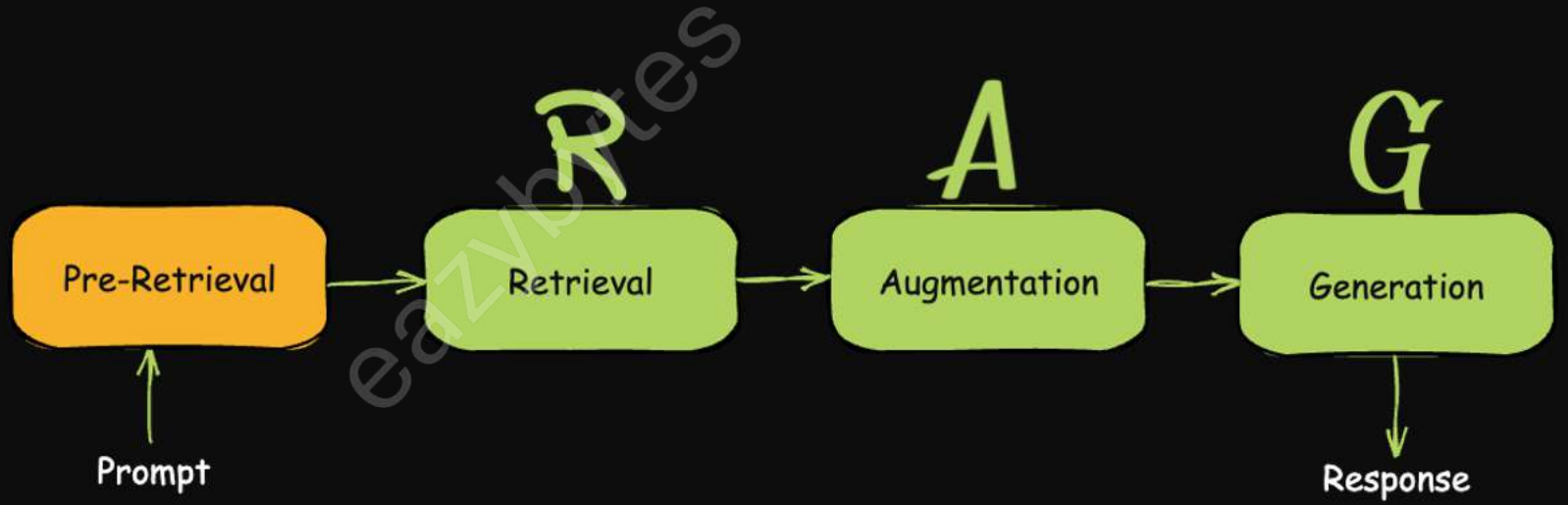


RAG with Pre-Retrieval

eazy bytes

Pre-Retrieval

- This step transforms the input query into a more effective form for retrieval.
- It handles poorly structured queries, resolves ambiguous terms, and simplifies complex vocabulary.
- It can also adapt queries across unsupported languages for better search results.

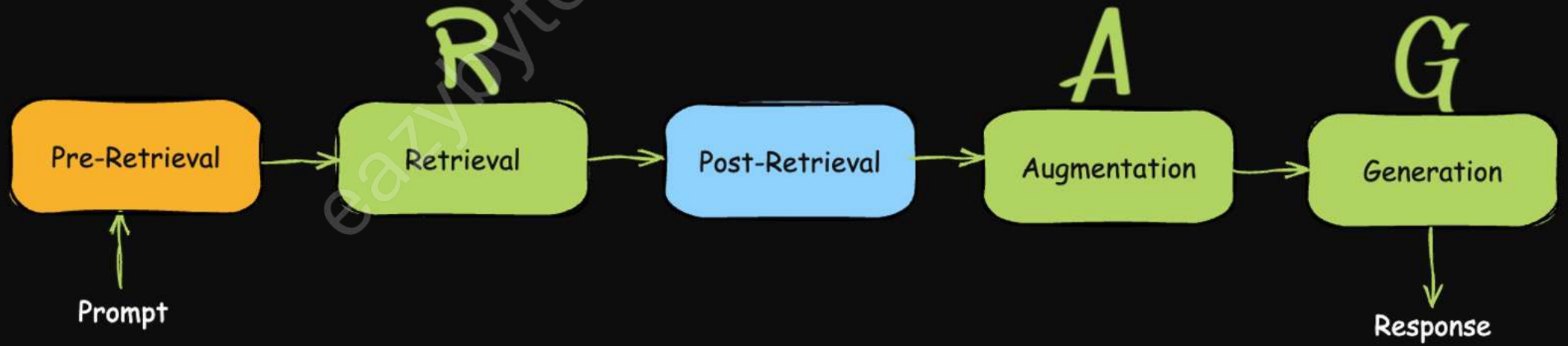


RAG with Post-Retrieval

eazy bytes

Post-Retrieval

- This step refines the retrieved documents before augmentation.
- It tackles issues like "lost-in-the-middle," context length limits, and redundant data.
- Examples include ranking relevance, compressing content, or masking sensitive information (e.g., emails, phone numbers).



Tool Calling in AI

eazy bytes

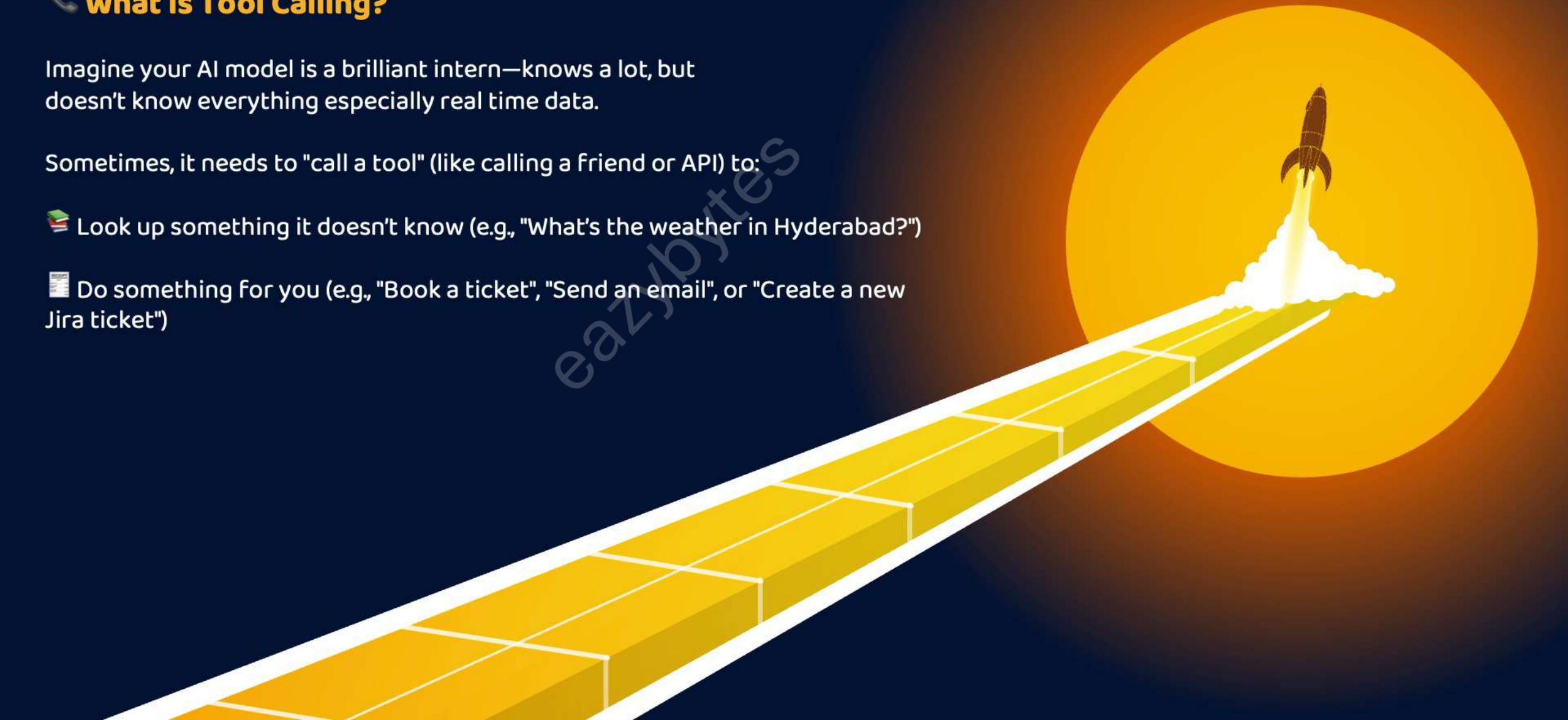
📞 What is Tool Calling?

Imagine your AI model is a brilliant intern—knows a lot, but doesn't know everything especially real time data.

Sometimes, it needs to "call a tool" (like calling a friend or API) to:

📚 Look up something it doesn't know (e.g., "What's the weather in Hyderabad?")

📄 Do something for you (e.g., "Book a ticket", "Send an email", or "Create a new Jira ticket")



Tool Calling in AI

eazy bytes

Two Superpowers of Tool Calling:

Superpower	What it does	Real-world analogy
📖 Information Retrieval	Fetches real-time data from external systems	Like googling the latest cricket score for you
🚀 Taking Action	Performs an action in your app	Like asking Siri to "Set an alarm at 5 AM"

So... How Tool Calling Different from RAG?

Tool Calling	RAG
Like calling a plumber to fix a leak	Like reading a DIY manual to fix it yourself
The model says: "I need help, here's what to do"	The model says: "Let me read some documents and figure it out"
Takes live action or fetches live data	Focuses on augmenting answers using retrieved content
Requires client app to execute the tool	Just retrieves documents and continues generating response

Let's Make Our AI Call a Tool – With Spring AI!

eazy bytes

What Are We Doing?

We're building an AI app that can:

- Understand user queries like "What's the time in New York?"
- Call real Java methods (tools!) to get the live answer
- Respond like a smart assistant 📅🕒

This is called Tool Calling — when the AI calls backend methods to do real work!

Key Components

1. The Tool Class – DateTimeTools.java

```
@Component
public class TimeTools {

    @Tool(name="getCurrentLocalTime", description = "Get the current time in the user's timezone")
    String getCurrentLocalTime() { }


    @Tool(name = "getCurrentTime",description = "Get the current time in the specified time zone.")
    public String getCurrentTime(@ToolParam(
            description = "Value representing the time zone") String timeZone) { }
}
```

Two methods: One uses the user's current locale and other accepts a time zone string (like "Asia/Kolkata"). Think of these like utility methods that AI can call.

Let's Make Our AI Call a Tool – With Spring AI!

eazy bytes

2. The Tool-Aware ChatClient – ToolsChatClientConfig

```
chatClientBuilder
  .defaultTools(timeTools)
```

We're injecting the tool into the ChatClient so the model knows about it

3. The Controller – ToolsCallingController

```
@GetMapping("/local-time")
public ResponseEntity<String> localTime (...)
```

Uses chatClient.prompt() to interact with the AI

What Makes This Powerful?

