dgroove
Tecnologia emozionale.

ICT Training Center

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

Note

Spring AI

Generative Artificial Intelligence con Java

Simone Scannapieco

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

Novembre 2025

Note

Tool Calling

Note

- Tecnica di mitigazione della *knowledge cut-off*…
- …ma non solo
  - Ricerca di informazioni non presenti nella sua base di conoscenza (es. “Che tempo fa a Venezia?”)
  - ⚠ Quindi, differenza con RAG *web search*...?!
  - Messa in atto di ciò che comprende
    - “Prenota un biglietto”
    - “Invia un’email”
    - “Crea un nuovo *ticket* di segnalazione guasto”
  - Primo passaggio nella transizione da LLM a ALM (Action Language Model)

Note

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

Note

Tool Calling

Method as tool

Note

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

- Solitamente 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

Note

Classe di definizione dei *tool*

```java
@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.")
    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*

Note

`ChatClient` potenziato con *tool*

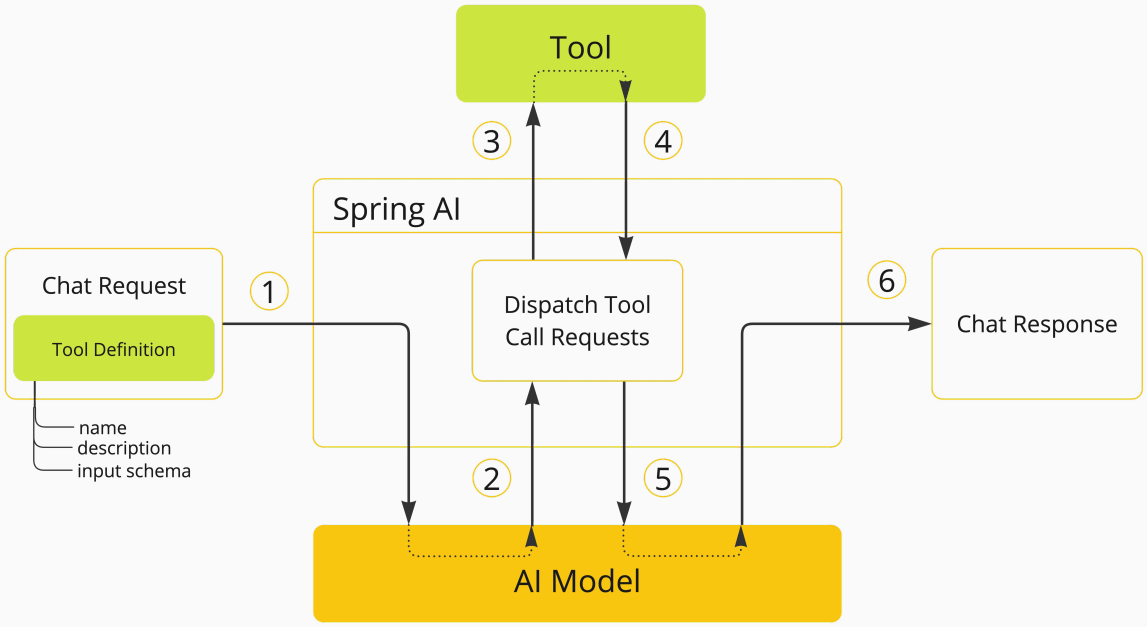
```
@Bean
public ChatClient ollamaChatClient(OllamaChatModel ollamaChatModel, TimeTools timeTools) {

