



# 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

- 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)

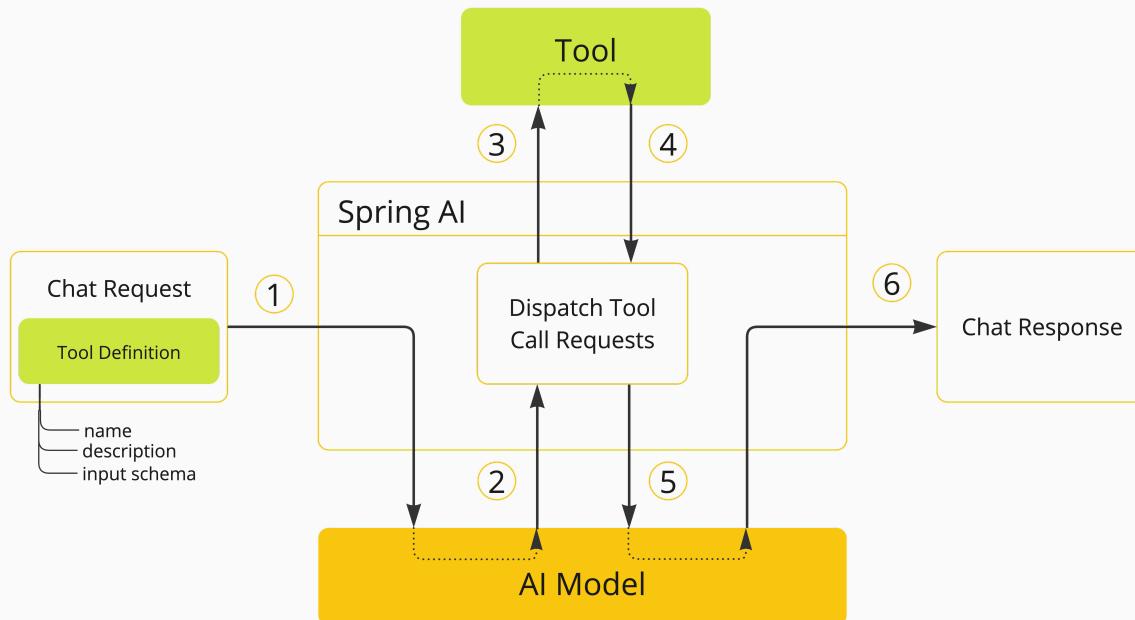
## Note

RAG	Tool calling
Leggere un manuale fai-da-te per risolverla 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'app client di eseguire il tool