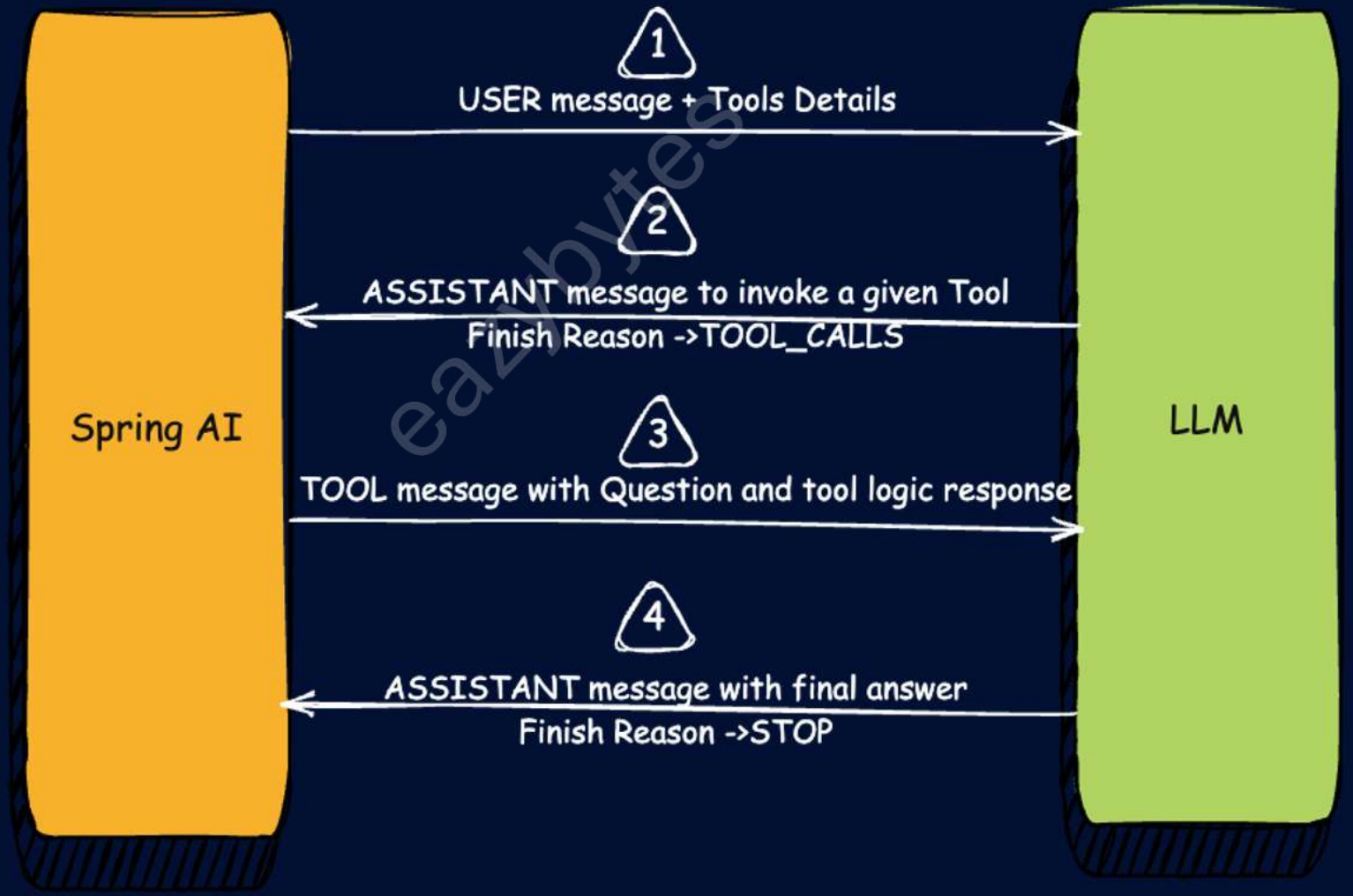
- The model isn't guessing — it's using real-time data!
- Tools are type-safe, reusable, and easy to expose
- You're extending the LLM with your own Java logic

You can have multiple @Tool-annotated methods in a class, and all will be available to the LLM. Just ensure each has a unique name—either set via the name attribute or by defaulting to the method name.

How the Tools get called by LLM ?

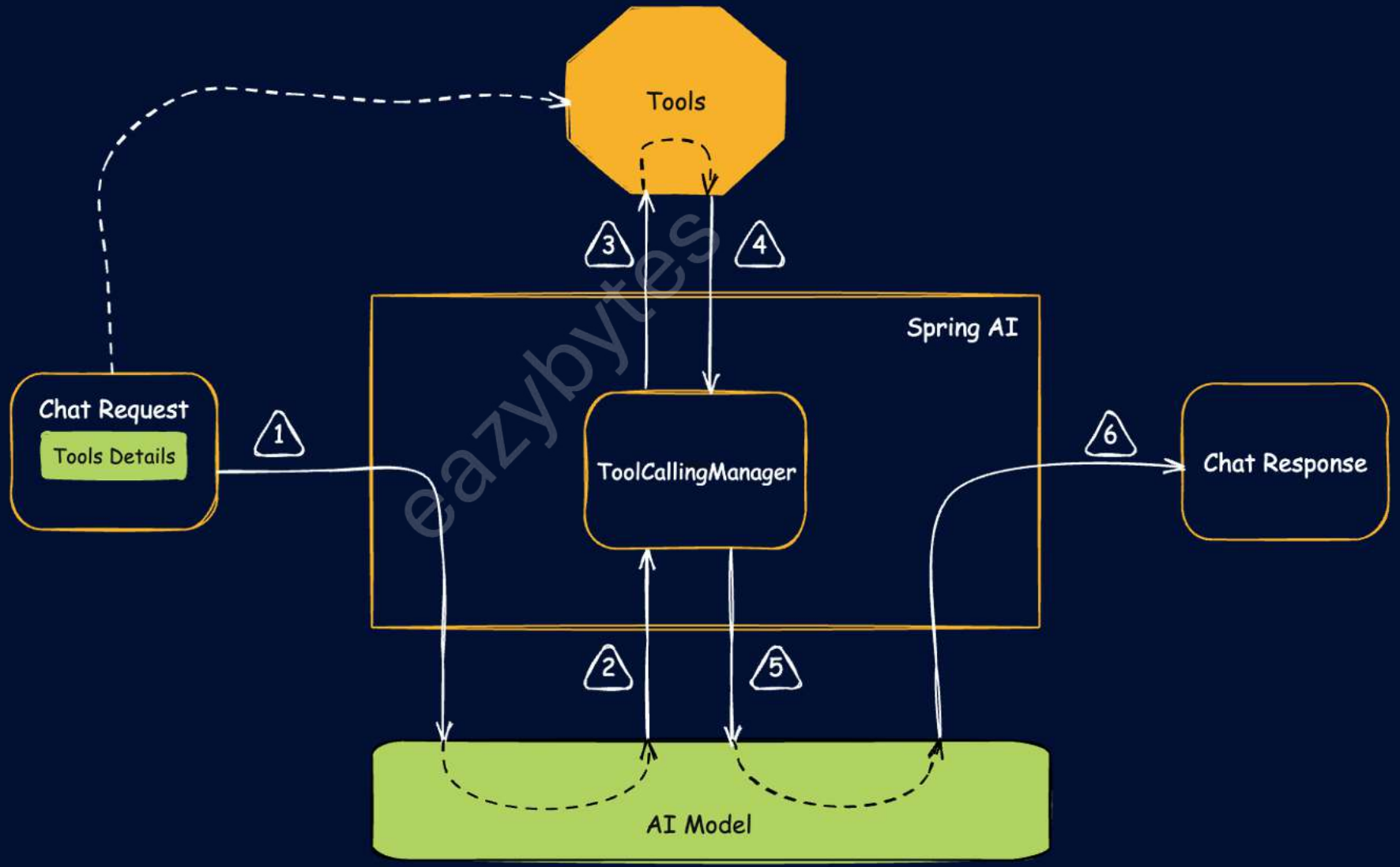
eazy bytes

It may seem like Spring AI exposes your tool as an endpoint for OpenAI to call, but that's not what happens. Instead, a behind-the-scenes conversation occurs between your app and the model whenever a tool is involved.



How the Tools get called by LLM ?

eazy bytes



Tool Calling in AI

eazy bytes

Who Actually executes the Tools logic ?

- LLMs don't really "run" tools. They're like a project manager—they delegate.
- The client application receives the request, calls the tool/API, and sends the result back to the model.

How Spring AI Helps:

- You can define tools as simple Java beans or annotated methods.
- Spring AI:
  - ✓ Detects tool call requests from the model
  - ✓ Maps input arguments automatically
  - ✓ Executes the right tool method
  - ✓ Sends results back to the model like a tech-savvy assistant!

Quick Analogy to Remember:

"If RAG is your AI reading a book to answer a question, Tool Calling is your AI dialing a friend to get things done!"

Tool Context – Sharing Extra Info

eazy bytes

ToolContext is a mechanism in Spring AI that allows you to inject extra data—not part of the main prompt or tool arguments—into the tool execution flow. This is useful for scenarios where a tool needs access to user/session-specific metadata like username, userId, or organizationId.

1. Passing Contextual Data

This context is not directly visible to the AI model. It is meant purely for backend logic during tool invocation.

```
String answer = chatClient.prompt()
                .advisors(a -> a.param(CONVERSATION_ID, username))
                .user(message)
                .tools(helpDeskTools)
                .toolContext(Map.of("username", username))
                .call().content();
```


2. Accessing Contextual Data in the Tool

The ToolContext is automatically injected when the tool is called. toolContext.getContext().get("username"): Retrieves the context value (username) that was passed earlier.

```
@Tool(description = "Fetch the status of the open tickets based on a given username")
List<HelpDeskTicket> getTicketStatus(ToolContext toolContext) {
    String username = (String) toolContext.getContext().get("username");
    return service.getTicketsByUsername(username);
}
```

Tool Execution & Result Conversion

eazy bytes

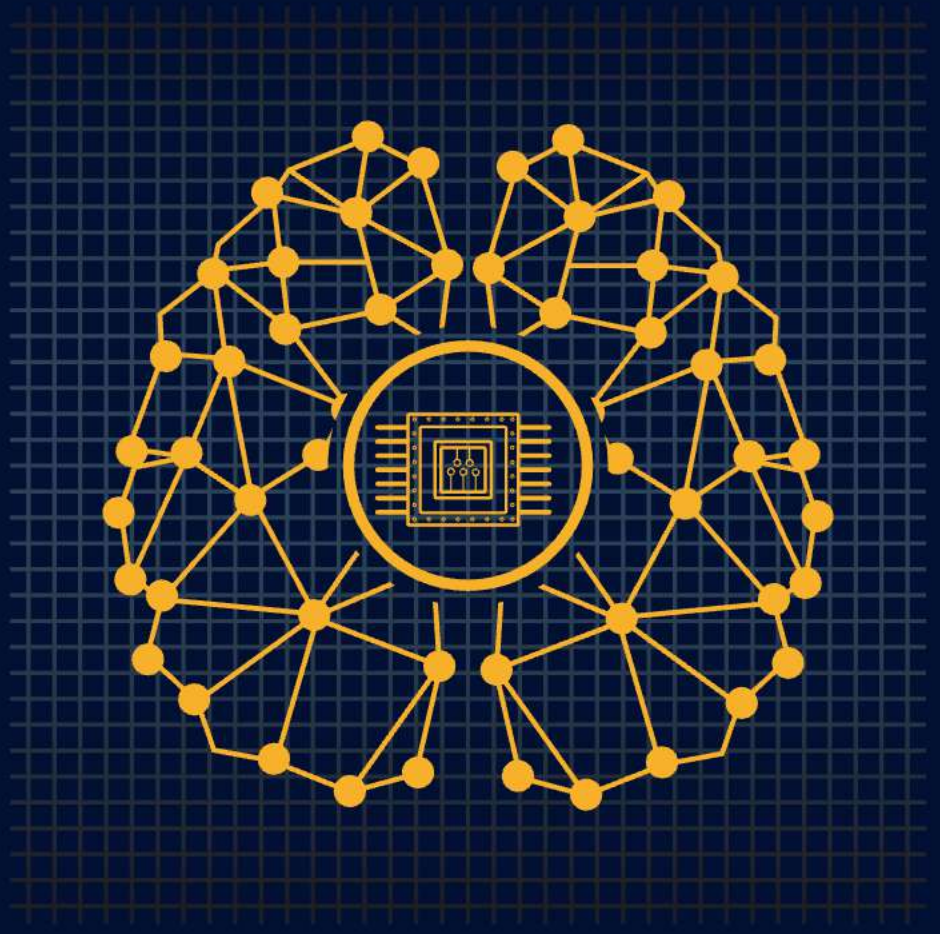
Tool Execution – Meet the Manager

Tool execution involves invoking a tool with the given input arguments and returning the result. This process is managed by the ToolCallingManager interface, which oversees the entire execution lifecycle.