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

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

}
```

- Disponibilità dei *tool* al `ChatClient` attraverso *injection*

Note

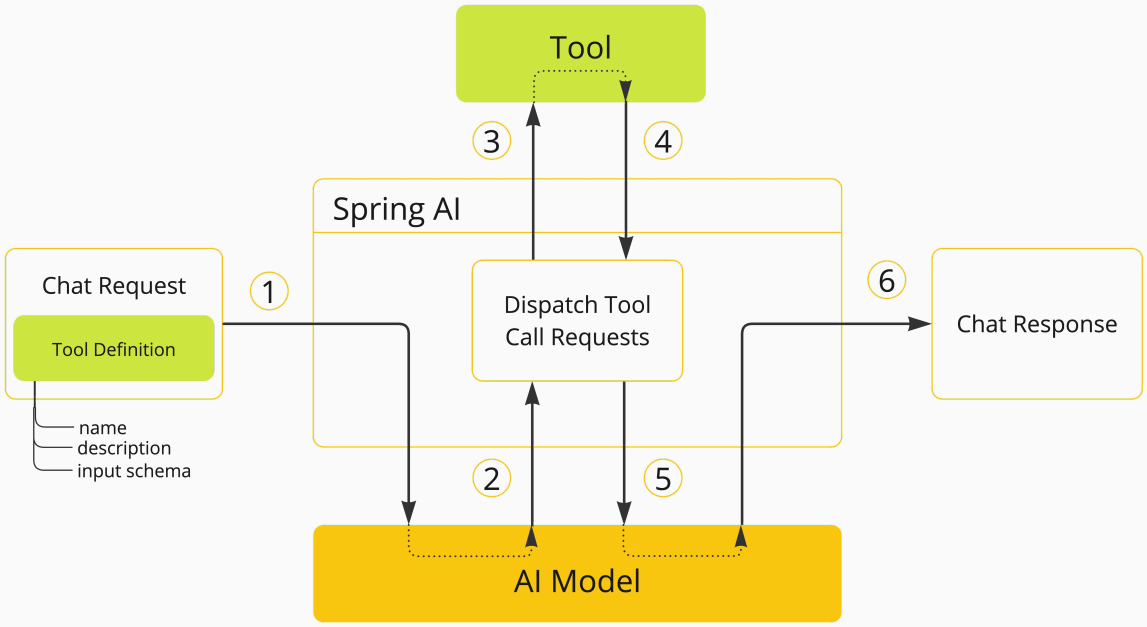
TOOL CALLING
WORKFLOW

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



Note

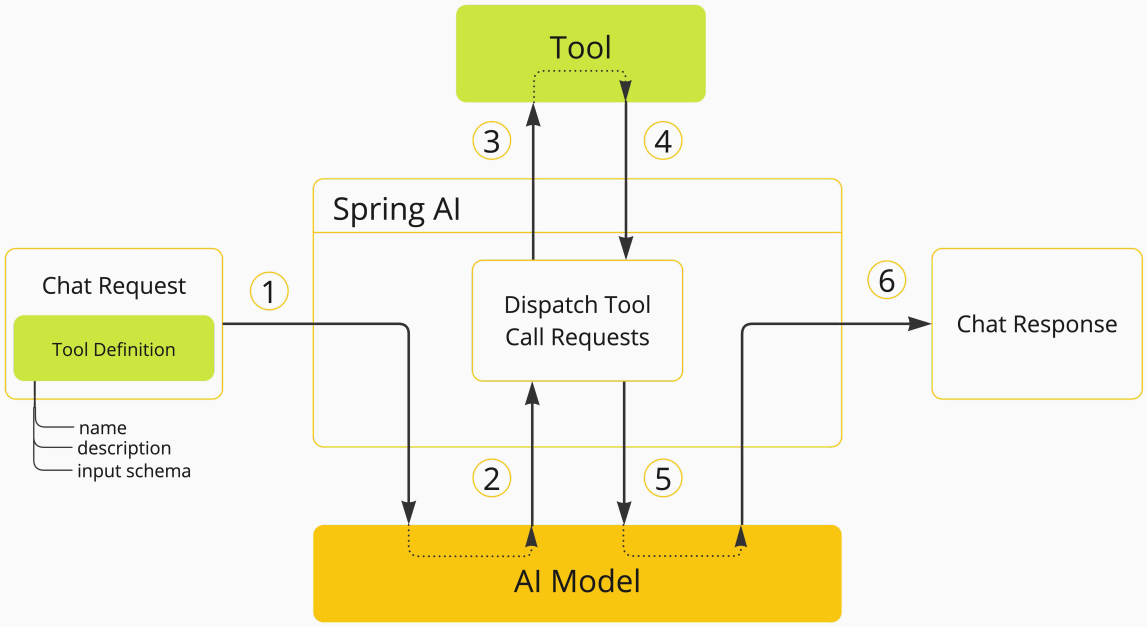
2 Se il LLM decide di chiamare un *tool*, invia una *response* con nome del *tool* e parametri istanziati, seguendo la schema definito da `@Tool` e `@ToolParam`



©Spring AI

Note

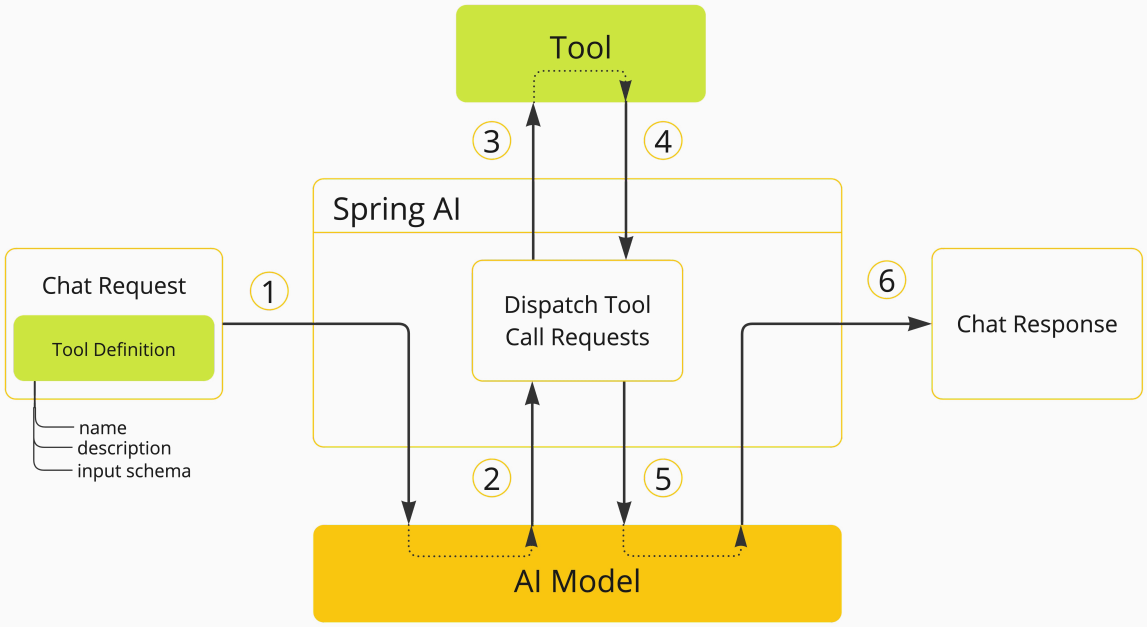
3. L'applicativo interroga il `@Bean` corrispondente alla classe che contiene il metodo con i parametri istanziati suggeriti dal LLM



©Spring AI

Note

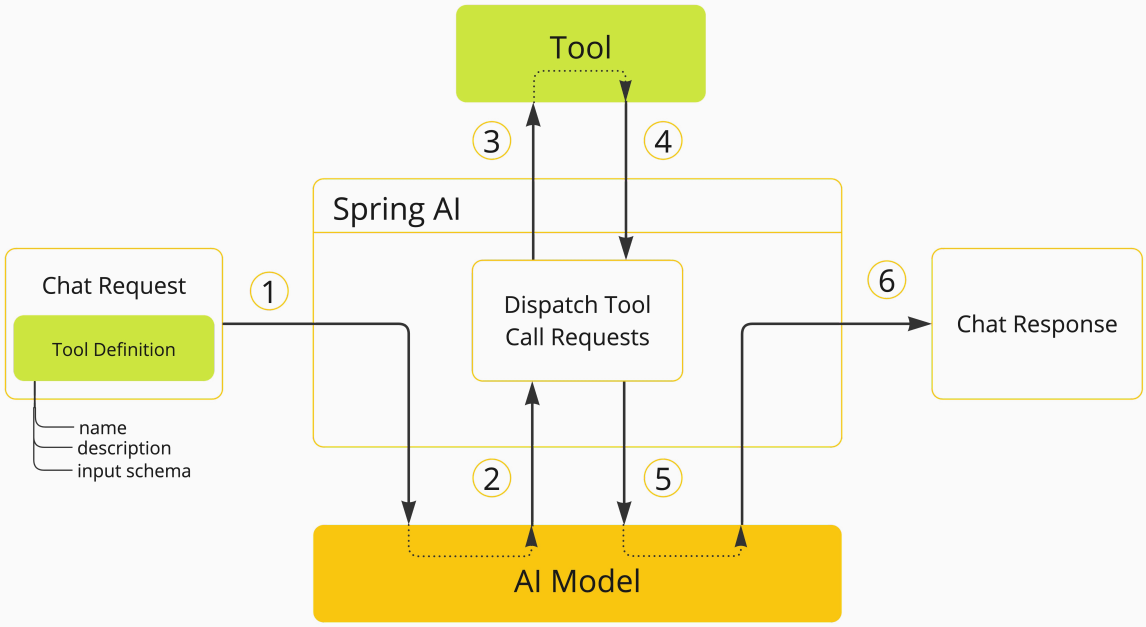
