UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e implementazione di un ecosistema basato su Generative AI per l'accesso in linguaggio naturale ai web service di un ERP

Design and implementation of a Generative AI-based ecosystem for natural language access to the web services of an ERP

Relatore

Prof. Domenico Ursino

Candidata

Silvia Ciuffreda

Correlatori

Gianluca Bonifazi

Francesco Monteleone

ANNO ACCADEMICO 2025-2026

A chi c'è sempre stato.
A chi se n'è andato.

A mio padre,
che ha visto questo percorso iniziare e non finire.
Alla sua presenza silenziosa,
che ora sa di nostalgica assenza.
A tutti i suoi sacrifici fatti affinché io potessi essere qui, oggi.

Ovunque tu sia.
Questo traguardo lo dedico a te.

Sommario

Negli ultimi anni l'Intelligenza Artificiale ha assunto un ruolo sempre più rilevante nei processi di innovazione tecnologica, trovando una crescente applicazione anche nei contesti aziendali. In questa tesi viene proposta una PoC basata sulla Generative AI e sul Model Context Protocol (MCP) per consentire l'accesso ai web service di un sistema ERP reale a partire da richieste formulate in linguaggio naturale. In particolare, viene progettata un'architettura in cui Gemini CLI svolge il ruolo di ambiente di interazione con il modello linguistico, mentre l'accesso alle funzionalità dell'ERP è delegato ai tool esposti da un server MCP. Quest'ultimo, realizzato in Python tramite il framework low-code FastMCP, integra le API REST esposte dall'ERP in tool invocabili dall'LLM a partire da una specifica OpenAPI opportunamente adattata. Il lavoro comprende, inoltre, la gestione dell'autenticazione tramite il token JWT e la trasformazione delle risposte restituite dai servizi ERP, così da renderle più adatte alla rielaborazione da parte del modello. Infine, il sistema viene valutato attraverso scenari rappresentativi di utilizzo, volti a verificare le prestazioni complessive e a discutere le principali criticità emerse durante la sperimentazione.

Keyword: Generative AI, Large Language Model, Model Context Protocol, ERP, Web service REST, OpenAPI, FastMCP, Gemini CLI.

Indice

Elenco delle figure

Introduzione

L'Intelligenza Artificiale è una delle frontiere più affascinanti e innovative della tecnologia contemporanea. Tale campo interdisciplinare, in costante evoluzione, è ormai parte integrante della quotidianità. In particolare, l'avvento della Generative AI sta rivoluzionando il modo in cui le macchine comprendono e producono l'informazione, aprendo nuove prospettive nell'automazione dei processi produttivi. A partire dal rilascio pubblico di ChatGPT, alla fine del 2022, l'interesse verso queste tecnologie è cresciuto in modo significativo, anche grazie alle notevoli capacità da esse dimostrate nella comprensione e nella generazione di testo in linguaggio naturale in numerosi ambiti applicativi.

In tale scenario, le aziende stanno progressivamente trasformando il modo in cui concepiscono l'automazione dei propri processi operativi. Questa evoluzione apre nuove opportunità per lo sviluppo di assistenti virtuali capaci di affiancare la figura umana, ampliandone le capacità e permettendo ad essa di concentrarsi sulle attività più gratificanti, che richiedono, invece, un approccio umano diretto. La possibilità di formulare richieste in linguaggio naturale rappresenta, infatti, un vantaggio significativo, soprattutto in contesti in cui le informazioni sono distribuite tra sistemi complessi, quali ERP, CRM, database interni o servizi applicativi. In questi contesti, l'obiettivo non si limita a ottenere semplici interazioni conversazionali, ma consiste, soprattutto, nel semplificare l'accesso alle funzionalità aziendali che, normalmente, richiedono l'utilizzo di interfacce tecniche o procedure operative specifiche.

Sebbene siano estremamente potenti nell'elaborare il linguaggio naturale, gli LLM operano principalmente su una base di conoscenza statica limitata dai dati di addestramento e non dispongono, di per sé, di un accesso diretto e aggiornato ai sistemi esterni. Tale vincolo, però, risulta estremamente rilevante nei contesti aziendali dove le informazioni cambiano frequentemente e dove le risposte devono essere coerenti con lo stato effettivo dei sistemi informativi.

Per superare tale limite, è stato introdotto il concetto di function calling, che consente di connettere gli LLM ai servizi esterni allo scopo di renderli in grado di eseguire azioni nel mondo reale. Tuttavia, questo approccio presenta una criticità. Ogni singola applicazione, infatti, necessita di un'integrazione custom, attraverso l'implementazione delle logiche di comunicazione, della gestione degli errori e del mapping tra il linguaggio naturale e i comandi applicativi; ciò aumenta la complessità e riduce la portabilità delle soluzioni. Anche la presenza di numerosi framework per lo sviluppo di applicazioni basate sugli LLM ha contribuito a creare un panorama frammentato, in cui ciascun sistema adotta modalità proprie per rappresentare strumenti, dati e interazioni.

Un approccio che consente di mitigare, almeno in parte, le problematiche sopra descritte è rappresentato dal Model Context Protocol (MCP), un protocollo che permette agli LLM

di connettersi ai servizi esterni in modo uniforme. Definito da Anthropic e rilasciato come progetto open source nel novembre 2024, l'MCP può essere considerato l'"USB-C per l'Intelligenza Artificiale", ovvero un'interfaccia universale per la comunicazione tra i modelli e le applicazioni. Il protocollo si basa sul concetto di function calling, già introdotto nei modelli precedenti, ma ne estende le capacità, eliminando la necessità di sviluppare integrazioni personalizzate per ciascun caso d'uso. Grazie all'utilizzo dell'MCP, dunque, diventa possibile realizzare applicazioni più modulari e sensibili al contesto, nelle quali un LLM può accedere alle risorse esterne, utilizzare gli strumenti e orchestrare i flussi di lavoro complessi in modo standardizzato.

In questo contesto si colloca il presente lavoro di tesi, che nasce dall'esigenza di interfacciare un LLM con i web service esposti dal sistema SAM ERP2, resi disponibili come tool invocabili dal server MCP, con lo scopo di consentire agli utenti di interrogare il sistema utilizzando il semplice linguaggio naturale, senza la necessità di conoscere la struttura tecnica sottostante. Per riuscire nell'intento, l'implementazione viene svolta sfruttando le potenzialità del framework FastMCP, il quale consente di mappare facilmente le API REST esposte dal sistema ERP in tool MCP, senza dover gestire manualmente tutti i dettagli di integrazione.

L'obiettivo conclusivo del sistema è quello di dimostrare, attraverso una Proof of Concept, le potenzialità di un'architettura basata sull'MCP applicata a questo dominio operativo, riducendo la necessità per l'utente di conoscere direttamente gli endpoint, i parametri tecnici o la struttura delle risposte restituite dalle API.

La presente tesi è composta da otto capitoli organizzati come di seguito specificato:

- Nel Capitolo 1 sarà fornita una panoramica sul mondo della Generative AI, ripercorrendo le principali tappe che ne hanno favorito la diffusione per poi approfondire gli LLM, analizzandone l'architettura e i limiti.

- Nel Capitolo 2 verrà introdotto l'MCP, chiarendo le ragioni per cui può essere considerato un'evoluzione del meccanismo del function calling; in seguito verranno analizzati l'architettura, il workflow e i principali vantaggi che l'MCP introduce rispetto alle integrazioni custom sviluppate per i singoli casi d'uso.

- Nel Capitolo 3 verrà descritto il contesto aziendale, presentando l'azienda Centro Software e definendo l'ambiente tecnologico in cui si inserisce il progetto.

- Nel Capitolo 4 verrà presentata l'analisi dei requisiti del sistema sviluppato, con l'obiettivo di definire i requisiti funzionali e tecnici del progetto.

- Nel Capitolo 5 verrà illustrata la progettazione per la realizzazione dell'ecosistema, descrivendo le tecnologie selezionate e le scelte architetturali adottate.

- Nel Capitolo 6 verrà descritta l'implementazione tecnica del sistema, analizzando la struttura del progetto, la preparazione della specifica OpenAPI e l'implementazione del server MCP, con particolare attenzione alla definizione dei tool e alla gestione dell'autenticazione tramite token JWT.

- Nel Capitolo 7 verrà analizzato il funzionamento pratico dell'ecosistema, presentando alcuni scenari rappresentativi di utilizzo, scelti per evidenziare le capacità del sistema di gestire tipologie di richieste differenti; infine, si concluderà con una discussione sui risultati ottenuti e sui principali limiti emersi.

- Nel Capitolo 8 verranno tratte le conclusioni del lavoro svolto e saranno delineati alcuni possibili sviluppi futuri.

CAPITOLO 1

Introduzione alla Generative AI

La Generative AI rappresenta una forma avanzata di AI in grado di generare contenuti nuovi e originali, quali testi, immagini, musica e video.

Anche se l'attenzione verso questa tecnologia è cresciuta in modo significativo soltanto negli ultimi anni, le sue radici teoriche risalgono a diversi decenni fa.

In questo capitolo verranno dapprima ripercorse le principali tappe che hanno portato alla diffusione della Generative AI, per poi approfondire gli LLM, analizzandone architettura e limiti.

Infine, sarà mostrata una panoramica dei principali framework per lo sviluppo di applicazioni basate su questa tecnologia.

1.1 Definizione di Generative AI

La Generative AI è una branca dell'Intelligenza Artificiale che sta rivoluzionando il modo in cui le macchine producono e comprendono l'informazione.