## Note

## 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

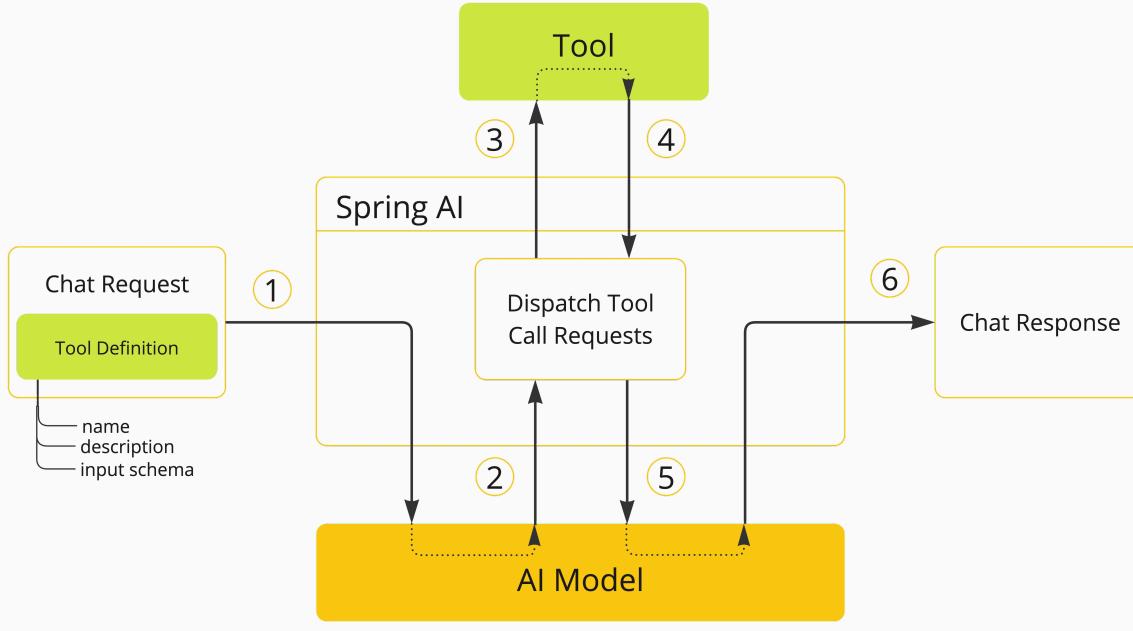


©Spring AI

## Note

## TOOL CALLING WORKFLOW IN SPRING AI - DEFAULT

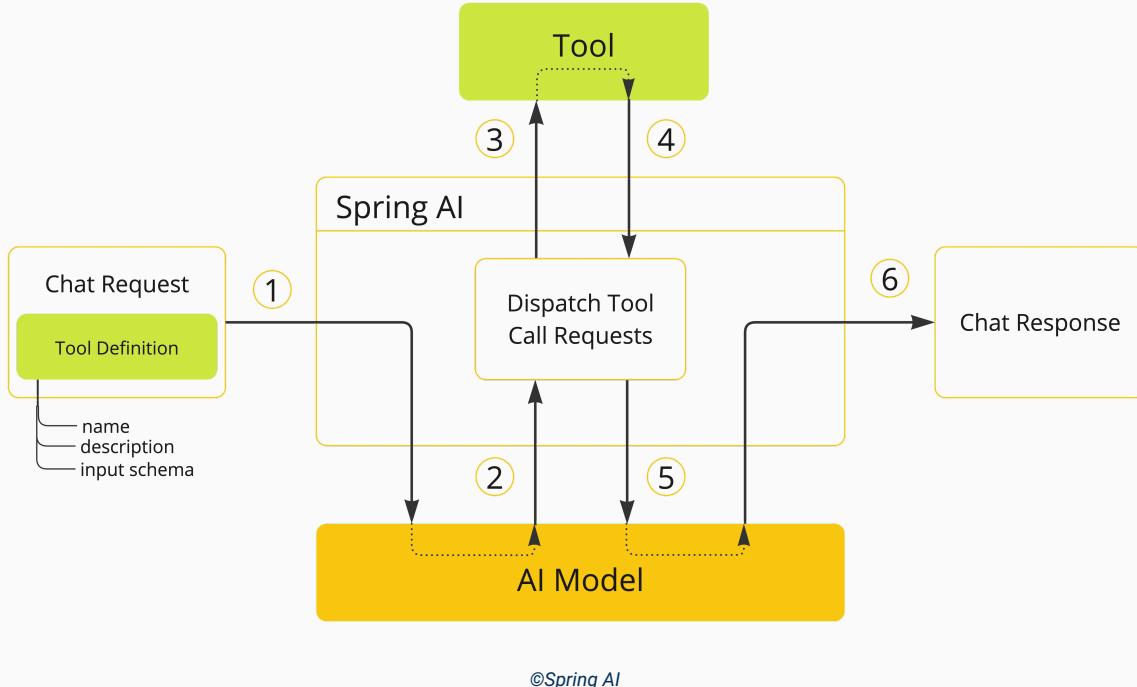
- 2 Se il LLM decide di chiamare un *tool*, invia una *response* con nome del *tool* e parametri istanziati, seguendo la schema definito per il *tool*



©Spring AI

## Note

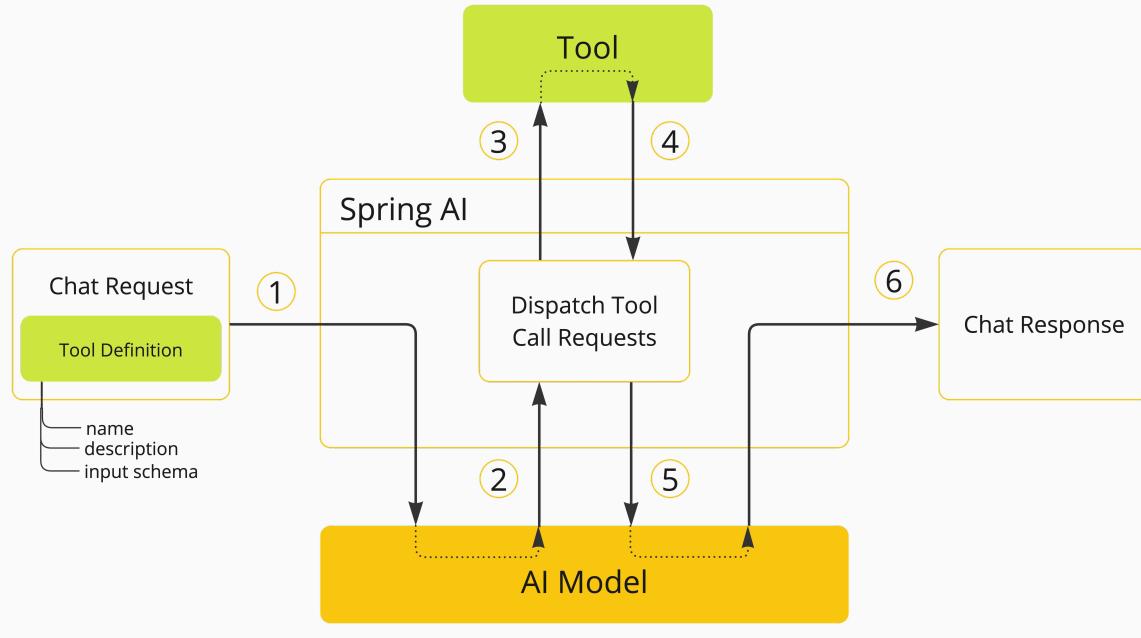
- 3** L'applicativo inoltra all'effettiva implementazione del tool nel container Spring la richiesta con istanziazione dei parametri suggeriti dal LLM



