Inteligência Artificial 2

Aula 21: Aprofundamento no Spring AI

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Inteligência Artificial 2

Sumário

1. Arquitetura e Fundamentos
2. Programação Reativa e Streaming
3. Formatadores Estruturados
4. LLMs como Agentes (Tool Calling)
5. Integração Local e Observabilidade

Onde Paramos?

Na aula anterior, completamos a **integração Full-Stack**:

- CORS configurado no Spring Boot (`@CrossOrigin`).
- Front-end HTML+JS consumindo a API via `fetch()`.
- Indicador "Pensando..." para feedback visual ao usuário.

A Pergunta de Hoje

O sistema funciona, mas **é frágil**. Se a API cair,
o usuário vê um erro 500. Como torná-lo **resiliente**?

Objetivos da Sessão

Ao final desta aula, você será capaz de:

1. **Implementar** `@Retryable` com Exponential Backoff no Spring.
2. **Criar** métodos `@Recover` para fallback amigável.
3. **Separar** Controller $\rightarrow$ Service (arquitetura MVC limpa).
4. **Simular** falhas e validar a resiliência nos logs.
5. **Justificar** por que resiliência é obrigatória em produção.

Recapitulando o Acoplamento de IA

- Consumir IA generativa por meio de APIs *REST* puras (RestTemplate ou fetch) cria um vínculo forte (*Vendor Lock-in*) com o formato *JSON* do provedor.
- O **Spring AI** nasceu para isolar essa camada de infraestrutura, permitindo que desenvolvedores Java tratem os LLMs como um "banco de dados cognitivo" padronizado.
- Uma vez que a interface é padronizada, podemos aplicar conceitos avançados de Engenharia de Software, como Programação Reativa, Chamada de Funções (*Tool Calling*) e Tipagem de Saída de forma agnóstica.

A Arquitetura Modular do Spring AI

O *framework* é dividido em bibliotecas distintas (*Starters*) para evitar o inchaço do pacote final (*Fat JAR*).

- **Spring AI Core:** Contém as abstrações principais (ChatClient, Document, VectorStore).
- **Model Starters:** Integrações específicas com LLMs (spring-ai-openai, spring-ai-anthropic, spring-ai-ollama).
- **Vector Store Starters:** Adaptadores para bancos de dados (spring-ai-pgvector, spring-ai-chroma).
- O Desenvolvedor adiciona apenas as peças de *Lego* necessárias.

ChatClient vs ChatModel

ChatModel (A API de Baixo Nível)

- Recebe listas brutas de Message.
- Útil para desenvolvedores que estão criando *frameworks* em cima do Spring AI ou precisando manipular os *HTTP Headers* da requisição.

ChatClient (A API de Alto Nível)

- Baseado no padrão *Fluent Builder*.
- Ideal para lógica de negócio corporativa, pois oculta a complexidade e permite injeção nativa de *Advisors* (como o RAG).

Configuração Avançada de Modelos (Options)

Podemos calibrar o determinismo do modelo diretamente no código, sobressaindo as configurações globais.

```
ChatResponse response = chatModel.call(
    new Prompt(
        "Faca um resumo do documento.",
        OpenAiChatOptions.builder()
            .withModel("gpt-4o")
            .withTemperature(0.1) // Determinismo extremo
            .withMaxTokens(500)
            .build()
    )
);
```

O Gargalo da Requisição Síncrona

- Em microsserviços *WebMVC* tradicionais, a *Thread* do servidor fica bloqueada esperando o LLM terminar de gerar todos os tokens (5 a 20 segundos).
- Com milhares de usuários simultâneos, o esgotamento do *Thread Pool* do *Tomcat* causa negação de serviço.
- A Matemática da Geração Sequencial (Autorregressiva) nos permite receber "pedaços" de resposta enquanto a rede neural ainda está computando a próxima palavra.