Il termine "Generative Artificial Intelligence" è costituito da due elementi: "Artificial Intelligence" e "Generative". Con Artificial Intelligence si fa riferimento all'insieme di tecniche e metodologie che consentono a un sistema informatico di svolgere compiti che, tradizionalmente, richiederebbero l'intervento umano. L'aggettivo Generative, invece, indica la capacità di tali sistemi di produrre nuovi contenuti, come testi, immagini, musica o codice software.

Negli ultimi anni, l'attenzione verso la Generative AI è cresciuta in modo significativo, anche grazie alla diffusione di applicazioni accessibili al grande pubblico. Un momento chiave è stato il rilascio, nel novembre 2022, di ChatGPT, uno strumento sviluppato dalla società americana OpenAI, che ha rapidamente attirato l'interesse per le sue avanzate capacità di interazione linguistica tra l'uomo e la macchina.

L'enorme successo ottenuto ha contribuito ad accelerare lo sviluppo di nuove soluzioni basate su modelli generativi, evidenziando il potenziale di questa tecnologia in numerosi ambiti applicativi.

1.1.1 Gli strati dell'Intelligenza Artificiale

L'Intelligenza Artificiale (IA) è un ramo dell'informatica che si riferisce alle capacità di macchine o sistemi informatici di simulare processi cognitivi, tipicamente associati all'intelligenza umana, in grado di ragionare, apprendere e agire in modo autonomo.

All'interno di questo ampio dominio, l'IA può essere interpretata come una struttura gerarchica composta da diversi strati tecnologici, ciascuno dei quali rappresenta un sottoinsieme progressivamente più specializzato del precedente.

La Figura 1.1 mostra una visione d'insieme dei principali livelli di cui è composta l'IA.

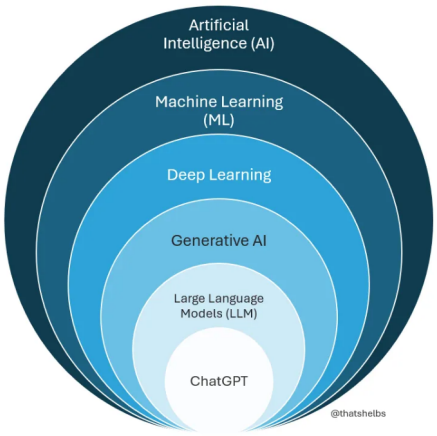


Figura 1.1: Panoramica dei vari strati di cui è costituita l'AI

Un primo sottoinsieme dell'IA è rappresentato dal Machine Learning (ML) che consente a un modello di apprendere pattern dai dati di addestramento di input con l'obiettivo di prevedere nuovi valori di output.

All'interno del Machine Learning si colloca il Deep Learning (DL), il quale utilizza reti neurali in grado di elaborare modelli più complessi rispetto all'apprendimento automatico. Tali reti, ispirate alla struttura e al funzionamento del cervello umano, sono costituite da più strati di nodi interconnessi che eseguono ciascuno un'operazione matematica.

La Generative AI rappresenta una specializzazione del DL e si caratterizza per la capacità di generare nuovi contenuti sempre più coerenti e realistici rispetto all'attività umana. Tale approccio, dunque, segna un passaggio significativo dall'IA prevalentemente predittiva a un'IA orientata alla creazione di contenuti.

All'interno della Generative AI si collocano i Large Language Models (LLM), modelli addestrati su enormi set di dati testuali e progettati per l'elaborazione e la generazione del linguaggio naturale. I task che sono in grado di svolgere sono diversi, tra i quali la generazione di testo, la traduzione automatica e la creazione di diversi tipi di contenuti creativi.

Un esempio concreto di applicazione di Generative AI è rappresentato dai modelli della famiglia GPT (Generative Pre-trained Transformer), che fungono da base per applicazioni conversazionali, come ChatGPT, progettate per l'interazione naturale con l'utente.

1.2 Evoluzione storica della Generative AI

Il concetto di Generative AI, pur avendo raggiunto una rapida diffusione soltanto negli ultimi anni, affonda le proprie radici nella storia più ampia dell'IA, risalente agli anni '50 del Novecento.

Un contributo fondamentale fu offerto dal matematico britannico Alan Turing, considerato uno dei padri dell'informatica moderna. Già nel 1936, infatti, Turing formalizzò il concetto di computabilità, ossia di un modello teorico tramite cui definire i limiti di ciò che un sistema computazionale è in grado di calcolare.

Nel 1950 lo stesso Turing si pose una delle domande che appare quanto più attuale al giorno d'oggi «Le macchine possono pensare?». A questo proposito, propose il cosiddetto

Test di Turing, ovvero un criterio pragmatico secondo il quale se una macchina conversa in modo indistinguibile da quello umano, allora essa può essere considerata intelligente.

Il termine Intelligenza Artificiale fu successivamente coniato da John McCarthy in occasione della Conferenza di Dartmouth del 1956, definendola come l'idea che "ogni aspetto dell'apprendimento o qualunque altra caratteristica dell'intelligenza possa, in linea di principio, essere descritto in maniera talmente precisa da costruire una macchina in grado di simularla". Secondo McCarthy, infatti, sarebbe stato sufficiente formalizzare nei minimi dettagli le caratteristiche dell'apprendimento umano e fornire queste ultime ad una macchina capace di riprodurle.

Fin dalla nascita della disciplina dell'IA come campo di ricerca, dunque, la possibilità che una macchina potesse non solo elaborare informazioni, ma anche produrre nuovi contenuti, ha rappresentato uno degli obiettivi più ambiziosi.

1.2.1 La "Golden Age" dell'IA (1956-1974)

Il periodo immediatamente successivo alla conferenza di Dartmouth fu caratterizzato da uno straordinario ottimismo, al punto da essere ricordato come la "Golden Age dell'IA".

I primi sistemi sviluppati, seppur molto semplici, mostrarono capacità sorprendenti se rapportate alle limitate risorse e alla conoscenza dell'epoca, tanto da alimentare aspettative elevate.

Nel 1959 Arthur Samuel implementò un programma per giocare a dama che incorporava il concetto di Machine Learning; quest'ultimo consentiva al sistema di migliorare progressivamente le proprie mosse attraverso l'esperienza.

Un altro risultato significato fu Eliza, sviluppato da Joseph Weizenbaum tra il 1964 e il 1966, un chatbot primitivo in grado di simulare una conversazione attraverso semplici regole di pattern matching. Sebbene Weizenbaum fosse sorpreso dalla tendenza degli utenti ad attribuire una comprensione umana al suo programma, Eliza rappresentò un esperimento rilevante nell'elaborazione del linguaggio naturale e nell'interazione uomo-macchina.

1.2.2 Il primo inverno dell'Intelligenza Artificiale (1974-1980)

L'entusiasmo iniziale per l'IA iniziò ben presto a scontrarsi con i limiti di capacità delle macchine. Molti problemi che, inizialmente apparirono semplici da risolvere, in realtà, si rivelarono estremamente complessi, soprattutto perchè le risorse computazionali disponibili non permettevano di trasformare in pratica molte idee teoriche.

A tal proposito, il rapporto Lighthill del 1973 criticò severamente lo stato di avanzamento dell'IA, sostenendo che i sistemi sviluppati fino a quel momento funzionavano solo in rappresentazioni semplificate della realtà e che il trasferimento di questa tecnologia a problemi concreti era molto più difficile del previsto.

In particolare, Lighthill evidenziò la mancata risoluzione del problema dell'esplosione combinatoria, cioè un fenomeno in cui il numero di possibili combinazioni di un sistema cresce esponenzialmente con la dimensione del problema, rendendo, quindi, impossibile una soluzione raggiunta con la sola forza di calcolo.

In altre parole, lo stato dell'arte dell'IA non consentiva di ottenere dei risultati soddisfacenti, poichè ciò avrebbe richiesto tempi di esecuzione e potenza computazionale superiori a quelli messi a disposizione.

Questo periodo, caratterizzato da una profonda disillusione, divenne noto come "Primo Inverno dell'IA". Esso rappresentò un momento di importante riflessione per la comunità scientifica, in cui si rese evidente la necessità di perseguire progressi incrementali e realistici, piuttosto che alimentare aspettative rivoluzionarie.

1.2.3 La rinascita dei sistemi esperti (1980-1987)

Gli anni '80 videro una graduale rinascita dell'interesse per l'IA, guidata principalmente dal successo riscontrato da alcuni sistemi esperti, i quali dimostrarono di apportare un valore concreto in domini specifici, tra i quali la medicina e la geologia.

In particolare, un sistema esperto fu definito come un programma progettato per riprodurre i processi decisionali di esperti umani in un determinato ambito. Fondato su una base di conoscenza costituita da regole logiche formali relative al dominio di interesse, il sistema prevedeva un motore inferenziale, che applicava tali regole per dedurre nuove informazioni, e un'interfaccia, che consentiva all'utente di interagire con esso.

In questo periodo, inoltre, si affermarono concetti come il backpropagation e le reti neurali multilivello, che resero possibile addestrare modelli più complessi.

1.2.4 Il secondo inverno dell'Intelligenza Artificiale (1987-1993)

Nonostante l'entusiasmo iniziale, i sistemi esperti iniziarono ben presto a mostrare limitazioni significative. Essi, infatti, risultavano costosi da sviluppare e, tra l'altro, funzionavano correttamente soltanto in domini molto ristretti.