It:

- Invokes the tool
- Passes the arguments
- Coordinates everything

DefaultToolCallingManager is the auto-configured implementation of the ToolCallingManager interface. To customize tool execution behavior, you can define and register your own ToolCallingManager bean.



Result Conversion – Making the Output Speak Human

The tool call result is converted to a String using the ToolCallResultConverter interface and sent back to the AI model. By default, it's serialized to JSON with Jackson via DefaultToolCallResultConverter, but you can customize this by providing your own implementation.

Return Direct – Skip the Model Response

eazy bytes

By default, Tool result is sent back to the LLM so it can reason and continue chatting

But sometimes, you want to:

⚡ Return the tool's result directly to the user
🚫 Skip extra processing by the model

The returnDirect flag of @Tools annotation tells Spring AI not to send the tool's output back to the AI model for further processing. Instead, it immediately returns the tool's result directly as the final response to the user (or the caller).


```
@Tool(description = "Create the Support Ticket", returnDirect = true)
String createTicket( @ToolParam(required = true, description = "Details to create a Support ticket")
        TicketRequest ticketRequest,ToolContext toolContext) {
    String username = (String) toolContext.getContext().get("username");
    HelpDeskTicket savedTicket = service.createTicket(ticketRequest, username);
    return "Ticket #" + savedTicket.getId() + " created successfully for user " + savedTicket.getUsername();
}
```

When to Use returnDirect = true
Use it when:

- The tool generates a final, human-readable message.
- You don't want the model to alter, summarize, or continue the reasoning after the tool execution.
- You want to reduce latency by skipping the post-processing step.

Handling Errors Gracefully – ToolExecutionException

eazy bytes

If your tool throws an error, Spring wraps it in a ToolExecutionException

By default, Message is sent back to the LLM. Instead we can throw the exception to client app,

```
@Bean
ToolExecutionExceptionProcessor processor() {
    return new DefaultToolExecutionExceptionProcessor(true);
}
```

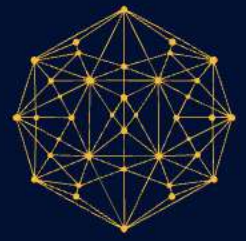

If you're using Spring AI Boot Starters, DefaultToolExecutionExceptionProcessor is the default handler for tool execution errors.

By default, it sends error messages back to the model.

You can set the alwaysThrow flag in its constructor to true to throw exceptions instead of returning error messages.

Generative AI Journey

1 LLM

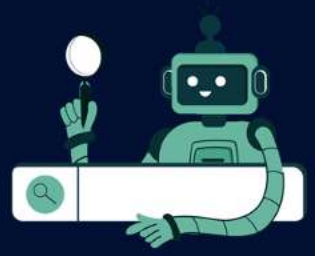


- Just answers from memory (training data).
- No real-time updates.

2 LLM + RAG

- Pulls data from external sources (e.g., private docs).
- Still only talks, doesn't take action.
- Smarter and more up-to-date advice.

3 LLM + Tools



- Can call APIs/tools: flights, hotels, maps.
- Starts doing things for you, not just recommending.

4 AI Agent

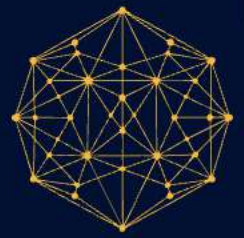
- Break down complex tasks into steps
- Make decisions based on context
- Execute multi-step plans

5 Agentic AI

- Learns from experience and improves over time
- Collaborates with other AI agents
- Anticipates your needs
- Handles complex, ambiguous requests

Generative AI Journey

1. LLM

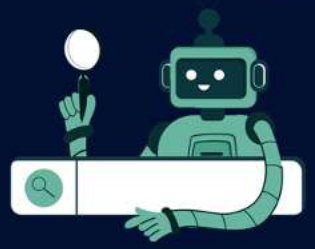


KNOWLEDGE

2. LLM + RAG

PERSONALIZED KNOWLEDGE

3. LLM + Tools



LIVE DATA

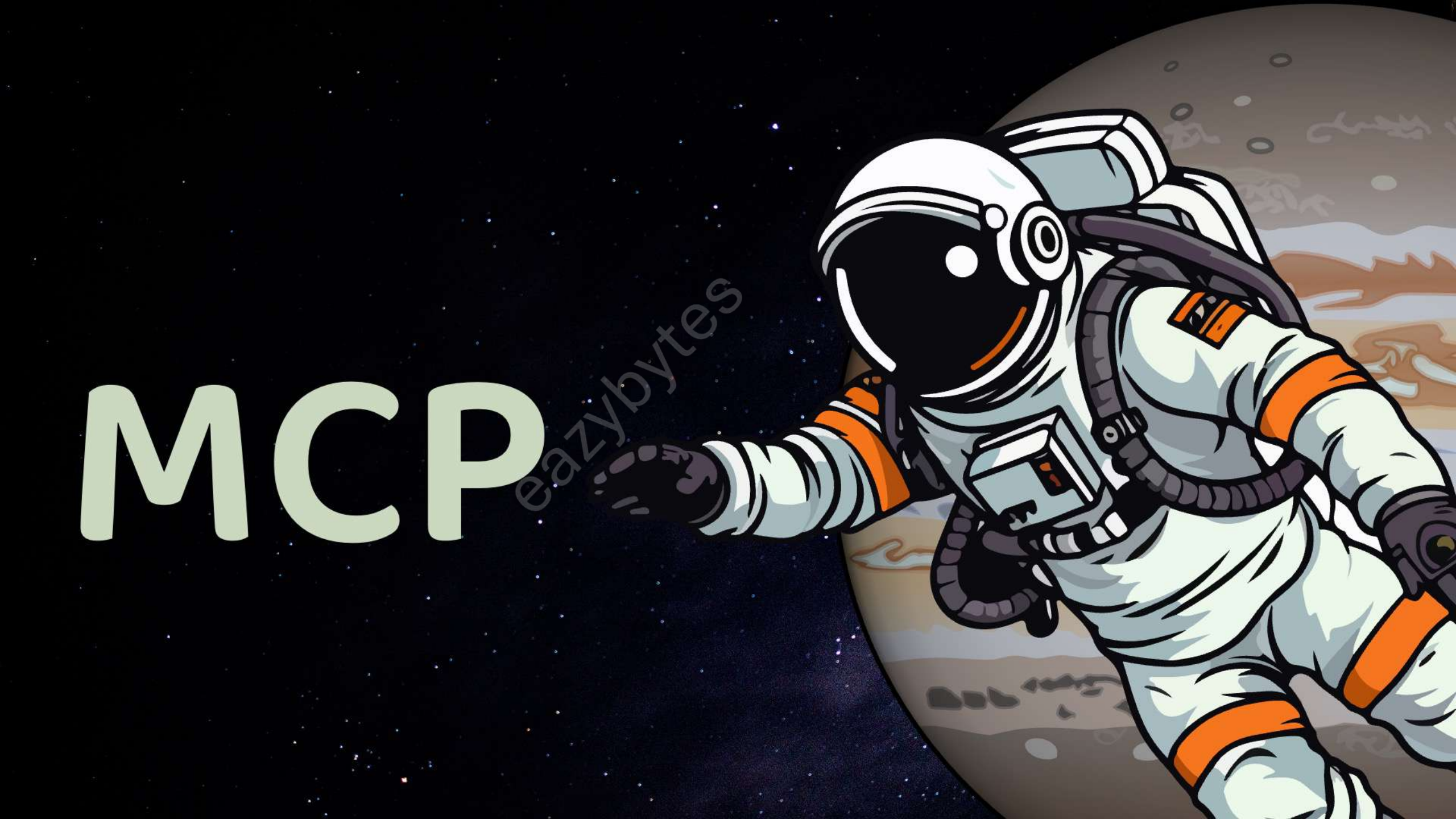
4. AI Agent

ACTIONS

5. Agentic AI

INTELLIGENCE

MCP



Model Context Protocol (MCP)

eazy bytes

MCP is an open protocol that standardizes how applications provide context to LLMs. Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools.

How MCP Works

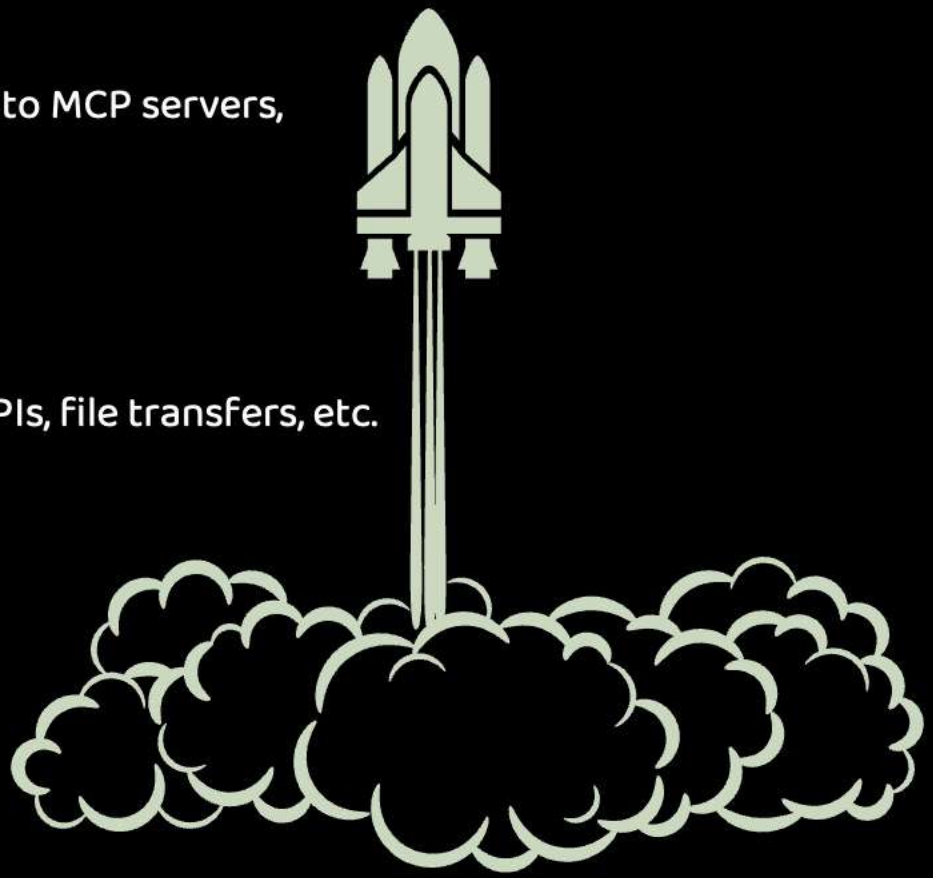
MCP follows a client-server architecture. In other words, MCP clients send requests to MCP servers, which respond with access to tools, resources, and prompts.