Programação Reativa (Spring WebFlux)

- A Programação Reativa baseia-se em fluxos de dados assíncronos não-blocantes (*Event-loop*), sustentando milhares de conexões com pouquíssimas *Threads*.
- O Spring AI integra-se nativamente com o *Project Reactor*, utilizando a classe **Flux¡T¿**.
- Em vez de retornar uma *String* massiva no final, a API devolve um Flux¡String¿, um túnel pelo qual palavras pingam uma a uma (*Server-Sent Events* - *SSE*).

Geração de Texto em Streaming (Flux)

```
@GetMapping(value = "/chat/stream",
            produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<String> streamChat(@RequestParam String mensagem) {

    // Note a chamada para .stream() no lugar de .call()
    return chatClient.prompt()
            .user(mensagem)
            .stream()
            .content();
}
```

Este é o segredo arquitetural para criar uma interface que digita texto em tempo real (como o ChatGPT nativo) ao invés de exibir uma barra de carregamento.

Output Parsers (Além do Texto Bruto)

- Se o *backend* Java está orquestrando a IA, ele raramente quer conversar. Ele quer Tipos Sólidos (Int, Arrays, Objetos).
- O OutputParser do Spring AI resolve a extração probabilística usando técnicas de indução.
- Ele injeta instruções ocultas de formatação no *System Prompt* e intercepta a resposta, "deserializando" o JSON para classes Java (*POJOs* ou *Records*).

O BeanOutputConverter: Por Baixo dos Panos

```
record Filme(String titulo, String diretor, int ano) {}

var converter = new BeanOutputConverter<>(Filme.class);

String resposta = chatClient.prompt()
    .user("Qual o filme mais famoso de Ficcao Cientifica?")
    // Injeta a obrigatoriedade do formato JSON do Record "Filme"
    .system(converter.getFormat())
    .call()
    .content();

Filme filmeAnalisado = converter.convert(resposta);
// Agora voce tem um objeto Java forte e tipado!
```

O Paradoxo da Incapacidade de Ação

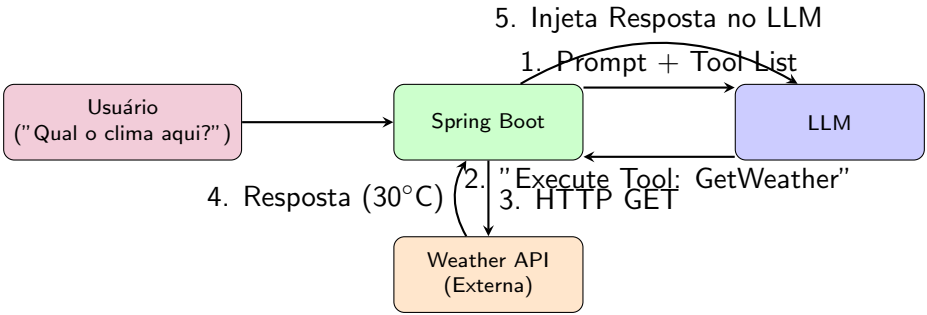
Um modelo LLM puramente generativo é uma "caixa isolada".

- Ele não consegue consultar a cotação do Dólar agora (pois só foi treinado com dados até o ano passado).
- Ele não consegue executar um comando SQL para saber se o cliente "Alessio" tem saldo na conta bancária corporativa.
- A revolução dos *Agentes Autônomos* ocorre com o **Tool Calling (Function Calling)**.

Tool Calling: O Conceito Matemático

- No *Tool Calling*, nós enviamos ao LLM, junto com a pergunta, um *JSON Schema* documentando funções de código que o Servidor Java possui (ex: getSaldo(String cpf)).
- Durante o passo autorregressivo, as camadas de Atenção do LLM avaliam se a pergunta do usuário requer o uso de uma dessas ferramentas.
- Se sim, o modelo **interrompe a geração de texto** e devolve um pacote JSON especial mandando o sistema executar a função getSaldo com os parâmetros matematicamente inferidos a partir da pergunta do cliente.