Alla luce di tali criticità, si iniziò a parlare di Knowledge Bottleneck, ovvero collo di bottiglia della conoscenza, riferendosi alla difficoltà dei sistemi esperti di rappresentare, comprendere, e utilizzare le informazioni disponibili.

In questo periodo, spesso indicato come il "Secondo Inverno", il campo dell'IA, inizialmente orientato verso un obiettivo comune, iniziò a diversificarsi in svariate sottodiscipline specializzate. Tra queste si affermarono la visione artificiale, la robotica, l'elaborazione del linguaggio naturale e l'apprendimento automatico, ciascuna caratterizzata da obiettivi e metodologie dedicati.

1.2.5 Le prime vere applicazioni (1993-2012)

In seguito alle problematiche riscontrate negli anni precedenti, gli esperti ridimensionarono le proprie ambizioni, divenendo maggiormente consapevoli dei limiti del settore.

A partire dagli anni '90, due forze cambiarono la prospettiva: da una parte la crescente disponibilità di dati, dall'altra una maggiore potenza computazionale che resero praticabili soluzioni che fino ad allora erano rimaste prevalentemente teoriche.

L'approccio simbolico venne progressivamente affiancato, e poi superato, dall'apprendimento statistico e dalle cosiddette Deep Neural Network (DNN). Si passò, quindi, dall'utilizzo di regole esplicite e conoscenza codificata manualmente a modelli in grado di estrarre automaticamente relazioni dai dati, per mezzo di principi matematici e statistici.

In altre parole, le macchine iniziarono a non essere più programmate, bensì allenate.

Ciò, inoltre, diede un forte impulso al campo della Computer Vision, disciplina che consentì ai modelli di interpretare i contenuti visivi e che divenne la base delle più moderne applicazione di IA, dai filtri realtivi alle foto, fino al riconoscimento di pattern in campo medico.

Ciò fu reso reso possibile grazie all'evoluzione degli algoritmi, che divennero capaci di comprendere strutture complesse nei dati e di mapparle in una gerarchia di informazioni in grado di catturare prima le caratteristiche più importanti per distinguere, ad esempio, un oggetto dall'altro, per poi aggiungere sempre maggiore contesto e dettagli, al fine di raffinarne la comprensione globale.

1.2.6 La rivoluzione (2016-2017)

Nel 2016 arrivò un'altra svolta con la diffusione di AlphaGo, un sistema in grado di imparare a giocare a Go non seguendo regole preprogrammate, bensì allenandosi sulla base dell'esperienza e simulando, quindi, partite contro se stesso.

In questo contesto, emerse il concetto di Reinforcement Learning, ossia l'idea che un modello potesse migliorare sulla base di tentativi, errori e ricompense, in modo analogo a quanto avviene nell'apprendimento umano.

Tale approccio introdusse nella ricerca la combinazione tra percezione, previsione e pianificazione, che sarebbe poi divenuta la base dei più moderni sistemi, capaci non solo di riconoscere, ma anche di decidere e pianificare.

Nel 2017 si verificò un vero e proprio cambio di paradigma. Con la pubblicazione dell'articolo *Attention is all you need*, infatti, nacque il Transformer, un'architettura capace di analizzare grandi quantità di testo e di apprendere relazioni sottili tra parole, frasi e concetti, diventando la base dei modelli linguistici su larga scala, noti come LLM.

1.3 I Large Language Model (LLM)

I Large Language Model (LLM) rappresentano uno degli sviluppi più significativi nel campo dell'IA degli ultimi anni. Essi sono modelli di Deep Learning addestrati su enormi quantità di dati e progettati per comprendere e generare output testuali, a partire da richieste formulate in linguaggio naturale.

Il cambiamento sostanziale introdotto da questi modelli si articola lungo tre dimensioni principali:

- *scala*, grazie alla possibilità di introdurre miliardi di parametri, che consentono di affinare il processo di apprendimento degli algoritmi;
- *dati*, con la capacità di analizzare quantità sempre maggiori di informazioni;
- *capacità emergenti*, quali ragionamento, traduzione, scrittura, ma anche programmazione di codice.

Gli LLM non si limitano a classificare i dati, ma sono in grado di generare nuovi contenuti e segnano, dunque, un passaggio da un'IA orientata alla classificazione a un'IA capace di produrre.

Con il lancio di ChatGPT nel 2022, l'IA è entrata nella vita quotidiana di milioni di persone, le quali hanno iniziato ad interagire con un modello linguistico come se fosse un interlocutore umano.

Una caratteristica chiave di questa evoluzione è la convergenza, ovvero un singolo modello capace di svolgere diversi compiti, come analizzare un'immagine, comprendere e sintetizzare un contenuto, rispondere a domande, ma anche generare codice software.

1.3.1 Architettura Transformer

Gli LLM si basano su un'architettura di rete neurale denominata Transformer, introdotta nel rivoluzionario paper *"Attention Is All You Need"* nel 2017, che costituisce il cuore alla base dell'attuale rivoluzione dell'IA Generativa.

Il Transformer si fonda sul meccanismo di *attenzione* con l'obiettivo di consentire al modello la comprensione del contesto globale di un testo. A partire da una sequenza di parole, infatti, il modello predice la parola successiva in modo parallelo e contestuale, superando i limiti delle vecchie reti ricorrenti che, invece, processavano le parole una alla volta in ordine

temporale. In tal senso, la previsione assume la forma di una distribuzione di probabilità su tutte le possibili parole che possono seguire la parola corrente.

Una volta prelevata la parola più probabile, essa viene concatenata alla sequenza di input e il processo viene iterato fino a quando il modello non incontra un simbolo di fine testo.

La Figura 1.2 illustra le principali componenti e le diverse fasi che caratterizzano il Transformer.

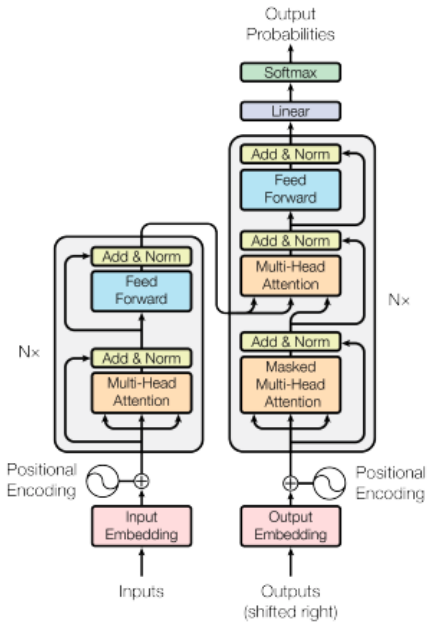


Figura 1.2: Principali componenti e fasi che caratterizzano l'architettura Transformer

Il primo passo è la suddivisione dell'input in unità minime chiamate *token*, che possono corrispondere a parole intere, sottoparti di parola, o persino, singoli caratteri, a seconda del vocabolario usato dal modello.

Dopodiché, ciascun token viene trasformato in un vettore numerico, chiamato *embedding*, che ne codifica il significato. Tali vettori risiedono in uno spazio ad alta dimensionalità, i cui valori sono inizialmente casuali ma vengono aggiornati durante l'addestramento, in modo tale che parole semanticamente simili risultino vicine nello spazio vettoriale. Inoltre, ai vettori di embedding vengono aggiunti i cosiddetti *positional encoding*, cioè una codifica numerica che rappresenta la posizione del token, al fine di facilitare la loro elaborazione.

La sequenza di vettori così ottenuta attraversa un'operazione nota come *attention block*, che consente ai vettori di interagire tra loro, allo scopo di determinare i token che, nel contesto, sono più rilevanti per l'aggiornamento dei vettori stessi.

Tale operazione viene eseguita in parallelo in un sorta di *Multi-"Head" Attention*, conciascuna specializzata per cogliere un tipo diverso di relazione; tra le relazioni considerate vi sono quelle grammaticali, semantiche e di dipendenza sintattica.

In seguito, questi vettori passano attraverso un *feed-forward network*, ovvero una piccola rete neurale che lavora in parallelo su tutti i token. Tale blocco prende ciascun vettore e lo trasforma in qualcosa di più elaborato, al fine di permettere al modello di rappresentare relazioni complesse.

Tale processo si ripete iterativamente fino a condensare il significato complessivo del testo in un vettore finale, a partire dal quale il modello decide quale parola deve seguire quella precedente. Per farlo, moltiplica il vettore finale per un'ulteriore matrice, chiamata *matrice di unembedding*, generando un elenco di valori, uno per ciascun token.

Infine, tali valori vengono normalizzati tramite la funzione *softmax*, che li trasforma in una distribuzione di probabilità dalla quale viene selezionato il token successivo.

1.3.2 LLM nel corso degli anni

Tale sezione presenta brevemente i principali LLM e ne descrive gli aspetti maggiormente innovativi.

Modelli GPT. I modelli GPT, creati da OpenAI. hanno rivoluzionato il campo dell'IA Generativa negli ultimi anni.

Essi hanno avuto inizio con GPT-1, sviluppato nel 2018 che, adottando l'architettura Transformer, utilizzava un paradigma basato su unsupervised pretraining seguito da supervised fine-tuning.

GPT-2, rilasciato successivamente, si è concentrato sulla modellazione linguistica non supervisionata senza fine-tuning specifico puntando, invece, alla risoluzione multi-task attraverso un approccio probabilistico.

GPT-3, sviluppato nel 2020, ha introdotto il concetto di In-Context Learning (ICL), consentendo al modello l'apprendimento few-shot o zero-shot.