Why do we need MCP when we already have HTTP?

HTTP is a general web protocol for any client-server communication - web pages, APIs, file transfers, etc. It's stateless and doesn't understand AI concepts.

MCP is AI-specific and built on top of protocols like HTTP.

Think of MCP as a specialized layer that makes HTTP more suitable for AI use cases. Similar to how GraphQL adds a query layer on top of HTTP. You could build AI integrations with raw HTTP, but MCP provides the AI-specific standards and abstractions that make it much more efficient and reliable.



MCP Architecture Overview

eazy bytes

Model Context Protocol (MCP) uses a host–client–server architecture to streamline how AI models interact with tools and contextual data.

MCP Host: The central coordinator—typically a chatbot, IDE, or AI application—that manages permissions, tool access, and session context. It decides when to invoke a tool, either based on user input or automatically.

MCP Client: Launched by the host, it handles all communication between the host and a specific MCP server. It's responsible for sending tool requests and receiving responses.

MCP Server: Connects to a local or remote system (e.g., a database, file storage, or third-party API) and exposes specific capabilities like "search file" or "create task". Developers can either use public MCP servers or build their own.

Let's say you've built a Spring AI application that uses OpenAI's LLM and interacts with GitHub through MCP-exposed tools.

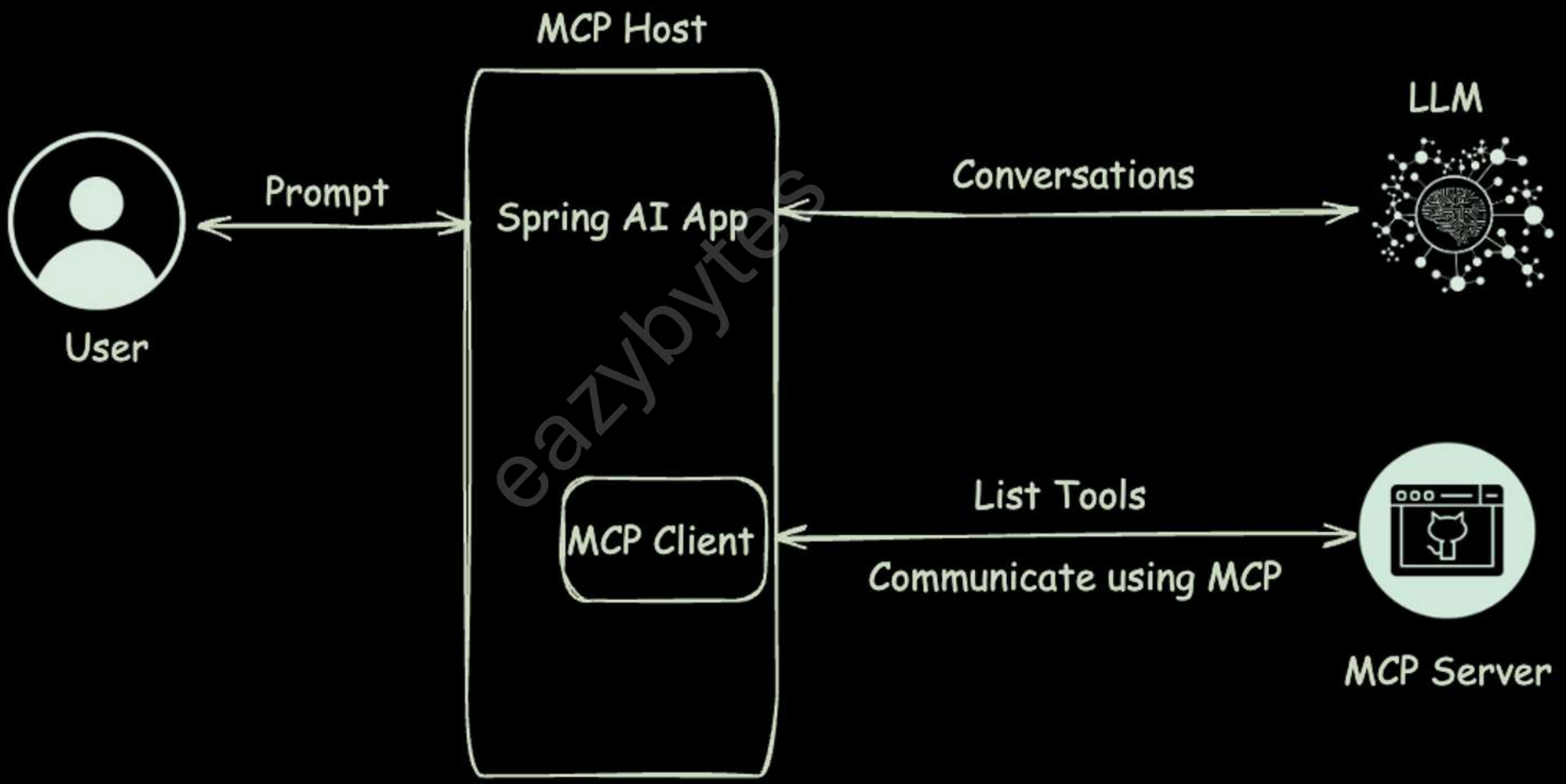
MCP Host → Your Spring AI app is the host. It manages the conversation, decides when to call a GitHub tool (e.g., "create issue")

MCP Client → The Spring AI app also initializes the MCP client to handle tool invocation requests and send them to the GitHub MCP server. It handles communication between Spring AI app and GitHub server

MCP Server → GitHub, in this case, acts as the server. It exposes GitHub capabilities ("create issue", "search repos", "get PRs") as callable tools via MCP.