LLM como Orquestrador de Fluxos



Declarando Funções em Java (@Bean)

O Spring AI transforma métodos Java em ferramentas para LLMs utilizando a anotação @Description.

```
@Configuration
public class FerramentasClima {

    // A descricao guia as Atencoes do modelo para saber QUANDO usar.
    @Bean
    @Description("Retorna a temperatura atual em uma cidade especifica")
    public Function<RequestClima, ResponseClima> climaAtual() {
        return request -> new ResponseClima(30.5, "Ensolarado");
    }
}
```

O *framework* varre essa classe em tempo de compilação, extrai o tipo de dado de entrada e formata o *Schema JSON* necessário.

Invocando o Agente Dinâmico

```
String resposta = chatClient.prompt()
    .user("Deveria levar guarda-chuva para Timoteo hoje?")
    // Injeta a autorizacao para a IA utilizar a funcao Java
    .functions("climaAtual")
    .call()
    .content();
```

Sem escrever nenhum "If-Else", o modelo decidirá invocar a API de clima, ler a temperatura "30.5", deduzir que não choverá e formular a string de resposta dizendo para não levar guarda-chuva.

Integração Local: Spring AI + Ollama

- Sistemas fechados (*OpenAI/Gemini*) exigem enviar dados corporativos sigilosos pela Internet para servidores estrangeiros (Quebra de *Compliance* bancário).
- O Projeto **Ollama** permite rodar modelos de peso aberto (ex: *Llama-3 8B*) inteiramente dentro do *hardware* do servidor (ou contêiner) da empresa.
- Alterando a dependência para spring-ai-ollama, o código Java passa a consultar a porta localhost:11434. Nenhum dado do cliente deixa o *Data Center*.

Vantagens da Infraestrutura Local (Ollama)

1. **Privacidade Absoluta (LGPD):** Contratos e laudos médicos podem ser processados com RAG sem risco de vazamento para Big Techs.
2. **Custos Previsíveis:** Ausência de cobrança por *Tokens*. O custo é puramente operacional (energia elétrica e amortização das GPUs H100).
3. **Desvantagem:** O custo do servidor inicial (CAPEX) é altíssimo e a latência de inferência é significativamente inferior à dos *clusters* otimizados do Google.

Padrões de Observabilidade (Micrometer)

Num cenário de microsserviços, se uma resposta de IA demorar 10 segundos, onde está o gargalo? Na busca do banco vetorial ou no LLM da nuvem?

- O Spring Boot incorpora observabilidade robusta por meio do **Micrometer**.
- O Spring AI nativamente emite métricas e rastros dinâmicos (*Distributed Tracing*) para servidores como Prometheus e Grafana.
- O time de Engenharia de Confiabilidade (SRE) consegue auditar os painéis para visualizar exatamente quantos milissegundos o *VectorStore* gastou para retornar os documentos e quantos o ChatGPT gastou para resumir.

Conclusão da Aula 21

- O *Spring AI* extrapola a simples comunicação textual, transformando os *Back-ends* em *Hubs Cognitivos*.
- A capacidade de modelar saídas estruturadas em *Records* resolve o maior obstáculo da adoção corporativa: o caos do formato de texto livre.
- O **Tool Calling** altera radicalmente a arquitetura de software: a IA passa de um oráculo passivo para um **Agente Ativo**, orquestrando execuções de métodos, *queries* SQL e navegação baseada no raciocínio da rede neural.
- Na Aula 22, condensaremos todas as primitivas vistas (RAG, MVC, Tipagem e Configurações) em um projeto unificado de Arquitetura Limpa.

Dúvidas?

Email: alessio@cefetmg.br
Aula: Segunda e Quarta-Feira, 7h as 8:40h (Sala 39)
WhatsApp: Dúvidas rápidas

Lembrete

Confira sua Issue no GitLab!

Próxima aula: **Entrega do TP2**

100% da Aula 21 Concluída!