



ICT Training Center

Il tuo partner per la Formazione e la Trasformazione digitale della tua azienda



SPRING AI




GENERATIVE ARTIFICIAL INTELLIGENCE CON JAVA

Simone Scannapieco

Corso avanzato per Venis S.p.A, Venezia, Italia

Novembre 2025

TOOL CALLING

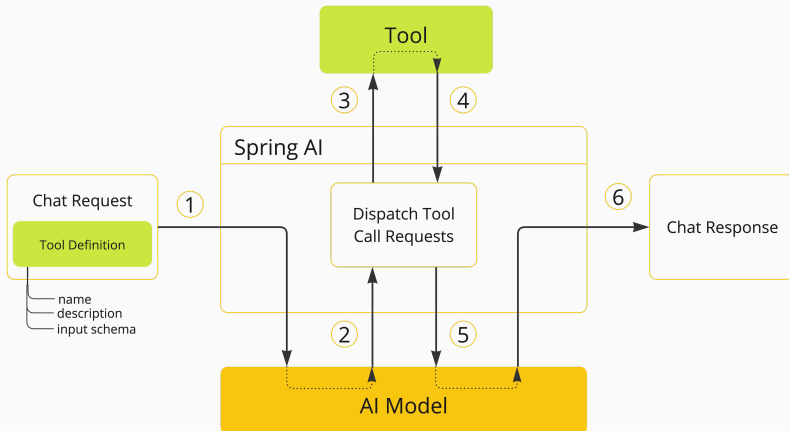
- ➔ *Design pattern* in AI (chiamata anche *function calling*)
- ➔ Tecnica di mitigazione della *knowledge cut-off*...
- ➔ ...ma non solo
 - ➔ Ricerca di informazioni non presenti nella sua base di conoscenza
 - ➔ "Che tempo fa a Venezia?" ~> "Soleggiato con temperature massime sui 20°C"
 - ⚠ Quindi, differenza con RAG *web search*...?!
 - ➔ Logica più semplice e mirata per recupero delle informazioni
 - ➔ Messa in atto di ciò che comprende
 - ➔ "Prenota un biglietto" ~> 
 - ➔ "Invia un'email" ~> 
 - ➔ "Fissa un appuntamento per martedì prossimo alle 18 in ufficio" ~> 
 - ➔ Primo passaggio nella transizione da LLM a **ALM** (Action Language Model)

RAG	<i>Tool calling</i>
Leggere un manuale fai-da-te per risolvere da solo	Chiamare un idraulico per riparare una perdita
“In base a quello che mi hai chiesto e avendo letto alcuni documenti, ti spiego”	“In base a quello che mi hai chiesto, ecco come far agire il sistema con i mezzi a disposizione”
Si concentra sulla generazione delle risposte utilizzando contenuti recuperati	Esegue azioni dal vivo o recupera dati dal vivo
Recupera solo documenti e continua a generare la risposta	Richiede all' <i>app client</i> di eseguire il <i>tool</i>

TOOL CALLING

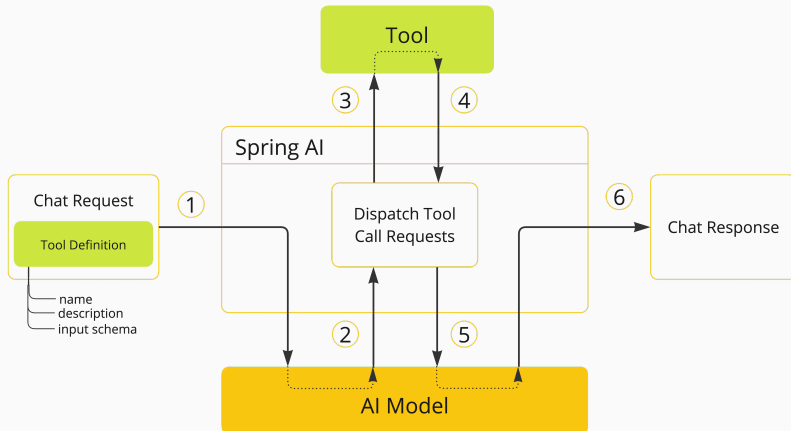
WORKFLOW IN SPRING AI - DEFAULT

- 1 L'utente invia la richiesta al LLM attraverso `ChatClient/ChatModel` riferendo al LLM i *tool* a sua disposizione