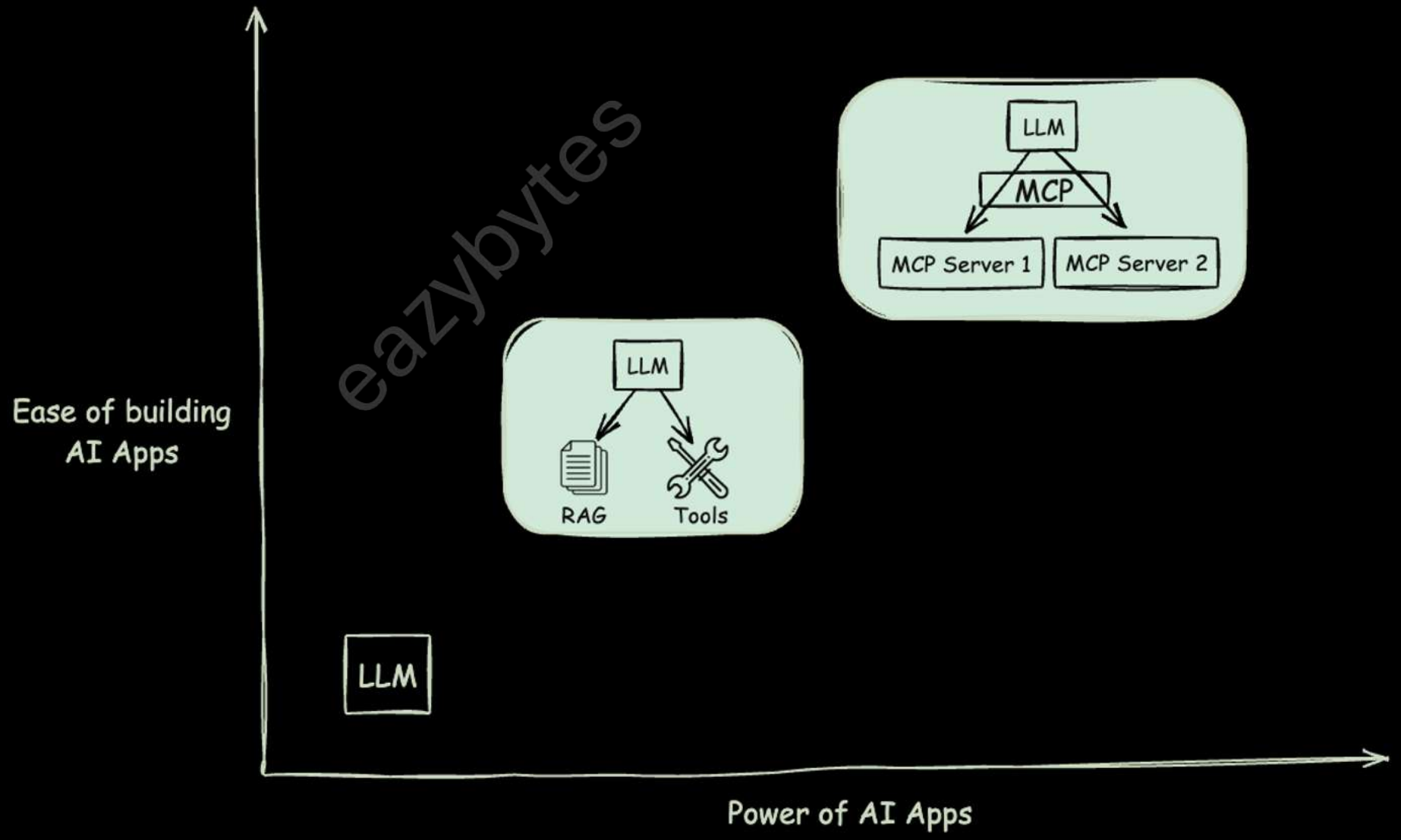
MCP Architecture Overview

eazy bytes



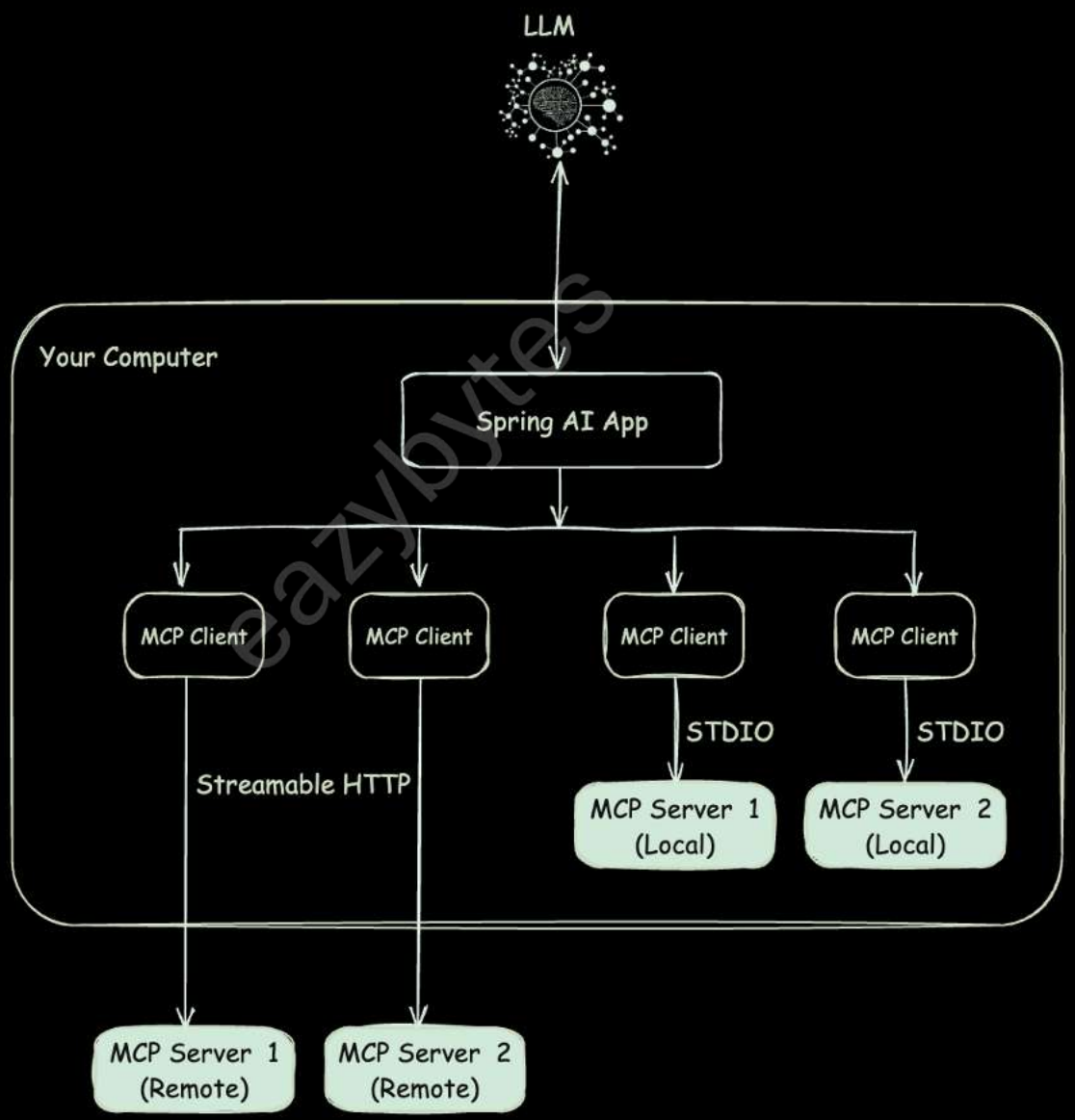
MCP Architecture Overview

eazy bytes



Built-in Transport Types in MCP

eazy bytes



Built-in Transport Types in MCP

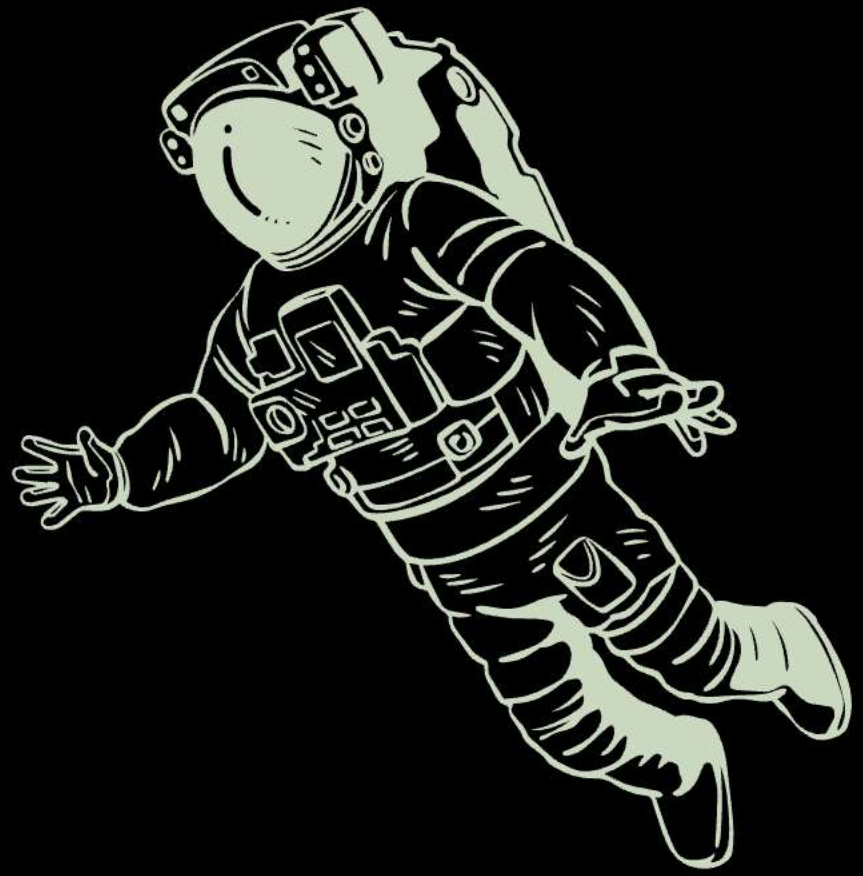
eazy bytes

MCP allows communication between clients and servers using built-in transport types. Currently, it supports two main ones:

Standard Input/Output (stdio)

This is like talking directly through a walkie-talkie. Data goes in one side and comes out the other — no internet, no browser, just local communication. Best for:

- Command-line tools (like terminal-based apps)
- Local setups (when your app runs on your own machine)
- Simple tools or scripts
- Quick and easy communication between processes



Streamable HTTP

This is the internet-friendly way! It uses HTTP POST to send data and optionally uses Server-Sent Events (SSE) to stream responses from server to client. Best for:

- Web apps or browser-based tools
- Scenarios needing two-way, ongoing communication
- Multiple users connected at the same time
- Keeping conversations alive across many requests

Built-in Transport Types in MCP

eazy bytes

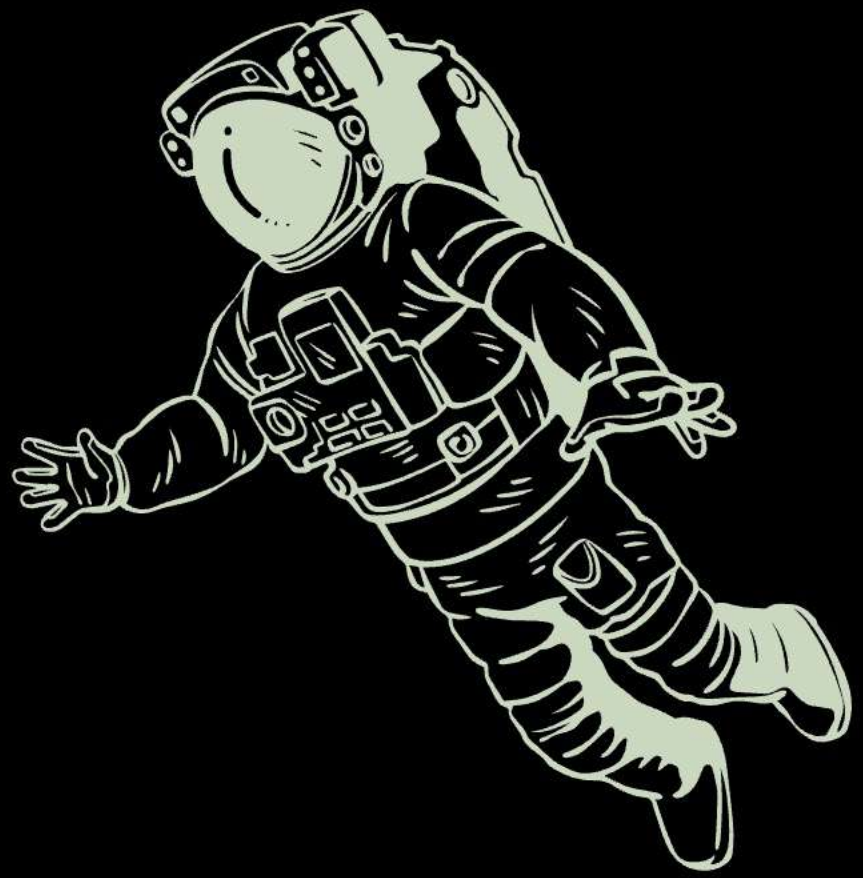
Server-Sent Events (SSE) — Deprecated

SSE was once a standalone transport type for streaming data from server to client only. But it's now old-school and deprecated since version 2024-11-05.

📦 It's now part of Streamable HTTP, so you don't need to use it alone.

SSE was useful when:

- Only one-way streaming (server → client) was needed
- Network was limited
- You just needed quick live updates (like stock prices)



Built-in Transport Types in MCP

eazy bytes

Why was SSE deprecated and Streamable HTTP was introduced ?

To fix the issues with SSE and make the protocol more powerful and flexible, MCP introduced Streamable HTTP. It combines the best of both worlds:

Feature	SSE	Streamable HTTP
Server-to-client streaming	✅	✅ (via optional SSE)
Client-to-server communication	❌	✅ (via HTTP POST)
Session management	❌	✅ (with Mcp-Session-Id)
Two-way communication	❌	✅
Multiple concurrent clients	❌	✅
Better compatibility	❌	✅

Steps to Set Up MCP Client with Filesystem Server in Spring AI

eazy bytes

Add Required Dependencies

In your pom.xml, include:

- spring-ai-starter-mcp-client – for using MCP protocol.
- spring-ai-starter-model-openai – for model interaction (fallback or default).
- Spring Boot web starter – to expose REST endpoints.

This sets up the base for Spring AI + MCP integration.

Configure MCP Server in JSON

Create a file like mcp-servers.json. Defines how to start the MCP Filesystem Server (here via npx).


```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "/Users/eazybytes/mcp"
      ]
    }
  }
}
```

Steps to Set Up MCP Client with Filesystem Server in Spring AI

eazy bytes

Set Property to Use This MCP Server

In application.properties or application.yml, add:

```
spring.ai.mcp.client.stdio.servers-configuration=classpath:mcp-servers.json
```

Tells Spring AI where to find the MCP server configuration.

Create Controller to Interact with the MCP Client

Define a controller McpClientFSController with an endpoint /api/chat that:

- Injects ChatClient using the builder.
- Registers tool callbacks (via ToolCallbackProvider).
- Adds a simple logging advisor.
- Accepts user message via GET and sends it to the MCP server.

Connects user input to the AI model via MCP protocol.