In generale, ChatGPT ha contribuito fortemente alla diffusione degli LLM al grande pubblico, grazie alle sue straordinarie capacità di generazione di testo, nonchè alla sua interfaccia semplice da utilizzare.

Nel marzo 2023 è stato introdotto GPT-4, il quale ha aggiunto funzionalità multimodali, consentendo all'utente di caricare documenti o immagini e di ricevere in output un'analisi completa che può includere sintesi testuali, descrizioni di immagini e approfondimenti derivati dalla comprensione combinata di contenuti testuali e visivi.

Nel 2025 è stato presentato GPT-5, versione ulteriormente ottimizzata per compiti complessi, con particolare attenzione al ragionamento avanzato, alla comprensione contestuale profonda e all'integrazione con strumenti esterni tramite API.

Modelli di Google. In seguito all'introduzione dell'architettura Transformer, nel 2018 Google ha lanciato BERT, un modello innovativo di elaborazione del linguaggio naturale, la cui principale novità risiede nella capacità di comprendere il contesto delle parole in modo bidirezionale, considerando sia le parole precedenti che quelle successive all'interno di una frase.

A partire dal 2024, entra in scena Gemini, sviluppato con l'obiettivo di realizzare un modello multimodale nativo. A differenza dei primi LLM, progettati principalmente per l'elaborazione testuale, Gemini è stato concepito fin dall'origine per supportare diverse sorgenti informative, tra cui testo, immagini, audio e codice.

Un ulteriore elemento distintivo è l'attenzione all'integrazione con strumenti esterni e servizi cloud, che consente al modello l'accesso a informazioni aggiornate e l'interazione con ambienti software complessi.

Nel febbraio 2026 è stato rilasciato Gemini 3.1 Pro, attualmente considerato il modello più avanzato della serie, con ulteriori miglioramenti nelle capacità di ragionamento, nella comprensione di contesti e nel reasoning strutturato.

Modelli di Anthropic. Tra gli attori più rilevanti nel panorama degli LLM si colloca Anthropic, azienda fondata con un forte interesse alla sicurezza e all'affidabilità dei sistemi di IA.

Il modello più recente sviluppato è Claude4, con l'obiettivo di integrare elevate capacità di ragionamento con un'attenzione strutturale agli aspetti di sicurezza.

Uno degli elementi distintivi di Claude4 è l'approccio denominato "Constitutional AI", una metodologia di addestramento che integra principi espliciti per guidare il comportamento del modello, riducendo risposte dannose senza ricorrere alla supervisione umana.

Questo approccio, quindi, mira a rendere il modello più trasparente e coerente nei contesti applicativi sensibili.

1.4 Limitazioni degli LLM

Nonostante la loro potenza, gli LLM presentano alcune debolezze, poiché essi vengono addestrati su corpora molto ampi che contengono qualsiasi forma di informazione. Ciò può portare a comportamenti indesiderati o a distorsioni nella comprensione del linguaggio, con un rischio maggiore di diffondere disinformazione.

1.4.1 Come gli LLM memorizzano la conoscenza

A differenza delle basi di conoscenza tradizionali, gli LLM memorizzano la conoscenza implicitamente attraverso i loro parametri, conservandola come un insieme di relazioni probabilistiche tra token. Ciò significa che quando si chiede al modello di rispondere a una domanda o di generare del testo esso non ricorda la risposta in modo letterale, ma ricostruisce la sequenza più probabile basandosi sui pattern appresi.

Alla luce di ciò, è facile intuire che i contenuti generati dagli LLM possono spesso risultare inaccurati o deviare dalla verità, a causa di induzioni errate o informazioni obsolete.

La conoscenza immagazzinata nei pesi rappresenta il sapere di base del modello, ovvero ciò consente ad esso di comprendere il linguaggio e di produrre testo coerente.

Per fornire al modello informazioni specifiche e aggiornate, è necessario inserirle direttamente nel prompt o metterle a disposizione attraverso fonti esterne. Quest'approccio permette, dunque, di ampliare la conoscenza del modello estraendo dati da fonti aggiornabili in tempo reale.

A tal fine, è stata sviluppata la tecnica denominata Retrieval-Augmented Generation (RAG), la quale combina un modello linguistico con un sistema di recupero delle informazioni, che cerca nei documenti esterni le conoscenze pertinenti al prompt. Le informazioni così trovate vengono integrate nel contesto del modello e utilizzate per generare una risposta più accurata e fondata sui dati forniti.

1.4.2 Allucinazioni

Nel contesto degli LLM, il termine "allucinazione" si riferisce alla generazione di contenuti falsi o non verificabili, che possono variare da lievi incongruenze a vere e proprie affermazioni inventate, pur apparendo plausibili dal punto di vista linguistico.

In altre parole, il modello produce risposte che sembrano vere ma che, in realtà, non derivano da informazioni accurate, nè dal contesto effettivamente fornito. Ciò accade poichè il modello non utilizza un sistema di verifica dei fatti, ma genera testo sulla base di pattern statistici appresi durante la fase di addestramento.

Le allucinazioni derivano, quindi, da processi di generalizzazione che portano il modello a integrare informazioni probabili, ma non necessariamente coerenti con la realtà.

Le allucinazioni possono essere categorizzate su diversi livelli di granularità.

Al livello più basso si collocano le contraddizioni di frasi, che si verificano quando il modello genera un'affermazione in conflitto con una precedente all'interno dello stesso testo.

Un secondo livello riguarda le contraddizione rispetto al prompt, ossia contesti in cui la risposta generata contraddice le informazioni fornite dall'utente.

Infine, le contraddizioni fattuali si verificano quando un'affermazione è in conflitto con conoscenze consolidate e verificabili, generando informazioni errate, sebbene le stesse vengano presentate come veritiere.

Identificare le cause che portano il modello ad "allucinare" non è semplice, poiché il processo di generazione dell'output risulta, in gran parte, una scatola nera.

Tuttavia, sono stati individuati alcuni fattori principali. Uno di questi riguarda la qualità e la natura dei dati di addestramento. Gli LLM, infatti, vengono pre-addestrati su un ampio corpus di testo provenienti da fonti eterogenee, che possono contenere errori, rumore, distorsioni o incongruenze testuali.

Inoltre, tali dati potrebbero non coprire tutti i possibili domini su cui l'LLM è chiamato ad operare, inducendo il modello a generalizzare in maniera scorretta.

Un altro fattore è basato sul metodo di generazione del testo. A tale scopo, gli LLM utilizzano diverse strategie, quali la beam search, il sampling, la stima della massima verosimiglianza o il Reinforment Learning. Queste tecniche implicano un compromesso tra coerenza, creatività e accuratezza. Ad esempio, parametri che favoriscono risposte più originali possono aumentare il rischio di produrre contenuti non completamente corretti.

Un'altra causa comune di allucinazioni è legata all'ambiguità del contesto nel prompt di input. D'altronde, il prompt costituisce la guida principale per la generazione dell'output; se esso è incompleto, vago o ambiguo, il modello può interpretarlo in maniera errata e generare risposte non pertinenti.

Per tale ragione, il modo più semplice per ridurre le allucinazioni consiste nel fornire prompt chiari e specifici al sistema. Più preciso e dettagliato è il prompt di input, maggiore è la probabilità che gli LLM generino output pertinenti e più accurati.

Un'ulteriore tecnica è rappresentata dall'utilizzo di few-shot learning, la quale consiste nel fornire al modello più esempi dell'output desiderato. Ciò consente al modello di cogliere meglio lo stile e il contenuto della risposta attesa dall'utente.

Infine, l'uso di strategie di mitigazione attiva, come la Retrieval-Augmented Generation (RAG) permettono di integrare nel prompt informazioni aggiornate e verificate da fonti esterne, riducendo l'affidamento esclusivo alla conoscenza probabilistica interna del modello.

1.4.3 Bias

I bias negli LLM rappresentano visioni distorte o unilaterali che influenzano il modo in cui il modello interpreta e genera il testo.

Esistono diversi tipi di bias, principalmente causati da imprecisioni nella fase di addestramento.

Se il corpus di addestramento contiene linguaggio discriminatorio, stereotipi di genere o visioni politicamente sbilanciate, tali pattern linguistici vengono interiorizzati dal modello e possono riemergere nei suoi output.

Vi sono, inoltre, bias di selezione dei dati, per cui anche le decisioni su quali dati includere o escludere dal corpus possono introdurre distorsioni. Ad esempio, se una categoria di contenuti o una lingua è sottorappresentata, il modello risulterà meno accurato o più incline a errori in quei contesti.

Rimuovere completamente i bias è estremamente difficile, in quanto il linguaggio stesso è intrinsecamente carico di bias. Le parole riflettono valori culturali, norme sociali e prospettive storiche, e i testi da cui gli LLM apprendono sono il prodotto di società in cui, purtroppo, esistono i pregiudizi.

Di conseguenza, gli LLM non solo apprendono regole grammaticali e strutture semantiche, ma anche pattern culturali e ideologici difficili da separare dai contenuti neutri. Ad esempio, un modello addestrato prevalentemente su testi in lingua inglese potrebbe riflettere una prospettiva eurocentrica o occidentale. Ciò significa che anche un modello ben addestrato

può presentare bias sottili, come preferire certi aggettivi per descrivere specifiche categorie di persone, o generare risposte più dettagliate per certi argomenti rispetto ad altri.