4. Il `@Bean` invoca il metodo con i parametri restituiti dal LLM e restituisce il risultato al *container* Spring



©Spring AI

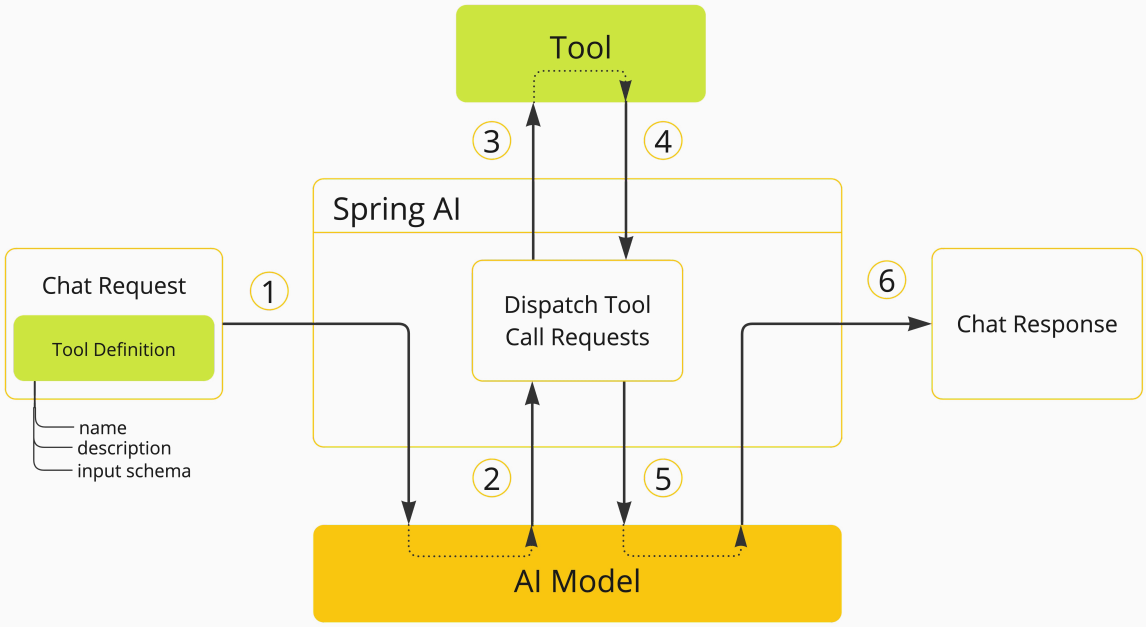
Note

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



Note

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



©Spring AI

Note

- 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 (`@Bean`)

Note

Tool Calling

Function as tool

Note

Definizione di *tool* come `@Bean`

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

    @Bean
    @Description("Get the weather in location")
    Function<WeatherRequest, WeatherResponse> currentWeather() {
        return weatherService;
    }

    @Bean
    @Description("Get the current temperature in a specific city")
    Function<TemperatureRequest, TemperatureResponse> cityTemperature() {
        return temperatureService;
    }
}
```