## Note

## TOOL CALLING WORKFLOW IN SPRING AI - DEFAULT

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

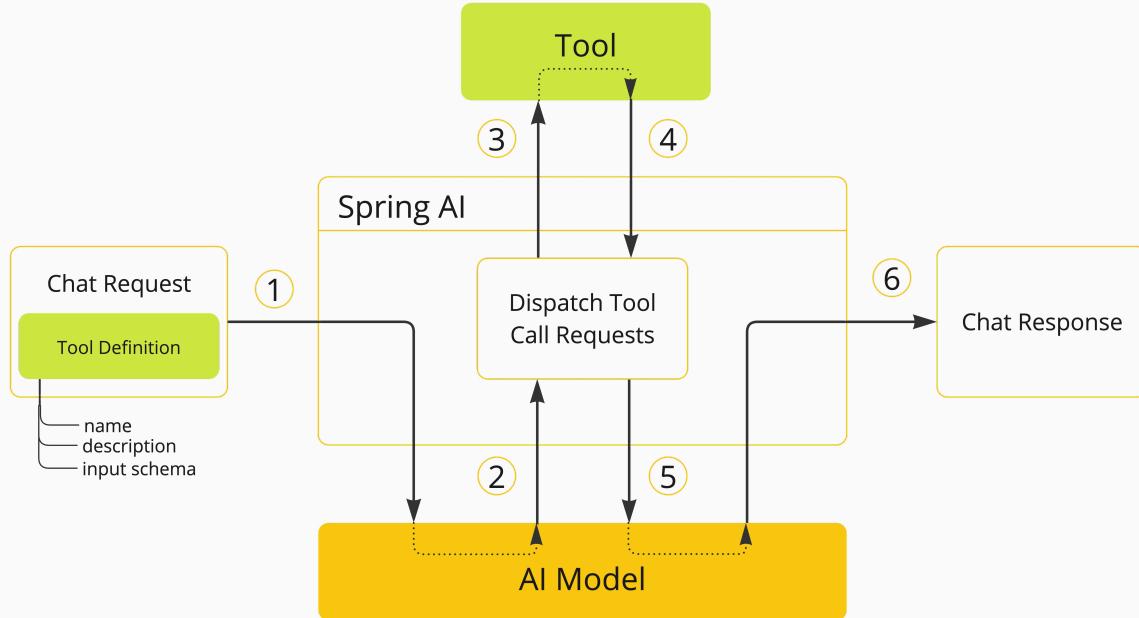


©Spring AI

## Note

## TOOL CALLING WORKFLOW IN SPRING AI - DEFAULT

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

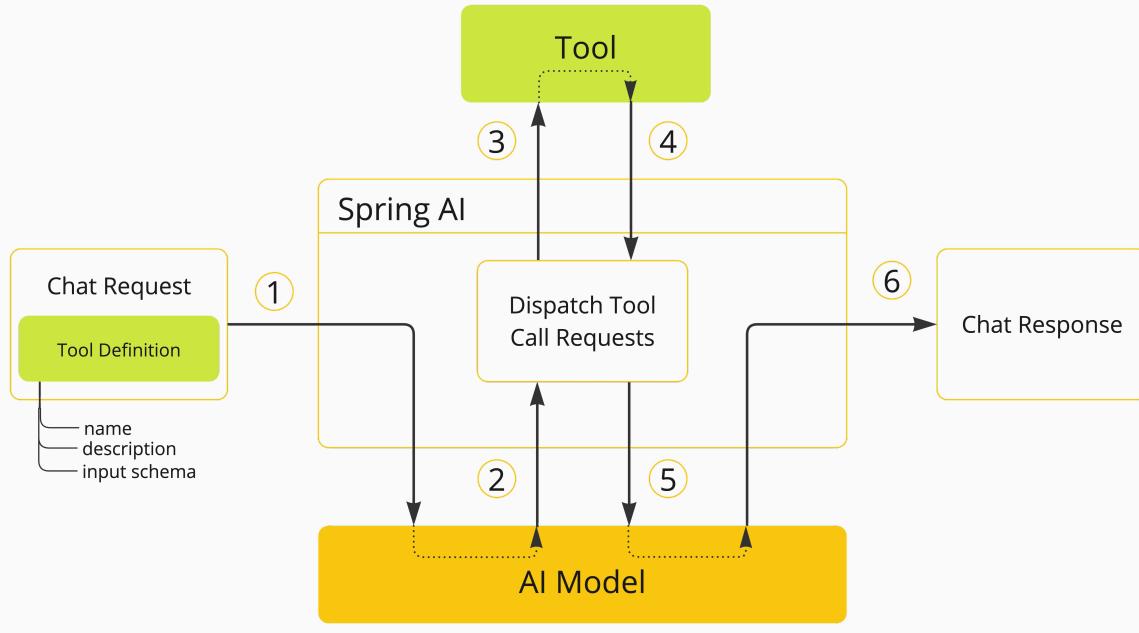


©Spring AI

## Note

## TOOL CALLING WORKFLOW IN SPRING AI - DEFAULT

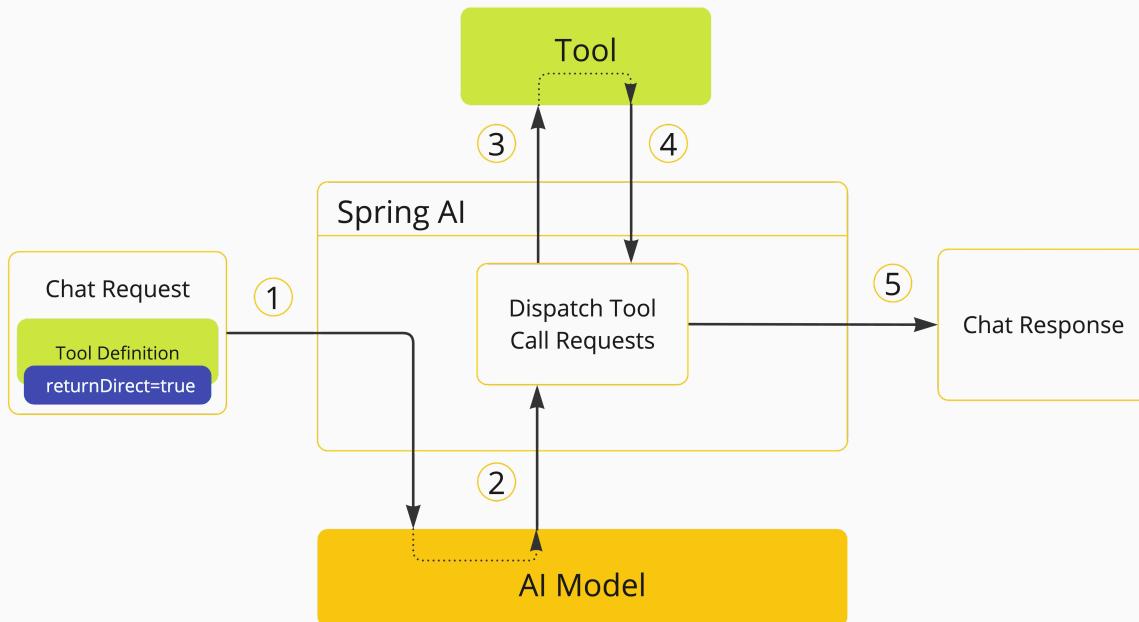
- 6** Il LLM restituisce la risposta finale al ChatClient/ChatModel sfruttando generazione di testo e contesto



©Spring AI

## Note

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



©Spring AI

## Note

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

## 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.",
          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

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

 "D: Che ore sono a New York?"

> **(thinking) – Scansano i tool a disposizione... –**

>\_ (thinking) – Vedò 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

## **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 getOllamaToolLocalTimeAnswer(QuestionRequest request) {

        return new Answer(this.ollamaToolChatClient
            .prompt()
            .tools(timeTools)
            ...
            .call()
            .content());
    }
}
```

## Note

## Classe di definizione dei tool

```
public class TimeTools {  
    String getCurrentLocalTime() {  
        ...  
    }  
}
```

## Utilizzo approccio *method-tool callback*

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

```

- ➔ Il sistema usa *reflection* per isolare il metodo
  - ➔ Costruito un `ToolCallback` che definisce la logica di utilizzo del *tool*
  - ⚠ Definizione `ToolCallback` *under-the-hood* anche con approccio dichiarativo

## 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

```
// 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