1.5 Framework e tool per la Generative AI

La rapida evoluzione della Generative AI ha stimolato lo sviluppo di vari framework progettati per facilitare la progettazione e l'implementazione basati su questa tecnologia. Tali strumenti, infatti, consentono di creare applicazioni più sofisticate e intelligenti, riducendo la complessità associata all'utilizzo diretto degli LLM.

Di seguito verrà illustrata la panoramica dei principali framework e strumenti di IA Generativa.

1.5.1 LangChain

Sviluppato da Harrison Chase e lanciato nell'ottobre 2022, LangChain è una piattaforma open source che mira a fornire un toolkit completo per l'integrazione degli LLM in diversi casi d'uso, tra cui chatbot personalizzati, risposte automatiche a domande e sintesi di testi (Figura 1.3)

Figura 1.3: Icona di LangChain

I componenti principali di cui è composto Langchain sono i seguenti:

- *LLM*: LangChain funge da interfaccia standard che consente l'interazione con un'ampia gamma di modelli linguistici attraverso chiamate API.
- *Modelli di prompt*: LangChain offre una varietà di classi e funzioni progettate per semplificare il processo di creazione e gestione dei prompt. In particolare, il prompt engineering viene definito come l'arte di comporre prompt che forniscano in modo efficiente il contesto necessario all'LLM, al fine di interpretare l'input e strutturare l'output nel modo più utile per l'utente.
- *Memoria conversazionale*: LangChain incorpora moduli di memoria che consentono la gestione delle conversazioni passate; tale caratteristica è fondamentale per i chatbot che devono richiamare le interazioni precedenti.
- *Agenti intelligenti*: LangChain fornisce agli agenti un kit di strumenti completo; gli agenti possono scegliere quali strumenti utilizzare in base agli input dell'utente.
- *Indici*: Langchain fornisce metodi per organizzare i documenti in modo da facilitarne l'accesso.
- *Chain*: sebbene l'utilizzo di un singolo LLM possa essere sufficiente per compiti più semplici, LangChain fornisce un'interfaccia standard e implementazioni utilizzate per concatenare gli LLM tra loro per applicazioni più complesse.

Figura 1.4: Icona di LlamaIndex

1.5.2 LlamaIndex

LlamaIndex (Figura 1.4) è un framework di orchestrazione avanzato progettato per amplificare il potenziale degli LLM.

Nonostante le avanzate capacità degli LLM, un ostacolo significativo emerge quando si utilizzano i modelli con informazioni specifiche e riservate. Sebbene vengano addestrati su ampi set di dati pubblici, gli LLM spesso non dispongono dei mezzi per interagire con dati privati o specifici di un determinato dominio.

LlamaIndex si propone di colmare questa lacuna, fornendo un insieme di strumenti per l'acquisizione, l'indicizzazione e l'interrogazione di varie fonti di dati, al fine di diventare una soluzione versatile alle esigenze applicative di Generative AI.

Come illustrato in Figura 1.5, una delle caratteristiche più interessanti di LlamaIndex è la capacità di importare i dati nel sistema acquisendoli direttamente dalla loro fonte di origine. A tal proposito, il framework dispone di numerosi estrattori in grado di gestire la maggior parte dei formati documentali comuni, quali PDF, CSV, Word o Powerpoint.

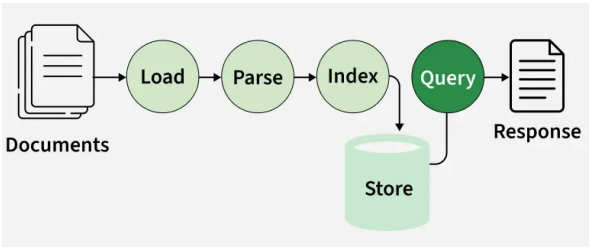


Figura 1.5: Componenti principali di LlamaIndex

I documenti forniti in input vengono successivamente processati ed indicizzati in un formato standardizzato, organizzandoli in modo tale che i modelli di linguaggi li possano comprendere facilmente. In particolare, le informazioni testuali vengono convertite in rappresentazioni vettoriali mediante il meccanismo degli embedding.

Al fine di gestire in modo efficiente documenti di grandi dimensioni, LlamaIndex suddivide i dati in unità più piccole e gestibili, chiamate *chunk*. Processando solo una porzione del dataset alla volta, esso evita il sovraccarico della memoria, migliora la stabilità del sistema e facilita la parallelizzazione delle operazioni di elaborazione, riducendo, così, i tempi di esecuzione.

1.5.3 Datapizza AI

Datapizza AI è un framework open source italiano per la Generative AI, sviluppato da Datapizza (Figura 1.6), una start-up e tech community italiana attiva nei settori del software e dell'IA; esso è stato rilasciato ufficialmente il 13 ottobre 2025.

Tale framework è implementato in codice Python con l'obiettivo di consentire la creazione di agenti AI, sistemi RAG e pipeline di automazione, adottando un approccio engineer-first.

Il progetto nasce con una finalità ben definita, ovvero portare la Generative AI in produzione, al fine di garantire controllo, trasparenza e modularità, ed evitare, al contempo,

Figura 1.6: Icona di Datapizza AI

la complessità tipica dei framework troppo astratti o percepiti come black box che, spesso, rallentano lo sviluppo e il debugging delle applicazioni.

Tra le principali criticità riscontrate nei framework esistenti compaiono:

- *Debugging difficile*: le diverse astrazioni rendono complesso capire dove si verificano gli errori. Spesso ci si ritrova a ricostruire manualmente ogni singolo passaggio, navigando tra diversi livelli di astrazione che nascondono il vero problema.
- *Dipendenza da terze parti*: ogni bug esterno può bloccare un progetto intero.
- *Curva di apprendimento ripida*: ogni framework ha regole proprie da imparare.
- *Personalizzazione limitata*: è difficile adattare il comportamento ai diversi casi d'uso.

A partire dall'individuazione di tali problemi, il team di Datapizza ha progettato un'architettura basata su un layer sottile e trasparente costruito sopra gli SDK nativi dei principali provider, tra cui OpenAI, Gemini, Anthropic e Mistral.

Tale approccio consente di mantenere un'architettura componibile, osservabile e modificabile a basso livello, ponendo al centro il principio del pieno controllo su ogni componente del sistema.

I pilastri fondamentali del framework Datapizza AI possono essere sintetizzati come segue:

- *Bassa astrazione e controllo completo*: ogni azione è leggibile e monitorabile mediante un tracing completo di ogni step, uno jogging strutturato e una gestione trasparente degli errori.
- *Massima flessibilità e personalizzazione*: ogni componente è intercambiabile e personalizzabile. Il basso livello di astrazione permette di creare moduli custom adatti ad ogni esigenza, che rimangono compatibili con il resto del framework. In aggiunta, non si è vincolati dalle scelte del framework, ma è possibile costruire architetture personalizzate mantenendo la compatibilità nativa.
- *Pronto per la produzione*: prima del rilascio pubblico, Datapizza-AI è stato testato su oltre dieci progetti enterprise reali. Il framework è già stato usato per agenti multicanale, sistemi RAG interni, e automazioni documentali complesse, riducendo i tempi di sviluppo e debugging.

È bene sottolineare come Datapizza-AI non si pone in contrapposizione ai framework esistenti, piuttosto come una loro evoluzione ispirata ai punti di forza delle soluzioni attuali. Ad esempio, framework come Langchain risultano efficaci per task semplici, ma diventano rigidi appena si esce fuori dai binari che lo stesso framework impone. È proprio questa limitazione che ha motivato il team di Datapizza a sviluppare una soluzione propria, guidata anche dalla convinzione che anche dall'Italia sia possibile costruire tecnologie di alto livello.

1.6 L'impatto della Generative AI sul mondo del lavoro

L'AI, e in particolare l'AI Generativa, è ormai parte integrante della quotidianità e sta cambiando profondamente anche il ruolo delle persone in ambito lavorativo.

Questo passaggio epocale non si limita soltanto all'integrazione di strumenti tecnologici, bensì impone una revisione profonda della relazione tra le competenze umane e quelle artificiali.

In un mondo dove le macchine non si riducono più solo a eseguire, ma generano contenuti, ipotesi e soluzioni, il confine tra ciò che è digitale e ciò che è umano diventa sempre meno netto.

La narrazione dominante tende spesso a rappresentare l'IA come una minaccia per l'occupazione. In realtà, la sua introduzione nei contesti aziendali può essere interpretata come un acceleratore di cambiamento, capace di automatizzare compiti ripetitivi, aumentare l'efficienza e liberare risorse umane da attività a basso valore aggiunto.

Accanto alla possibile riduzione di alcuni ruoli tradizionali, si sta assistendo alla nascita di nuove professionalità, molte delle quali legate alla gestione, progettazione e supervisione dei sistemi intelligenti.

La trasformazione, quindi, non è tanto quantitativa quanto qualitativa, per cui cambiano le attività, ma soprattutto le competenze richieste.

In questo contesto, diventa necessario sapere come formulare obiettivi operativi per l'IA, come leggere criticamente gli output generati e come decidere quando fare affidamento sui suggerimenti della macchina o quando, invece, correggerli.

Un aspetto ancora poco esplorato è l'effetto a lungo termine sull'identità professionale. Se l'IA scrive, crea, risponde, organizza, non resta spazio per la creatività e la competenza personale. Per evitare che queste ultime si appiattiscano, diventa allora essenziale capire non solo quali competenze sviluppare, ma anche come farlo in modo critico e consapevole.

L'IA è, dunque, destinata ad affiancare l'uomo, ampliandone le capacità e permettendogli di concentrarsi sulle attività più gratificanti, che richiedono un approccio umano, etico e flessibile.