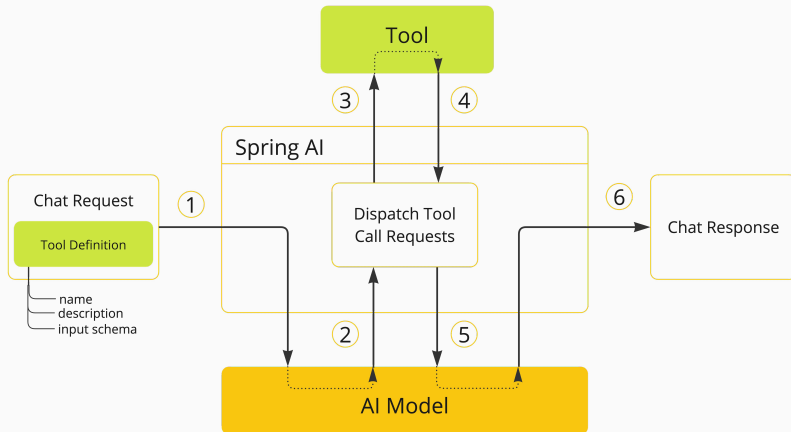
©Spring AI

- 2 Se il LLM decide di chiamare un *tool*, invia una *response* con nome del *tool* e parametri istanziati, seguendo lo schema definito per il *tool*



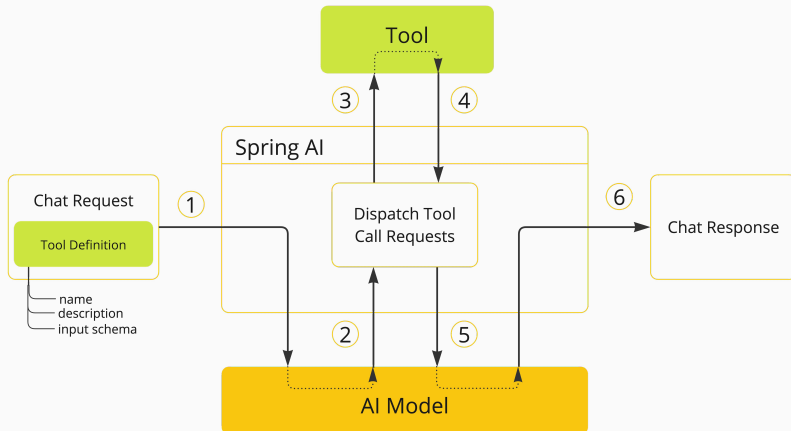
©Spring AI

- 3 L'applicativo inoltra all'effettiva implementazione del *tool* nel *container* Spring la richiesta con istanziazione dei parametri suggeriti dal LLM



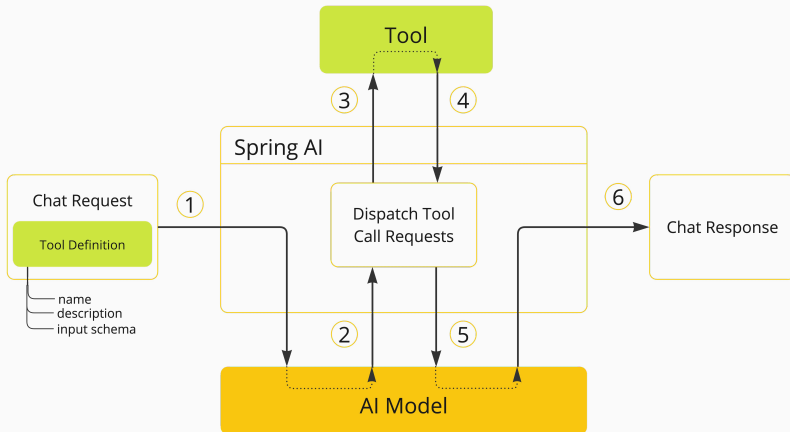
©Spring AI

- 4 Il *tool* esegue la sua logica di implementazione con i parametri restituiti dal LLM e restituisce il risultato al *container* Spring