Building an MCP Server with Spring AI and STDIO Support

eazy bytes

Step 1 – Add the MCP Server Dependency

Spring AI provides a dedicated starter to simplify building an MCP-compliant server. This dependency auto-configures the server with MCP support and tool registration.

```
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-mcp-server</artifactId>
</dependency>
```

Step 2 – Disable Web Environment for STDIO

```
spring.main.web-application-type=none
logging.level.root=ERROR
spring.main.banner-mode=off
```


Step 3 – Register Tool Callbacks

```
@Configuration
public class McpServerConfig {

    @Bean
    List<ToolCallback> toolCallbacks(HelpDeskTools helpDeskTools) {
        return List.of(ToolCallbacks.from(helpDeskTools));
    }
}
```

Building an MCP Server with Spring AI and STDIO Support

eazy bytes

Step 4 – Define the MCP Server in Client App (mcp-servers.json)

In the MCP client app, define the server in a .json file with the command to run the MCP server over STDIO. This makes the server discoverable and callable by the MCP host (e.g., Spring AI client).

```
{
  "mcpServers": {
    "spring-ai-mcp": {
      "command": "/path/to/java",
      "args": [
        "-jar",
        "/path/to/mcpserverstdio-0.0.1-SNAPSHOT.jar"
      ]
    }
  }
}
```


Step 5 – Wire Up MCP Client to Server

```
spring.ai.mcp.client.stdio.servers-configuration=classpath:mcp-servers.json
```

Tell the Spring AI client to use the custom mcp-servers.json configuration file. This connects your Spring AI app with the MCP server via STDIO at runtime.

Building an MCP Server with Remote Invocation using Spring AI

eazy bytes

Step 1 – Add the WebMVC MCP Server Dependency

This dependency sets up an HTTP-based MCP server using Spring WebMVC, enabling remote tool invocation over HTTP.

```
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-mcp-server-webmvc</artifactId>
</dependency>
```

Step 2 – Define the Server Port

```
server.port=8090
```

Step 3 – Register Tool Callbacks

```
@Configuration
public class McpServerConfig {

    @Bean
    List<ToolCallback> toolCallbacks(HelpDeskTools helpDeskTools) {
        return List.of(ToolCallbacks.from(helpDeskTools));
    }
}
```

Building an MCP Server with Remote Invocation using Spring AI

eazy bytes

Step 4 – Configure MCP Client for Remote Access

On the client side, configure the MCP client to connect to the server using HTTP/SSE. This tells Spring AI to send tool invocation requests to http://localhost:8090.

```
spring.ai.mcp.client.sse.connections.eazybytes.url=http://localhost:8090
```

eazy bytes

UNIT TESTING

Why Testing Is Absolutely Essential

Example 1: NASA's $125 Million Mistake

In 1999, NASA's Mars Climate Orbiter was lost in space. Because, one team used Imperial units (pounds), while another used Metric (newtons) — and the mismatch wasn't caught due to missing integration testing.

Example 2: PayPal Bug Sent Users Quadrillions

A PayPal user once found $92 quadrillion credited to his account due to a calculation error during transaction testing. Because the test system didn't correctly validate numeric overflows or edge cases in money transfers.

Testing Challenges with Generative AI

eazy bytes

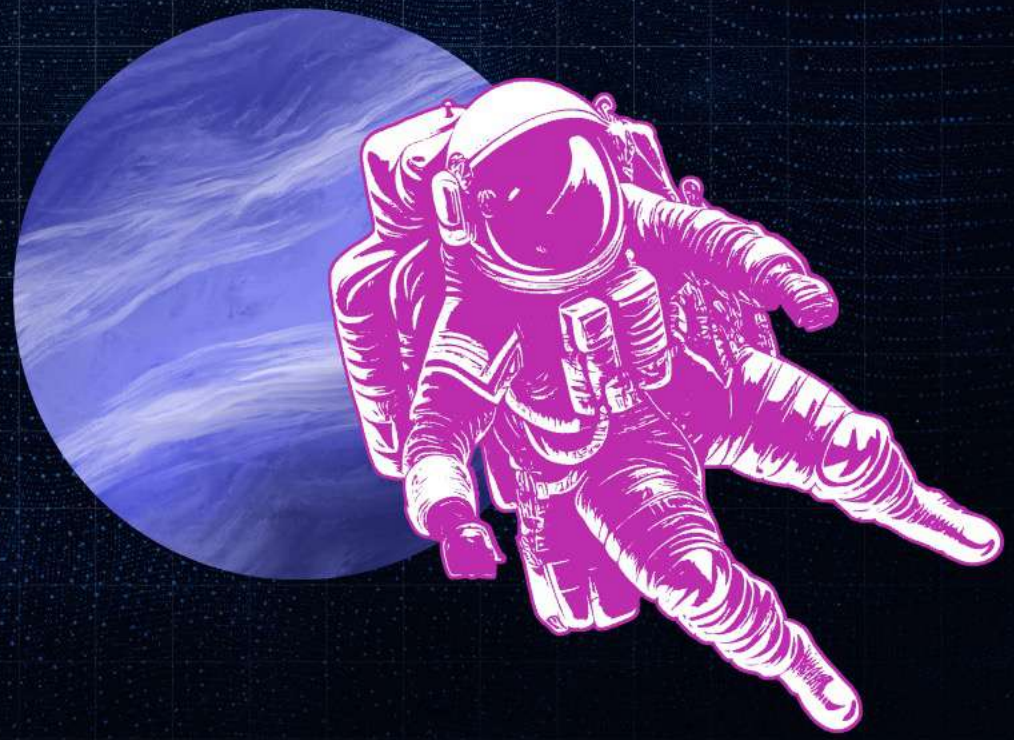
The Problem: Non-Deterministic Responses

- Generative AI components (like LLMs) don't return the same output for the same prompt every time.
- Traditional testing approach like assertEquals(expected, actual) doesn't work.
- This makes automated unit testing unreliable for LLM-generated outputs.

The Solution: Spring AI Evaluators

What is an Evaluator?

A component that checks if the LLM response is appropriate for the given prompt. Instead of looking for exact matches, it checks for satisfactory or acceptable responses.

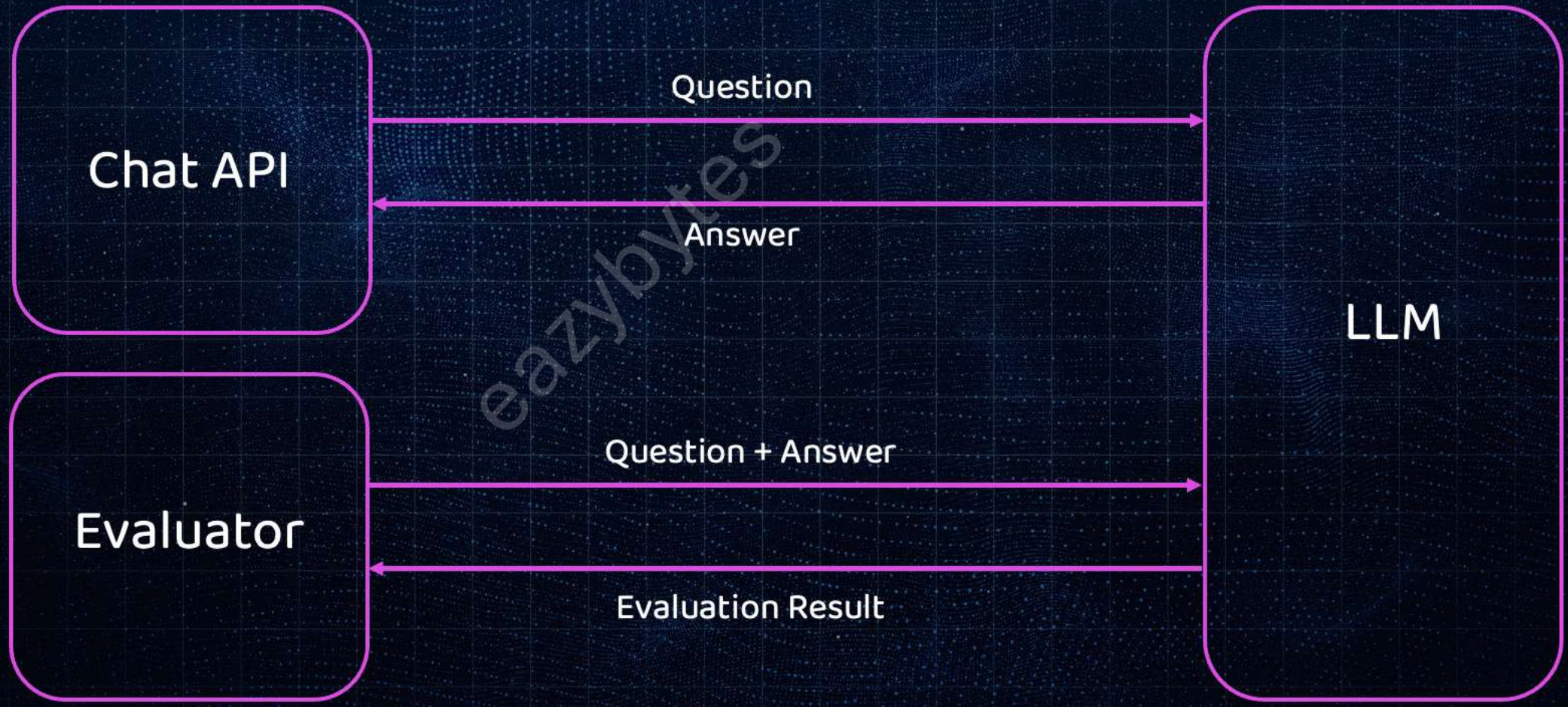


🧠 How It Works:

- Takes two inputs:
  - ✓ The prompt submitted to the LLM
  - ✓ The response returned by the LLM
- Uses another LLM to decide if the response passes.

Testing Challenges with Generative AI

eazy bytes



Spring AI Evaluators: Making LLM Testing Smart

eazy bytes

```
@FunctionalInterface
public interface Evaluator {

    EvaluationResponse evaluate(EvaluationRequest evaluationRequest);

}
```

- A contract for evaluating LLM responses.
- Takes an EvaluationRequest (contains prompt, response, and context).
- Returns an EvaluationResponse indicating whether the output is valid/acceptable.

Implementations of Evaluator

Below are the two implementation classes of Evaluator interface which can be used to validate the response,

- `RelevancyEvaluator`
- `FactCheckingEvaluator`

Spring AI Evaluators: Making LLM Testing Smart

eazy bytes

```
public class RelevancyEvaluator implements Evaluator
```

- Purpose: Checks how relevant the response is to the original query.
- Use Case: Ensures LLM doesn't drift away or add unrelated info.
- Ideal for: Chatbots, search-based assistants, summarization use cases.

```
public class FactCheckingEvaluator implements Evaluator
```

- Purpose: Checks factual accuracy of the LLM response based on a provided document or context.
- Tackles the LLM issue of hallucination (making up facts).
- Ideal for: Legal, medical, or financial domains where truth matters.

Evaluators = Quality Control for AI responses.
They ensure what your AI says is not just smart-sounding, but also relevant and truthful.

Relevancy Check — General Prompt

eazy bytes

Goal: Ensure the AI response is relevant to the user's question.

Prompt: String question = "What is the capital of India?";

AI actual response: String aiResponse = chatController.chat(question);

Evaluator Usage:

```
EvaluationRequest evaluationRequest = new EvaluationRequest(
    question, List.of(), // No context (non-RAG)
    aiResponse);
EvaluationResponse evaluationResponse =  relevancyEvaluator.evaluate(evaluationRequest);
assertThat(aiResponse).isNotBlank();
assertThat(evaluationResponse.isPass()).isTrue();
assertThat(evaluationResponse.getScore()).isGreaterThan(minRelevancyScore);
```

Factual Accuracy Check

eazy bytes

Goal: Verify the AI gives a factually accurate response.