In questo senso, la sfida principale non è quella di sostituire il lavoro umano, bensì di costruire una collaborazione uomo-macchina, dove la dignità professionale non venga erosa, bensì potenziata.

CAPITOLO 2

Il protocollo MCP

Nel presente capitolo verrà approfondito il Model Context Protocol (MCP), un protocollo che consente agli LLM di connettersi ai servizi esterni in modo standardizzato e uniforme. In primo luogo verrà analizzata la relazione tra tale protocollo e il meccanismo delle function calling, al fine di comprendere le ragioni per cui l'MCP può essere considerato un'evoluzione di quest'approccio. Successivamente verranno illustrati l'architettura e il workflow del protocollo, per poi concludere con una panoramica dei principali vantaggi che l'MCP introduce rispetto alle integrazioni custom sviluppate per i singoli casi d'uso.

2.1 Dal function calling alla necessità di uno standard

Negli ultimi anni si è assistito a una rapida evoluzione degli strumenti di Generative AI che hanno cambiato radicalmente il modo di interagire con le informazioni e la tecnologia.

Sebbene siano estremamente potenti nell'elaborare il linguaggio naturale, gli LLM operano su una base di conoscenza statica limitata dai dati di addestramento, e priva di un accesso diretto alle informazioni aggiornate o ai sistemi reali. Tale vincolo riduce il potenziale degli LLM nel fornire risposte pertinenti e accurate in diversi scenari applicativi.

Per superare tale limite, è stata introdotta la possibilità di connettere gli LLM ai servizi esterni, con lo scopo di renderli in grado di eseguire azioni nel mondo reale.

In tale contesto si inserisce il concetto di *function calling*, ovvero un processo attraverso il quale il modello genera chiamate strutturate verso funzioni, fornendo un "ponte" tra l'LLM stesso e i sistemi esterni. Tuttavia, questo approccio presenta una criticità. Ogni singola applicazione, infatti, necessita di un'integrazione custom, attraverso l'implementazione delle logiche di comunicazione, della gestione degli errori e del mapping tra il linguaggio naturale e i comandi applicativi; ciò aumenta la complessità e riduce la portabilità delle soluzioni.

Negli ultimi anni sono emersi numerosi framework con l'obiettivo di semplificare lo sviluppo di applicazioni basate sugli LLM e la loro interazione con l'ecosistema circostante. Tuttavia, anche in tal caso, ciascun framework adotta delle proprie modalità di comunicazione e rappresentazione dei dati, contribuendo alla creazione di un panorama frammentato e privo di standard condivisi.

È proprio in tale scenario che entra il gioco il Model Context Protocol (MCP), un protocollo che permette agli LLM di connettersi ai servizi esterni in modo uniforme. Definito da Anthropic e rilasciato come progetto open source nel novembre 2024, l'MCP può essere considerato l'"USB-C per l'Intelligenza Artificiale", ovvero un'interfaccia universale per la comunicazione tra i modelli e le applicazioni.

Il protocollo si basa sul concetto di function calling già introdotto nei modelli precedenti, ma ne estende le capacità, eliminando la necessità di sviluppare integrazioni personalizzate per ciascun caso d'uso.

Grazie all'utilizzo dell'MCP, dunque, diventa possibile realizzare applicazioni più modulari e sensibili al contesto, nelle quali un LLM può accedere alle risorse esterne, utilizzare gli strumenti e orchestrare i flussi di lavoro complessi in modo standardizzato.

2.1.1 La relazione tra il function calling e l'MCP

Prima di approfondire il funzionamento e l'architettura dell'MCP, risulta utile analizzare la relazione tra tale protocollo e il meccanismo delle function calling, al fine di comprendere le ragioni per cui l'MCP può essere considerato un'estensione di tale paradigma.

Inizialmente, il metodo più comune per consentire agli LLM di interagire con l'ambiente esterno era basato sulle function calling, spesso indicate anche come *tool call*. In tale approccio, le applicazioni espongono funzioni corredate da metadati descrittivi, quali nome, parametri e descrizione dello scopo, attraverso i quali il modello è in grado di "scoprire" e invocare tali funzioni estendendo, così, le proprie capacità di interazione con i servizi esterni.

In altre parole, il modello non si limita più a produrre contenuti testuali, ma può riconoscere lo strumento più appropriato in base alla richiesta dell'utente, e generare una tool call per eseguire un'operazione esterna, analogamente a quanto avviene nelle chiamate di funzione di un normale programma.

Per comprendere meglio il meccanismo del function calling, la Figura 2.1 mostra un esempio pratico in cui un utente chiede a un modello di fornire le previsioni meteo per una determinata città.

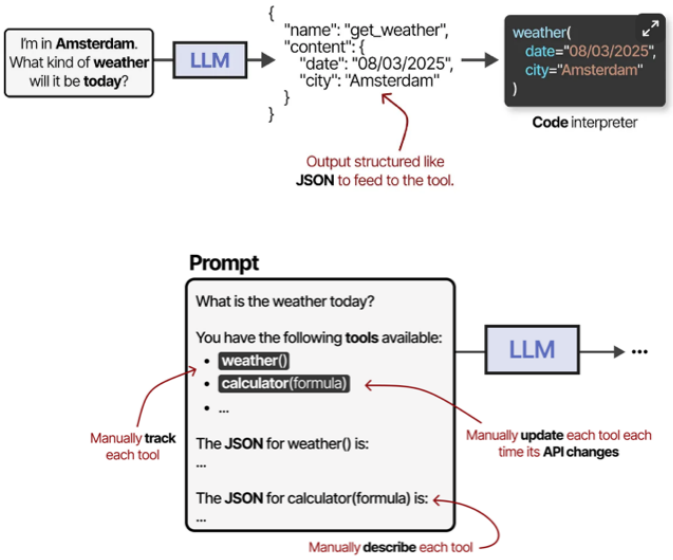


Figura 2.1: Esempio pratico del funzionamento del fuction calling

Il processo del function calling può essere schematizzato come segue:

- l'LLM riceve il prompt dell'utente, interpreta l'intento della richiesta e riconosce la necessità di ottenere dati meteorologici aggiornati, ma non può farlo direttamente perchè non dispone di un accesso nativo a tali informazioni.
- l'LLM seleziona lo strumento adatto in base alle descrizioni estratte dal contesto e genera una function call in formato strutturato (generalmente JSON), specificando

il nome della funzione e i parametri richiesti da essa. Una volta fatto ciò, il modello delega la responsabilità dell'esecuzione al sistema esterno di riferimento; quest'ultimo, successivamente, restituisce il risultato dell'operazione in modo strutturato al modello stesso.

- Infine, L'LLM interpreta il risultato e lo riformula in linguaggio naturale per renderlo facilmente leggibile dall'utente finale.

Le function call rappresentano, quindi, un importante passo in avanti, poichè permettono alle applicazioni di IA di superare i limiti del contesto statico, accedendo ai dati aggiornati e interagendo con i sistemi reali.

Il vero punto di forza di tale funzionalità risiede nella capacità del modello di decidere autonomamente quando e come utilizzare uno strumento, selezionando quello più adatto in base alla richiesta dell'utente, e di generare correttamente i parametri necessari alla chiamata di un determinato servizio esterno.

Tuttavia, tale approccio presenta diversi limiti pratici; in particolare:

- *La gestione del function calling richiede un intervento manuale*; ciascuna funzione, infatti, deve essere descritta nel system prompt specificando il nome, i parametri, il tipo di ritorno e la descrizione testuale del suo operato. Ogni modifica, quindi, deve essere apportata manualmente, rendendo il processo poco scalabile.

- *Il numero di strumenti utilizzabili è limitato*; poiché le descrizioni delle funzioni devono essere incluse nel contesto del modello, non è possibile definire un numero elevato di tool senza rischiare di saturare il prompt e ridurre, così, la qualità delle risposte.

- *Il modello è soggetto ad allucinazioni*; in presenza di molte funzioni o descrizioni ambigue, l'LLM può selezionare strumenti non pertinenti oppure generare parametri errati, compromettendo l'affidabilità del sistema.

- *Permane, in ogni caso, una mancanza di standardizzazione*; ciascun framework utilizzato per lo sviluppo di applicazioni di IA, come LangChain, Semantic Kernel o LlamaIndex, adotta modalità differenti per gestire le funzioni e le interfacce, dando origine a un ecosistema poco scalabile e difficilmente interoperabile.

Si delinea, quindi, un panorama sempre più frammentato che, invece di unificare le integrazioni tra i modelli e i servizi, rende la flessibilità, tanto ricercata, sempre più difficile da realizzare.

Tali criticità hanno portato alla necessità di un approccio più strutturato e scalabile, l'MCP, il quale ha introdotto un livello di astrazione standardizzato per consentire agli LLM di scoprire, descrivere e utilizzare gli strumenti, senza dipendere da implementazioni specifiche o da configurazioni manuali.

La Figura 2.2 illustra la principale differenza architetturale tra il function calling tradizionale e l'approccio basato sull'MCP.

Prima dell'introduzione del protocollo MCP, il modello interagiva con le API o gli strumenti esterni, ma per ciascuna interazione era necessario creare implementazioni specifiche. Ogni servizio richiedeva, infatti, i metodi di autenticazione, i formati di richiesta e le regole di comunicazione differenti.