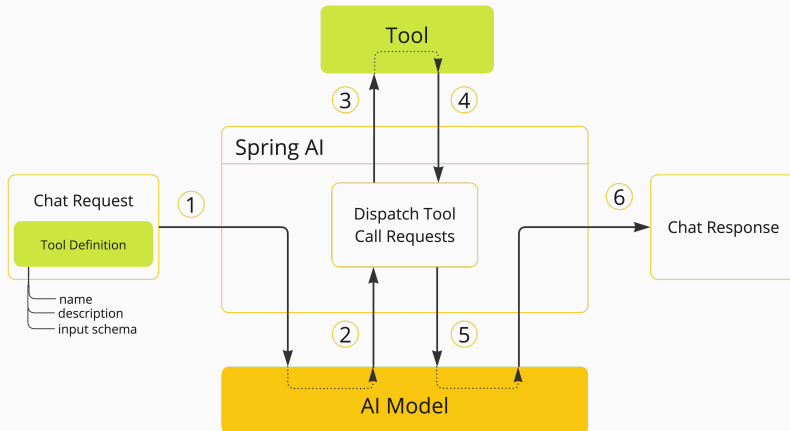
©Spring AI

- 5 Il sistema invoca nuovamente il LLM con la richiesta utente precedente ma ulteriormente contestualizzata dall'*output* del *tool*



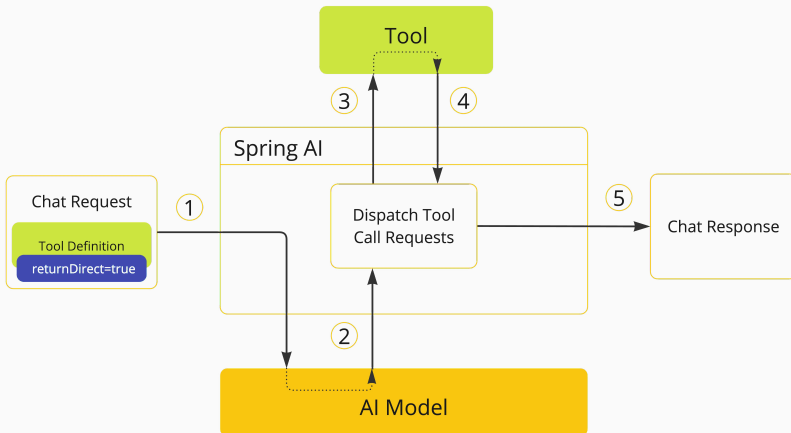
©Spring AI

- 6 Il LLM restituisce la risposta finale al ChatClient/ChatModel sfruttando generazione di testo e contesto



©Spring AI

- 5 Il sistema restituisce il risultato del *tool* direttamente a `ChatClient/ChatModel` senza interrogare nuovamente il LLM



©Spring AI

- ➔ Metodologie
 - ➔ Dichiarativa
 - ➔ Programmatica
- ➔ Paradigmi
 - ➔ *Method As Tool (MAT)*
 - ➔ *Function As Tool (FAT)*

TOOL CALLING

METHOD AS TOOL

Classe di definizione dei *tool*

```
@Component
public class TimeTools {

    @Tool(name="getCurrentLocalTime",
        description="Ottieni l'ora corrente nel fuso orario dell'utente.")
    String getCurrentLocalTime() {
        ...
    }

    @Tool(name="getCurrentTime",
        description="Ottieni l'ora corrente nel fuso orario specificato.",
        returnDirect=true)
    public String getCurrentTime(@ToolParam(description = "Valore che rappresenta il fuso orario.") String timeZone) {
        ...
    }
}
```

- ➔ Istanziati come `@Component`
- ➔ Ogni metodo a disposizione del LLM come *tool* deve essere annotato come `@Tool`
 - ➔ se `name` non specificato, Spring AI popola il parametro con il nome del metodo
 - ⚠ `name` come **identificativo univoco** per Spring AI
 - ➔ `description` fornisce il **contesto** al LLM per determinare se utilizzare il *tool* in base alla richiesta utente!
 - ➔ `@ToolParam` fornisce ulteriore contesto al LLM per iniettare parametri al metodo
 - ➔ `returnDirect` per sovrascrivere il comportamento di *default* se necessario

Classe di definizione dei tool

```
@Component
public class TimeTools {

    @Tool(name="getCurrentLocalTime",
        description="Ottieni l'ora corrente nel fuso orario dell'utente.")
    String getCurrentLocalTime() {
        ...
    }

    @Tool(name="getCurrentTime",
        description="Ottieni l'ora corrente nel fuso orario specificato.",
        returnDirect=true)
    public String getCurrentTime(@ToolParam(description = "Valore che rappresenta il fuso orario.") String timeZone) {
        ...
    }
}
```

👤 "D: Che ore sono a New York?"