Prompt: String question = "Who discovered the law of universal gravitation?";

AI actual response: String aiResponse = chatController.chat(question);

Evaluator Usage:

```
EvaluationResponse evaluationResponse = factCheckingEvaluator.evaluate(
    new EvaluationRequest(question, aiResponse));
assertThat(aiResponse).isNotBlank();
assertThat(evaluationResponse.isPass()).isTrue();
```

Factual Accuracy Check with RAG (Context-Aware)

eazy bytes

Goal: Ensure the AI's answer is grounded in the provided HR policy context.

Prompt: String question = "How many paid leaves do employees get annually?";

AI actual response: String aiResponse = chatController.promptStuffing(question);

Context: String context = hrPolicyTemplate.getContentAsString(StandardCharsets.UTF_8);

Evaluator Usage:

```
EvaluationResponse evaluationResponse = factCheckingEvaluator.evaluate(
    new EvaluationRequest(
        question,
        List.of(new Document(context)),
        aiResponse
    )
);
assertThat(aiResponse).isNotBlank();
assertThat(evaluationResponse.isPass()).isTrue();
```

Runtime Evaluation in App logic

eazy bytes

The Problem: Tests Can Get Lucky

LLMs are non-deterministic — they may return good answers during tests.....but bad ones in production 😬
Even passing evaluator-based tests don't guarantee safe runtime

The Solution: Use Evaluators at Runtime

Apply RelevancyEvaluator (or FactCheckingEvaluator) after every response. Filter out responses that are off-topic or irrelevant. Combine with Spring Retry to try again if the response is not good

App Code – Smart & Safe Chat


```
@GetMapping("/evaluate/chat")
@Retryable(retryFor = IrrelevantAnswerException.class, maxAttempts = 3)
public String chat(@RequestParam("message") String message) {
    String answer = chatClient.prompt().user(message).call().content();
    checkRelevancy(message, answer);  // ! Runtime validation
    return answer;
}
```

Runtime Evaluation in App logic

eazy bytes

Relevancy Evaluation at Runtime

```
private void checkRelevancy(String message, String answer) {
    EvaluationRequest request = new EvaluationRequest(message, List.of(), answer);
    EvaluationResponse response = relevancyEvaluator.evaluate(request);

    if (!response.isPass()) {
        throw new IrrelevantAnswerException(message, answer);
    }
}
```

Retry with Spring Retry

```
@Retryable(retryFor = IrrelevantAnswerException.class, maxAttempts = 3)
```

If the LLM gives an irrelevant answer, retry up to 3 times. After retries are exhausted...

```
@Recover
public String recover(IrrelevantAnswerException e) {
    return "I'm sorry, I couldn't answer your question. Please try rephrasing it.";
}
```

eazy bytes

Monitoring AI Operations

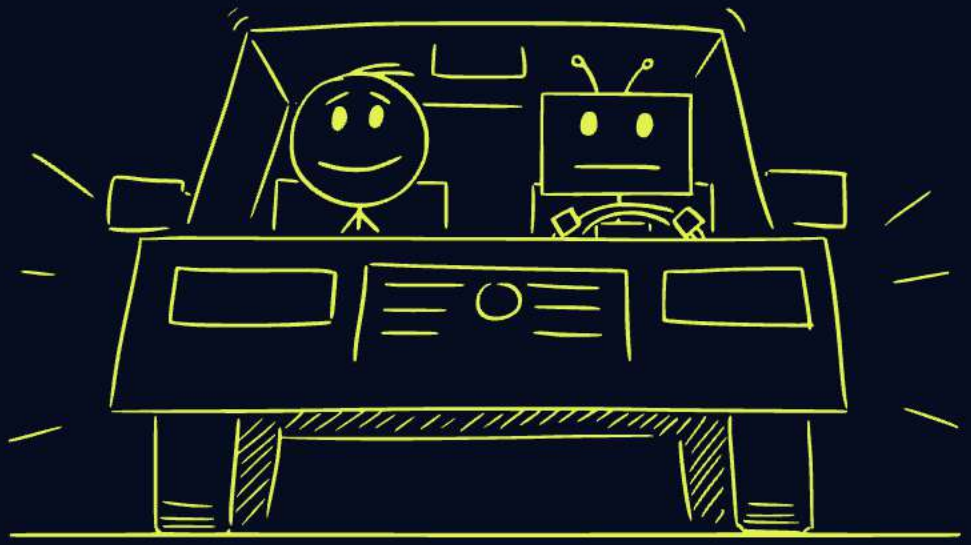
Observability: Your Application's Dashboard

eazy bytes

Imagine Your Application is like a Car...When you drive, you constantly monitor:

⛽ Fuel level
🌡️ Engine temperature
🚨 Warning lights
📊 Speed and RPM

Sometimes, you even go to a mechanic who plugs in diagnostic tools to check things you can't see from the dashboard — like engine codes, fuel injection issues, or battery performance.



The Same Principle Applies to Your Applications. Just like a car, your application has internal systems that needs to be monitored using :

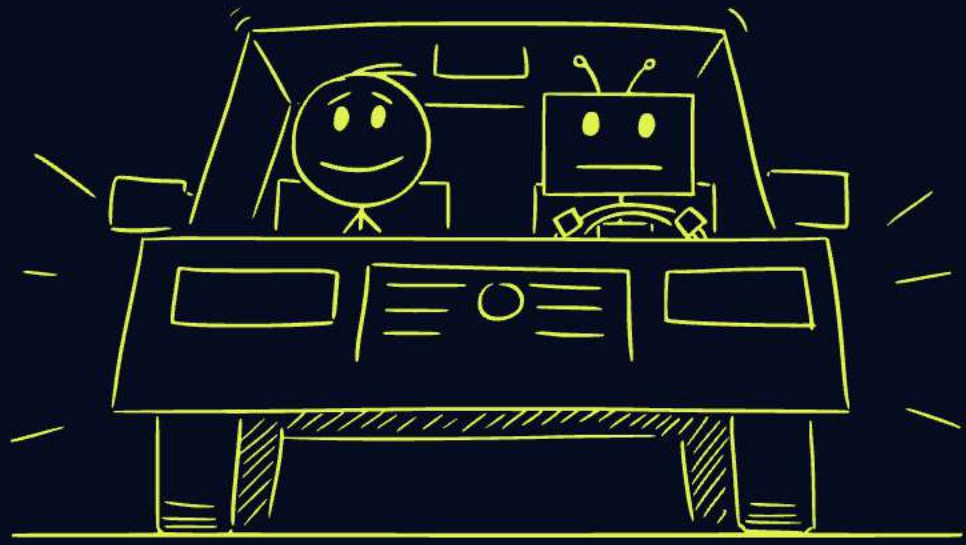
📈 Metrics (e.g., request counts, token usage)
🧩 Traces (to follow a request's journey through the system)
🧠 AI-specific insights (like model usage, latency, and failure rates)

Observability: Your Application's Dashboard

eazy bytes

🔍 What You'll Learn in this section

- How to enable observability in your Spring AI applications
- Understand the metrics Spring AI publishes
- Visualize those metrics using Prometheus and Grafana
- Dive deeper into request flows using tracing and spans
- Use this data to troubleshoot, optimize, and scale AI components



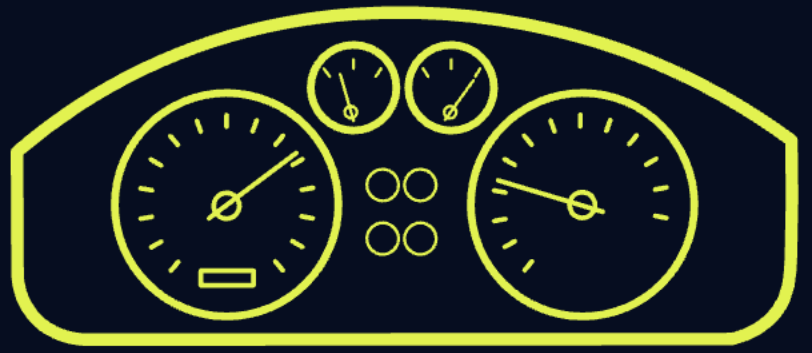
Enabling Metrics in Spring AI with Spring Boot Actuator

eazy bytes

Step 1: Add the Actuator Dependency

This brings in Spring Boot's Actuator module, which enables endpoints for monitoring and managing the app — like /health, /metrics, /env, etc.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```



Step 2: Configure Exposed Endpoints

By default, most actuator endpoints are not exposed. This line makes the /actuator/health and /actuator/metrics endpoints available over HTTP.

```
management.endpoints.web.exposure.include=health,metrics
```

Step 3: Access Metrics in Browser/Postman - This displays a list of all available metric names, such as: Spring AI metrics

```
http://localhost:8080/actuator/metrics
```

Enabling Prometheus Monitoring in Spring AI

eazy bytes

Step 1: Add Prometheus Registry Dependency

This allows Micrometer (used by Spring Boot) to export metrics in Prometheus format

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Step 2: Expose Prometheus Endpoint via Actuator

This makes /actuator/prometheus accessible, which Prometheus will scrape

```
management.endpoints.web.exposure.include=health,metrics,Prometheus
```


Enabling Prometheus Monitoring in Spring AI

eazy bytes

Step 3: Setup Prometheus with Docker

Launches Prometheus container and links it to your Spring AI app

```
prometheus:
  image: prom/prometheus
  container_name: prometheus
  volumes:
    - "./prometheus-config.yml:/etc/prometheus/prometheus.yml"
  networks:
    - spring-ai
  ports:
    - "9090:9090"
```


Enabling Prometheus Monitoring in Spring AI

eazy bytes

Step 4: prometheus-config.yml

Tells Prometheus to scrape metrics from your Spring AI app at /actuator/prometheus every 5 seconds.

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'spring-ai'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['host.docker.internal:8080’]
```


Step 5: Open Prometheus UI

Visit http://localhost:9090 and use the Prometheus UI to query metrics like `gen_ai_client_token_usage_total`

Setting Up Grafana Dashboard for Spring AI Metrics

eazy bytes

Step 1: Start Grafana (Docker)

Tells Prometheus to scrape metrics from your Spring AI app at /actuator/prometheus every 5 seconds.