Con l'introduzione dell'MCP, invece, le function call sono state astratte e standardizzate attraverso un livello intermedio di comunicazione. In tale scenario, l'LLM non si collega più direttamente a ciascun servizio esterno, ma comunica esclusivamente con il protocollo MCP, che funge da strato di orchestrazione capace di instradare le richieste verso le diverse API o i diversi strumenti disponibili.

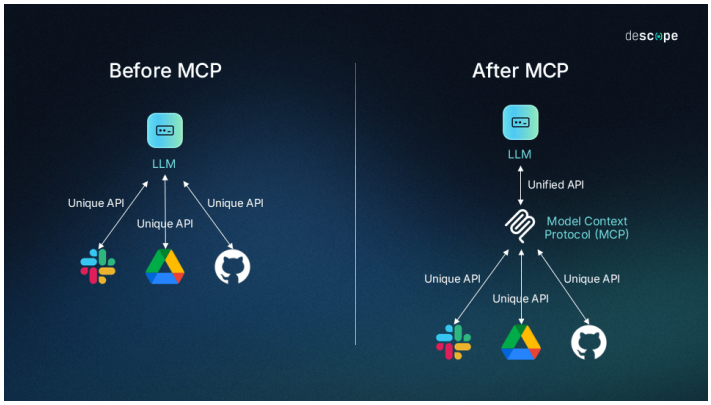


Figura 2.2: Differenza architetturale tra il function calling e l'MCP

In tal senso, il modello non deve conoscere i dettagli tecnici delle singole integrazioni, poichè è l'MCP stesso a gestire la connessione, la formattazione delle richieste e la comunicazione con i servizi sottostanti.

L'evoluzione architetturale rappresentata dall'MCP, dunque, non sostituisce il meccanismo del function calling, bensì lo estende introducendo uno strato di astrazione e standardizzazione, che ne semplifica la gestione e ne amplia le possibilità di integrazione.

2.2 L'architettura dell'MCP

L'MCP adotta un'architettura client-server, articolata intorno a tre principali attori, ovvero Host, Client e Server, ciascuno con un ruolo ben definito all'interno dell'ecosistema.

La Figura 2.3 illustra la struttura generale e i componenti del modello architetturale.

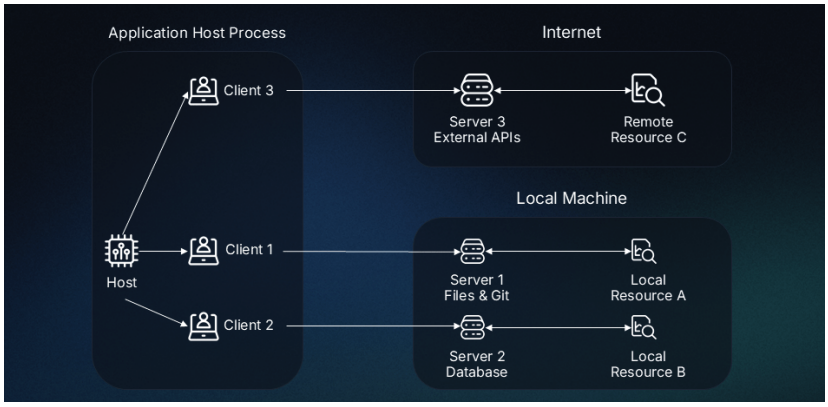


Figura 2.3: Architettura e componenti dell'MCP

Host MCP L'host è l'applicazione che ospita l'LLM e rappresenta l'interfaccia principale attraverso cui l'utente interagisce con il sistema. Esso è responsabile dell'avvio delle connessioni verso l'ecosistema MCP e della gestione del flusso di interazione con i server MCP. L'applicazione host esegue e orchestra l'utilizzo dei modelli di IA per generare la risposta finale destinata all'utente. Tra gli esempi di applicazioni host rientrano Claude Desktop, Gemini CLI e IDE integrati con funzionalità di IA, come Cursor.

Client MCP Il client MCP è il componente software che ha il compito di tradurre le richieste provenienti dall'interfaccia utente nel formato previsto dal protocollo MCP e di rielaborare i risultati generati dai server, affinché possano essere interpretati dal modello e strutturati nella risposta da fornire all'utente finale.

In altre parole, il client MCP funge da intermediario tra il modello linguistico e i servizi esterni, assicurando che i messaggi scambiati siano correttamente formattati secondo le specifiche del protocollo.

Dal punto di vista architetturale, host e client MCP sono, spesso, strettamente integrati e, in alcuni casi, possono anche coincidere in un'unica entità applicativa.

Il client MCP, inoltre, è responsabile della scoperta degli strumenti esposti dai server MCP e dell'orchestrazione tra le diverse componenti dell'ecosistema.

Server MCP Il server MCP costituisce il cuore operativo dell'MCP ed è il componente che espone, in maniera standardizzata, l'insieme degli strumenti e delle risorse esterne a cui il modello può accedere. Tra le operazioni tipiche offerte dai server MCP rientrano il recupero dei dati e l'accesso alle API esterne o ai servizi web.

In particolare, il server MCP ha il compito di gestire le richieste provenienti dai client e di recuperare le risposte restituite dai servizi esterni, garantendo che la comunicazione avvenga in maniera efficiente e affidabile, secondo le specifiche del protocollo.

In altri termini, se da un lato l'applicazione host, tramite il client, rappresenta l'interfaccia di utilizzo, i server MCP, dall'altro, sono i componenti che vengono sviluppati e messi a disposizione per fornire al modello le funzionalità necessarie al fine di elaborare le diverse richieste.

2.2.1 I servizi esposti dai server MCP

Una volta illustrate le principali componenti che costituiscono l'architettura dell'MCP, risulta utile analizzare i servizi che possono essere esposti dai server MCP. Tali funzionalità sono organizzate in tre categorie principali, come mostrato nella Figura 2.4.

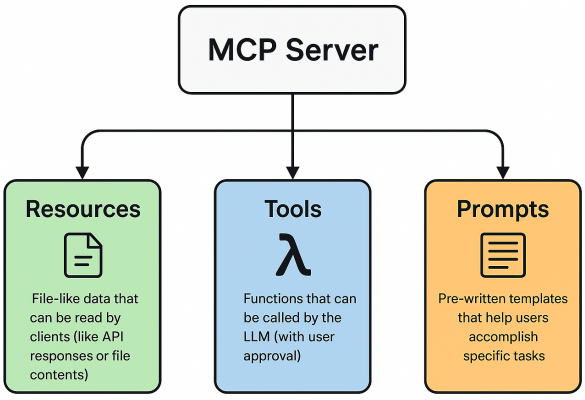


Figura 2.4: Servizi esposti dai server MCP

Tool I tool costituiscono il cuore del protocollo MCP; ogni tool è una funzione eseguibile registrata all'interno del server MCP e resa disponibile al modello tramite una descrizione standardizzata, che ne definisce il nome univoco, la funzione svolta e l'insieme di parametri di input e di output da essa previsti. Tali informazioni permettono al modello di comprendere

i parametri da fornire per l'invocazione del tool e il tipo di risultato da aspettarsi dalla sua esecuzione.

Risorse Le risorse rappresentano i contenuti utilizzabili sia dagli utenti che dai modelli di IA. Esse possono assumere diverse forme, tra cui:

- *i dati di contesto*, ovvero le informazioni utili al modello per prendere le decisioni o per svolgere determinate attività;
- *le basi di conoscenza o gli archivi documentali*, costituiti dalle raccolte di dati strutturati o non strutturati, come gli articoli, i manuali o le documentazioni tecniche;
- *i file locali o i database*, cioè i dati archiviati sui dispositivi o sui sistemi di gestione dei dati, accessibili per le attività di analisi;
- *le API e i servizi web*, che forniscono i dati o le funzionalità aggiuntive, permettendo l'integrazione con i servizi online.

Le risorse, inoltre, possono essere statiche o dinamiche. Nel primo caso, esse vengono fornite direttamente dall'utente come parte della richiesta (ad esempio un file o un documento allegato), per poi consentire al modello di utilizzarle come contesto, senza dover recuperare ulteriori informazioni tramite strumenti esterni. Nel secondo caso, invece, la risorsa può essere un output finale restituita dal server MCP come il risultato dell'esecuzione di una funzione espressa sotto forma di documento strutturato o link.

Prompt I prompt permettono ai server MCP di fornire i messaggi strutturati o le istruzioni predefinite, al fine di guidare l'interazione con i modelli linguistici. In tal senso, i prompt possono essere interpretati come dei workflow predefiniti, o degli schemi di conversazione standardizzati, che aiutano l'utente e il modello a svolgere determinate attività in modo più efficace e coerente.

Da un punto di vista concettuale, un prompt MCP è simile al prompt di un tradizionale LLM ma, in aggiunta, si hanno delle linee guida per lo svolgimento di determinati task che, in un certo senso, rappresentano le best practice, permettendo di ridurre l'ambiguità nell'interpretazione delle richieste.

In particolare, gli utenti possono esplorare i prompt messi a disposizione dai server, recuperarne il contenuto e, nel caso in cui essi siano parametrizzati, fornire valori specifici per personalizzarli. Infatti, i prompt possono includere i placeholder o le variabili che vengono riempite dinamicamente dal server in base agli argomenti forniti dall'utente, permettendo, così, di generare le istruzioni contestualizzate in tempo reale.

Tipicamente, tali prompt vengono attivati tramite alcuni comandi specifici a partire dall'interfaccia utente.