>_ (thinking) — Scansiono i tool a disposizione...—

>_ (thinking) — Vedo che ho dei tool che gestiscono il recupero dell'ora...—

>_ (thinking) — Il primo tool è relativo alla posizione dell'utente...—

>_ (thinking) — Il secondo tool è più astratto e determina la posizione attraverso parametrizzazione del fuso orario...—

>_ (thinking) — ...Il fuso orario di New York è America/New York...—

>_ "R: Utilizza il tool `getCurrentTime('America/New York')`."

Processo di method as tool

Tool-aware ChatClient - approccio statico

```
@Bean
public ChatClient ollamaTimeToolsChatClient(OllamaChatModel ollamaChatModel, TimeTools timeTools) {

    ChatClient.Builder chatClientBuilder = ChatClient.builder(ollamaChatModel);

    return chatClientBuilder
        .defaultTools(timeTools)
        ...
        .build();

}
```

Tool-aware ChatClient - approccio dinamico

```
@Override
public Answer getOllamaTimeToolsLocalTimeAnswer(QuestionRequest request) {

    return new Answer(this.ollamaTimeToolsChatClient
        .prompt()
        .tools(timeTools)
        ...
        .call()
        .content());

}
```

Classe di definizione dei *tool*

```
public class TimeTools {  
  
    String getCurrentLocalTime() {  
        ...  
    }  
  
    public String getCurrentTime(  
        @ToolParam(description = "Valore che rappresenta il fuso orario.") String timeZone) {  
        ...  
    }  
}
```

➡ `@ToolParam` permesso in approccio sia dichiarativo che programmatico

Utilizzo approccio *method-tool callback*

```
import java.lang.reflect.Method;

Method method = MethodToolCallbackReflectionUtils.findMethod(TimeTools.class, "getCurrentLocalTime");
ToolCallback toolCallback = MethodToolCallback.builder()
    .toolDefinition(ToolDefinitions.builder(method)
        .name("getCurrentLocalTime")
        .description("Ottieni l'ora corrente nel fuso orario dell'utente.")
        .build())
    .toolMethod(method)
    .toolObject(new TimeTools())
    .toolMetadata(ToolMetadata.builder()
        .returnDirect(true)
        .build())
    .build();
```

- ➔ Il sistema usa *reflection* per linkare il metodo
- ➔ Costruito un **ToolCallback** che definisce la logica di utilizzo del *tool*
 - ➔ **ToolDefinitions.Builder** permette di definire nome, descrizione e schema del *tool*
 - ➔ **ToolMetadata.Builder** responsabile della definizione strategia *default* vs *direct*
- ⚠ Definizione **ToolCallback** *under-the-hood* anche con approccio dichiarativo

Tool-aware ChatClient - approccio statico

```
@Bean
public ChatClient ollamaTimeToolsChatClient(OllamaChatModel ollamaChatModel) {

    ChatClient.Builder chatClientBuilder = ChatClient.builder(ollamaChatModel);

    return chatClientBuilder
        .defaultToolCallbacks(toolCallback)
        ...
        .build();

}
```

Tool-aware ChatClient - approccio dinamico

```
@Override
public Answer getOllamaTimeToolsLocalTimeAnswer(QuestionRequest request) {

    return new Answer(this.ollamaTimeToolsChatClient
        .prompt()
        .toolCallbacks(toolCallback)
        ...
        .call()
        .content());

}
```

- ➔ **Restrizioni di tipo** per parametri e valori di ritorno
- ⚠ **Tipi non supportati**
 - ➔ **Tipi Optional** - `Optional<T>` non è compatibile
 - ➔ **Costrutti asincroni** - `CompletableFuture`, `Future`
 - ➔ **Tipi reattivi** - `Flow`, `Mono`, `Flux`
 - ➔ **Tipi funzionali** - `Function`, `Supplier`, `Consumer`
- ⚠ I tipi funzionali sono supportati attraverso l'approccio **function-based** dedicato