- Alternative all'approccio con annotazioni di metodo (`@Tool`)
- Il nome del `@Bean` diventa l'identificativo del *tool*
- `@Description` fornisce la descrizione del *tool*
  - Aiutare il modello a comprendere lo scopo del *tool*
- **Tipi funzionali supportati:** `Function<I,O>`, `Supplier<O>`, `Consumer<I>`, `BiFunction<I1,I2,O>`

Note

Definizione di *tool* come `@Bean`

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

    @Bean
    @Description("Get the weather in location")
    Function<WeatherRequest, WeatherResponse> currentWeather() {
        return weatherService;
    }

    @Bean
    @Description("Get the current temperature in a specific city")
    Function<TemperatureRequest, TemperatureResponse> cityTemperature() {
        return temperatureService;
    }
}
```

⚠ **Limitazioni** (vincoli di risoluzione dei tipi a runtime):

- ✖ Tipi primitivi
- ✖ `Optional`
- ✖ Collezioni (`List`, `Map`, `Array`, `Set`)
- ✖ Tipi asincroni e reattivi

Note

Aggiunta di *tool* al `ChatClient`

```java
// Utilizzo dei tool definiti come @Bean
ChatClient.create(chatModel)
    .prompt("What's the weather like in Copenhagen?")
    .toolNames("currentWeather")  // riferimento al nome del @Bean
    .call()
    .content();

// Oppure con tool di default condivisi
ChatClient.Builder builder = ChatClient.builder(chatModel)
    .defaultToolNames("currentWeather", "cityTemperature");

ChatClient client = builder.build();
client.prompt("What's the weather in Rome?").call().content();
```

- Riferimento al *tool* tramite nome del bean con `toolNames()`
- Possibilità di definire *tool* di default condivisi tra più richieste con `defaultToolNames()`
  - ⚠ Attenzione alle implicazioni di sicurezza con *tool* di default
- **Generazione automatica dello schema** degli input
- `@ToolParam` permette personalizzazione per descrizioni e stato *required* anche per tipi nested

Note