```
grafana:
  image: grafana/grafana
  container_name: grafana
  ports:
    - "3000:3000"
  volumes:
    - grafana-storage:/var/lib/grafana
  networks:
    - spring-ai

volumes:
  grafana-storage:
```


Setting Up Grafana Dashboard for Spring AI Metrics

eazy bytes

Step 2: Open Grafana in Browser

Go to http://localhost:3000 and login

- Username: admin
- Password: admin

Step 3: Add Prometheus as Data Source

Click ☰ (hamburger menu) → "Connections" → "Add new connection"
Search Prometheus → Click on it
Set URL to:
http://prometheus:9090
Click Save & Test

Step 4: Create a Dashboard

Go to Dashboards → "+ Create dashboard"
Click "+ Add visualization"
Choose the Prometheus data source
In metric selector, type: `gen_ai_client_token_usage_total`
Click Run queries → See live time-series graph!

Tracing AI Operations

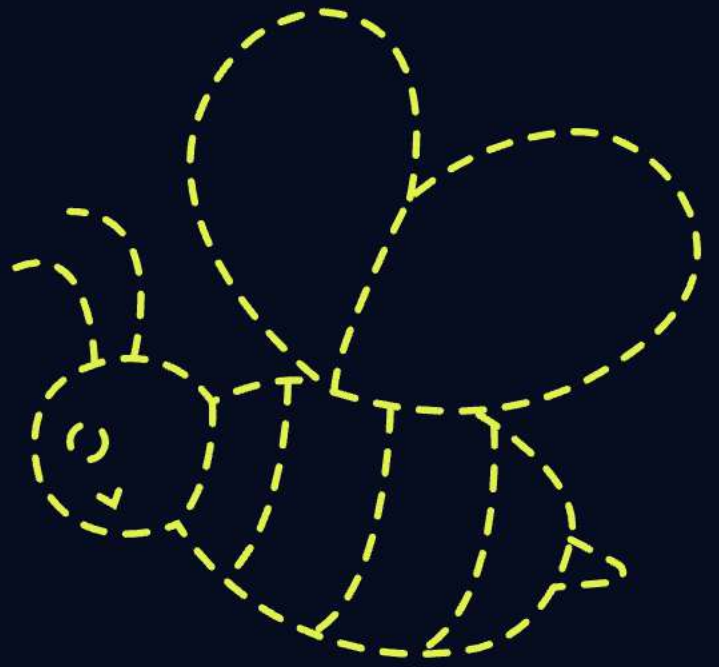
eazy bytes

Why Tracing?

Metrics tell you what happened. Tracing tells you where, when, and how it happened. Just like flight data recorders track an aircraft's journey, tracing tools track requests through your app — end-to-end.

Step 1: Add Tracing Dependencies

```
<!-- Bridge from Micrometer to OpenTelemetry -->
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>

<!-- OTLP exporter to send trace data -->
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
</dependency>
```

Tracing AI Operations

eazy bytes

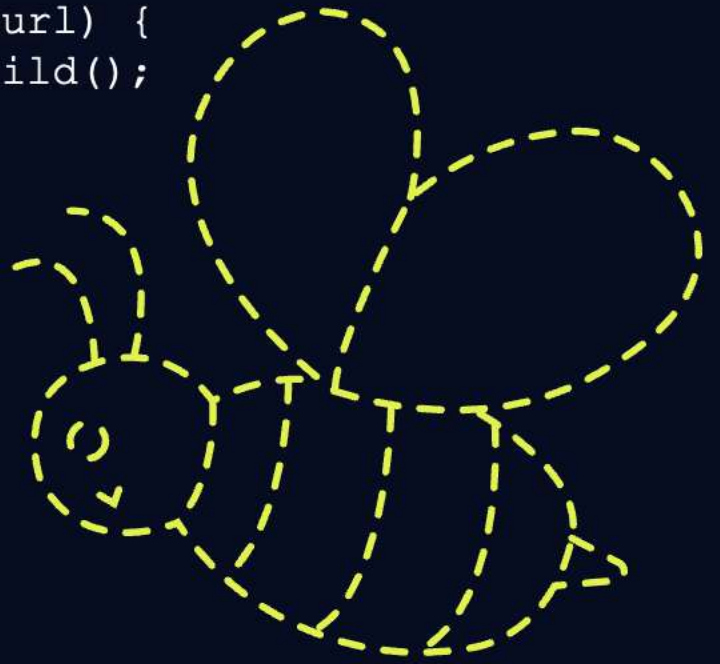
Step 2: Configure Exporter Bean

```
@Configuration
public class OpenTelemetryExporterConfig {

    @Bean
    public OtlpGrpcSpanExporter otlpExporter(
        @Value("${opentelemetry.exporter.otlp.endpoint}") String url) {
        return OtlpGrpcSpanExporter.builder().setEndpoint(url).build();
    }
}
```


Step 3: Add Tracing Properties

```
opentelemetry.exporter.otlp.endpoint=http://localhost:4317
management.tracing.sampling.probability=1.0
```

4317 is the port Jaeger listens on for trace data.
sampling.probability=1.0 means trace everything (in dev mode).

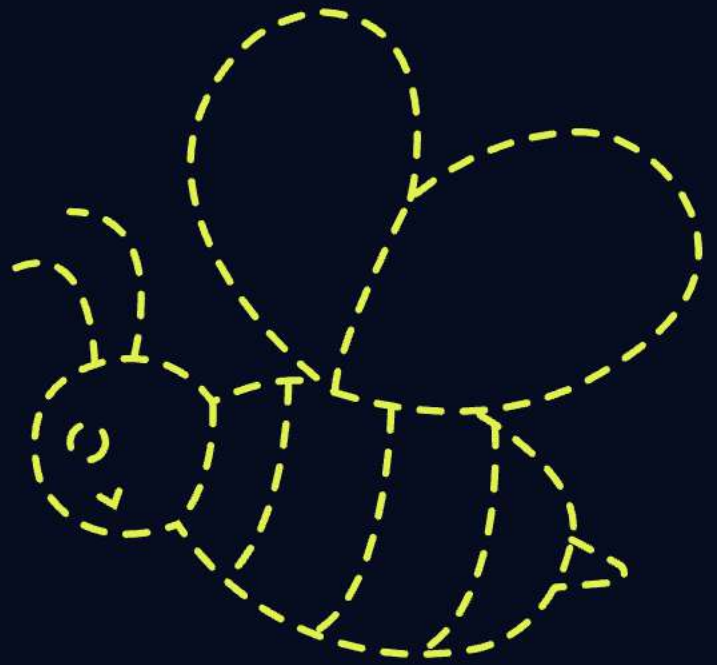


Tracing AI Operations

eazy bytes

Step 4: Run Jaeger with Docker

```
jaeger:
  image: jaegertracing/all-in-one:latest
  ports:
    - "16686:16686"    # Jaeger UI
    - "4317:4317"      # OTLP trace endpoint
  environment:
    - COLLECTOR_OTLP_ENABLED=true
  networks:
    - spring-ai
```

Step 5: View Traces in Jaeger

Run your app and call an endpoint (e.g. /ask)
Open http://localhost:16686 and Click "Find Traces"
🎯 Visualize AI request flows, bottlenecks, and timings

eazy bytes

Spring AI meets Speech

What Can Spring AI Do for Voice?

eazy bytes

Transcribe Speech -> Text

01

Transcription is the process by which text is produced from spoken audio.

- Use OpenAiAudioTranscriptionModel when OpenAI used
- Feed it an audio file (MP3/WAV)
- Get back text like magic

Text -> Speech

02

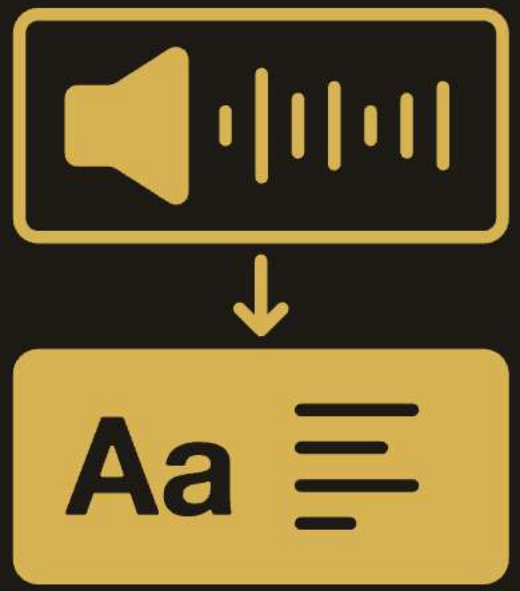
Various styles of Voices/Speeches can be generated from text

- Use SpeechModel
- Give it a message
- Get a talking byte array 🔊

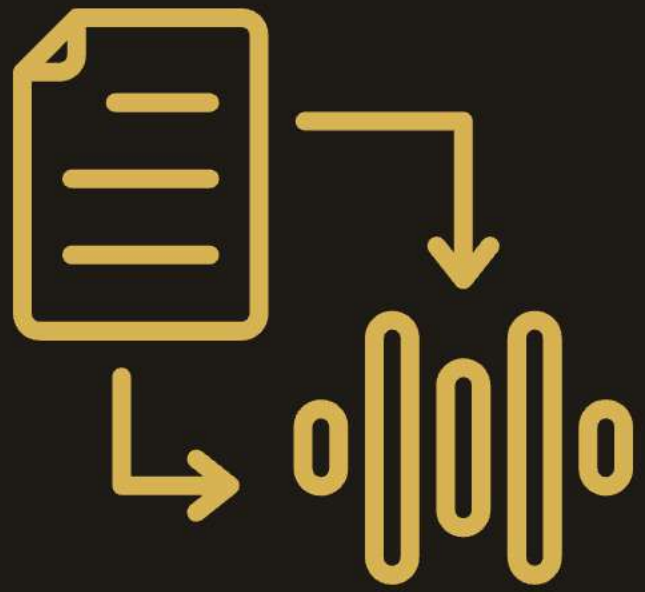
Code Sample for Transcribe

eazy bytes

```
@RestController
@RequestMapping("/api")
public class AudioController {

    private final OpenAiAudioTranscriptionModel transcriptionModel;

    AudioController(OpenAiAudioTranscriptionModel transcriptionModel) {
        this.transcriptionModel = transcriptionModel;
    }


    @GetMapping("/transcribe-options")
    String transcribeOptions(@Value("classpath:SpringAI.mp3") Resource audioFile) {
        var transcriptionResponse = transcriptionModel.call(
                new AudioTranscriptionPrompt(audioFile,
                        OpenAiAudioTranscriptionOptions.builder()
                .language("en")
                .prompt("Talking about Spring AI")
                .temperature(0f)
                .responseFormat(OpenAiAudioApi.TranscriptResponseFormat.VTT)
                .build()));
        return transcriptionResponse.getResult().getOutput();
    }

}
```

Code Sample for Text to Speech



eazy bytes

```
@RestController
@RequestMapping("/api")
public class AudioController {

    private final SpeechModel speechModel;

    AudioController(SpeechModel speechModel) {
        this.speechModel = speechModel;
    }

    @GetMapping("/speech-options")
    String speechProviderOptions(@RequestParam("message") String message) throws IOException {
        var speechResponse = speechModel.call(new SpeechPrompt(message,
                OpenAiAudioSpeechOptions.builder()
                .model("tts-1")
                .voice(OpenAiAudioApi.SpeechRequest.Voice.NOVA)
                .responseFormat(OpenAiAudioApi.SpeechRequest.AudioResponseFormat.MP3)
                .speed(2.0f)
                .build()));
        byte[] audioBytes = speechResponse.getResult().getOutput();
        Path path = Paths.get("speech-options.mp3");
        Files.write(path, audioBytes);
        return "MP3 saved successfully to " + path.toAbsolutePath();
    }


}
```

eazy bytes

Image generation using Spring AI

Code Sample for image generation

eazy bytes

```
@RestController
@RequestMapping("/api")
public class ImageController {

    private final ImageModel imageModel;

    ImageController(ImageModel imageModel) {
        this.imageModel = imageModel;
    }

    @GetMapping("/image-options")
    String generateImageWithOptions(@RequestParam("message") String message) {
        var imageResponse = imageModel.call(new ImagePrompt(message,
                OpenAiImageOptions.builder()
                        .quality("hd")
                        .height(1024)
                        .width(1024)
                        .style("natural")
                        .responseFormat("url")
                        .build()));
        return imageResponse.getResult().getOutput().getUrl();
    }

}
```



Congratulations
Thank You, See you next time !!