TOOL CALLING

FUNCTION AS TOOL

Classe di definizione dei tool

```
@Configuration(proxyBeanMethods = false)
class WeatherTools {

    public static final String GET_TEMP_IN_LOCATION_FUNCTION_NAME = "getTemperatureInLocation";

    @Bean(GET_TEMP_IN_LOCATION_FUNCTION_NAME)
    @Description("Ottieni la temperatura corrente nella località specificata.")
    Function<WeatherRequest, TemperatureResponse> currentTemperature() {
        return new TemperatureService();
    }
}

public enum Unit { C, F }

public record WeatherRequest(
    @ToolParam(description = "Il nome di una città o di una nazione.") String location, Unit unit) {}

public record TemperatureResponse(double temp, Unit unit) {}
```

- ⚠ **Tipi funzionali supportati:** `Function<I,O>`, `Supplier<O>`, `Consumer<I>`, `BiFunction<I1,I2,O>`
- ➡ Ciascun funzionale disponibile come *tool* attraverso definizione `@Bean`
- ⚠ Definire nome come **costante esportabile** per mancanza di garanzia di *type safety*
- ➡ `@Description` come *description* di MAT
- ➡ `@ToolParam` con medesima logica di MAT

Tool-aware ChatClient - approccio statico

```
@Bean
public ChatClient ollamaWeatherToolsChatClient(OllamaChatModel ollamaChatModel) {

    ChatClient.Builder chatClientBuilder = ChatClient.builder(ollamaChatModel);

    return chatClientBuilder
        .defaultToolNames(WeatherTools.GET_TEMP_IN_LOCATION_FUNCTION_NAME)
        ...
        .build();
}
```

Tool-aware ChatClient - approccio dinamico

```
@Override
public TemperatureResponse getOllamaTemperatureToolAnswer(@RequestBody WeatherRequest request) {

    return this.ollamaWeatherToolsChatClient
        .toolNames(WeatherTools.GET_TEMP_IN_LOCATION_FUNCTION_NAME)
        ...
        .call()
        .content();
}
```


Classe di definizione dei funzionali

```
public class TemperatureService implements Function<WeatherRequest, TemperatureResponse> {  
    public WeatherResponse apply(WeatherRequest request) {  
        ...  
    }  
}  
  
public enum Unit { C, F }  
  
public record WeatherRequest(  
    @ToolParam(description = "Il nome di una città o di una nazione.") String location, Unit unit) {}  
  
public record TemperatureResponse(double temp, Unit unit) {}
```

- ➔ Richiesta sola la logica pertinente al funzionale scelto (es. apply per Function)

Utilizzo approccio *function-tool callback*

```
public static final String GET_TEMP_IN_LOCATION_FUNCTION_NAME = "getTemperatureInLocation";

ToolCallback toolCallback = FunctionToolCallback
    .builder(GET_TEMP_IN_LOCATION_FUNCTION_NAME, new TemperatureService())
    .description("Ottieni la temperatura corrente nella località specificata.")
    .inputType(WeatherRequest.class)
    .toolMetadata(ToolMetadata.builder()
        .returnDirect(true)
        .build())
    .build();
```

- ➔ `FunctionToolCallback.Builder` linka un identificativo testuale ad una istanza di una nuova *function tool*
- ➔ Costruito un `ToolCallback` che definisce la logica di utilizzo del *tool*
 - ➔ `ToolDefinitions.Builder` permette di definire nome, descrizione e schema del *tool*
 - ➔ `ToolMetadata.Builder` responsabile della definizione strategia *default* vs *direct*
- ⚠ Definizione `ToolCallback` *under-the-hood* anche con approccio dichiarativo

Tool-aware ChatClient - approccio statico

```
@Bean
public ChatClient ollamaWeatherToolsChatClient(OllamaChatModel ollamaChatModel) {

    ChatClient.Builder chatClientBuilder = ChatClient.builder(ollamaChatModel);

    return chatClientBuilder
        .defaultToolCallbacks(toolCallback)
        ...
        .build();
}
```

Tool-aware ChatClient - approccio dinamico

```
@Override
public TemperatureResponse getOllamaTemperatureToolAnswer(@RequestBody WeatherRequest request) {

    return this.ollamaWeatherToolsChatClient
        .toolCallbacks(toolCallback)
        ...
        .call()
        .content();
}
```