Tra i principali casi d'uso si possono individuare:

- *i template statici o gli slash command*, utilizzati, ad esempio, per l'avvio di una conversazione o per le operazioni di onboarding;
- *i template parametrizzati per i workflow comuni*, attraverso i quali il server espone le best practice operative che l'utente può personalizzare, fornendo valori specifici per i parametri richiesti;
- *i prompt personalizzati definiti direttamente dall'utente* all'interno di file dedicati, caricati localmente oppure resi disponibili tramite URI e, successivamente, passati come contesto al modello.

2.2.2 La comunicazione e il livello di trasporto dell'MCP

Una volta illustrato il ruolo dei diversi componenti dell'architettura, è necessario esaminare il meccanismo di comunicazione e il relativo livello di trasporto su cui si basa l'MCP.

La comunicazione tra il client e il server all'interno del protocollo si basa sullo standard JSON-RPC 2.0, un protocollo leggero e indipendente dal trasporto, che utilizza il formato JSON per rappresentare, in modo strutturato, i messaggi scambiati tra le diverse parti.

A tal proposito, il modello di comunicazione previsto da JSON-RPC si articola in tre principali tipologie di messaggi, ovvero:

- *request*, tramite cui il client MCP invia al server una richiesta, specificando il metodo da invocare e gli eventuali parametri associati;
- *response*, con cui il server MCP restituisce il risultato dell'operazione richiesta, oppure un eventuale messaggio di errore, mantenendo lo stesso identificatore della richiesta originaria;
- *notification*, ossia i messaggi unidirezionali che non prevedono una risposta; sono generalmente utilizzati dai server MCP per comunicare gli eventi o gli aggiornamenti al client MCP.

Al di sopra dello standard JSON-RPC, inoltre, il protocollo MCP introduce un livello di trasporto flessibile, che specifica le modalità con cui tali messaggi vengono effettivamente trasmessi tra client e server MCP. Il protocollo supporta due principali modalità di trasporto, a seconda del contesto in cui esse vengono eseguite; tali modalità sono::

- *STDIO (Standard Input/Output)*: utilizzata prevalentemente nelle esecuzioni locali, ad esempio in applicazioni desktop o in estensioni di ambienti di sviluppi (IDE), dove il client MCP e il server MCP vengono eseguiti sulla stessa macchina. L'applicazione host considera il server come un sotto-processo e comunica con esso scrivendo sul suo input standard (stdin) e leggendo dal suo output standard (stdout). I casi d'uso per tale trasporto sono i tool locali, come l'accesso al file system o l'esecuzione di script locali. I principali vantaggi sono la semplicità, la mancanza di una configurazione di rete e la sicurezza garantita dal sistema operativo.
- *HTTP + SSE (Server-Send Events)*: impiegata per le comunicazioni remote, dove il client e il server MCP possono operare su macchine differenti. La comunicazione avviene tramite HTTP, e il server invia gli aggiornamenti al client attraverso meccanismi di streaming basati su SSE. I casi d'uso per tale trasporto sono la connessione alle API remote, i servizi cloud, o le risorse condivise. I principali vantaggi sono il funzionamento tramite le reti, l'integrazione con i web service e la compatibilità con gli ambienti serveless.

2.2.3 Il workflow dell'MCP

L'ultimo aspetto che resta da analizzare riguarda il flusso di lavoro che caratterizza l'interazione tra i diversi elementi del protocollo MCP. In particolare, il ciclo di vita di una determinata richiesta si articola in più fasi sequenziali, che partono dall'input dell'utente e terminano con la generazione della risposta finale all'utente da parte del modello.

In generale, il processo complessivo è illustrato in Figura 2.5.

L'interazione ha inizio nel momento in cui l'utente interagisce con un'applicazione host, inserendo un prompt o uno slash command per l'avvio di una conversazione.

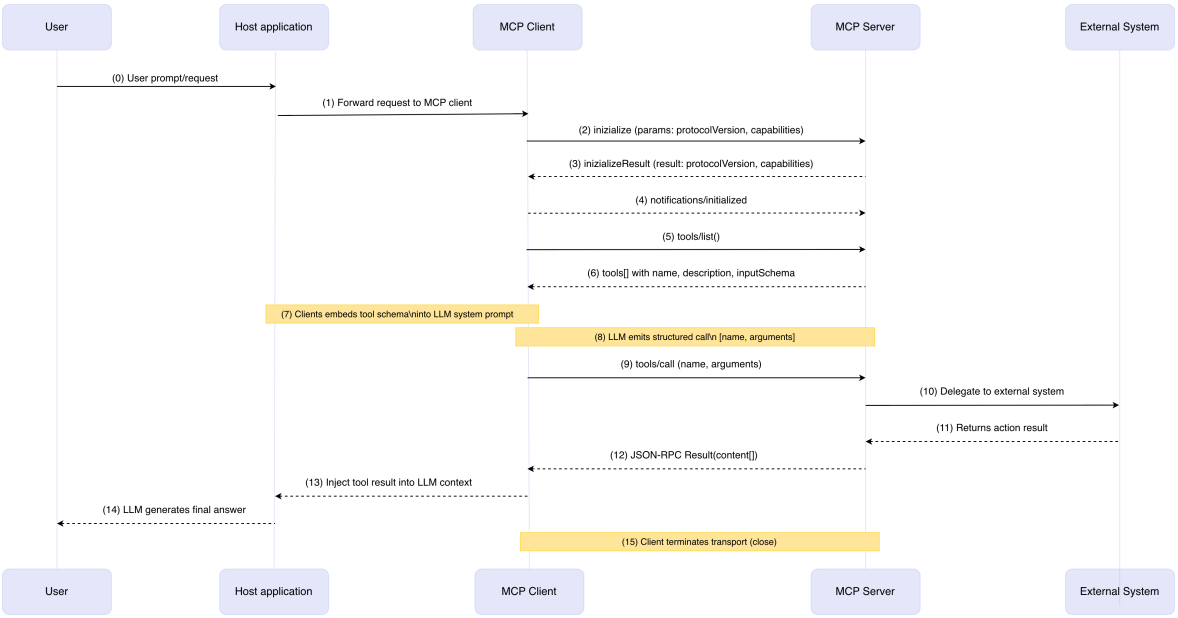


Figura 2.5: Workflow del protocollo MCP

Il client MCP raccoglie l'input e lo trasmette all'LLM integrato, il quale analizza la richiesta e ne deduce l'intento; qualora la risposta richieda l'accesso alle informazioni esterne o l'esecuzione di un'azione specifica, il modello può fare richiesta al server MCP per l'utilizzo degli strumenti disponibili.

Per fare ciò, innanzitutto il client inizializza una connessione con uno o più server MCP per mezzo di un protocollo di trasporto, STDIO oppure HTTP + SSE, a seconda del contesto applicativo.

Durante tale fase iniziale, tra client e server vengono inoltrate le informazioni relative alle versioni del protocollo supportate, ai tool, alle risorse disponibili ed agli eventuali parametri di autenticazione o di configurazione. Tale negoziazione preliminare consente di garantire che entrambi i componenti condividano le stesse specifiche operative e che la comunicazione possa avvenire in maniera sicura e coerente.

Terminata l'inizializzazione, il modello utilizza le descrizioni standardizzate dei tool fornite dal server MCP per determinare la funzione da invocare e i relativi parametri da fornire.

Una volta ricevuta la richiesta, il server MCP delega l'esecuzione della funzione al sistema esterno e restituisce, successivamente, il risultato dell'operazione al client nel formato JSON-RPC. Durante tale fase, inoltre, il server gestisce anche eventuali operazioni di autenticazione, di validazione dei parametri e di formattazione dei risultati, assicurando che l'output sia conforme alle specifiche del protocollo.

Infine, il client inoltra il risultato dell'operazione all'LLM, che lo utilizza come contesto aggiuntivo per generare la risposta finale in linguaggio naturale e presentarla all'utente attraverso l'interfaccia dell'applicazione host.

Una volta trasmessa la risposta all'utente finale, il client MCP interrompe la connesssione con il server MCP utilizzato per generare la risposta stessa.

Dunque, l'intero processo avviene in modo trasparente per l'utente e in maniera standardizzata per tutti i componenti coinvolti. Grazie alla natura modulare dell'architettura, host, client e server posssono essere sviluppati utilizzando framework differenti e possono operare in ambienti eterogenei mantenendo, al contempo, l'interoperabilità e la coerenza nel processo di comunicazione.

In altri termini, l'applicazione host ha il compito di incentrarsi sul dialogo e sul ragionamento in linguaggio naturale, il client MCP gestisce gli aspetti tecnici delle chiamate, e i server MCP si occupano dell'accesso ai dati e dell'esecuzione delle operazioni nei rispettivi domini applicativi; tale separazione delle responsabilità permette di mantenere una chiara distinzione dei ruoli.

2.2.4 I vantaggi nell'utilizzo dell' MCP

L'MCP si rivela particolarmente adatto a supportare lo sviluppo dei cosiddetti agenti AI, sistemi autonomi che, grazie alla gestione uniforme dei servizi esterni, possono individuare e utilizzare dinamicamante le funzionalità disponibili, al fine di pianificare e svolgere compiti complessi per soddisfare le richieste degli utenti.

L'adozione del protocollo MCP introduce numerosi benefici rispetto alle integrazioni custom sviluppate per ciascuna applicazione. In tale sezione vengono evidenziati i principali vantaggi che rendono quest'approccio particolarmente efficace nel contesto dell'integrazione tra i modelli linguistici e i servizi esterni.

Standardizzazione. L'MCP fornisce un'interfaccia comune per collegare gli LLM alle fonti di dati esterne, eliminando la necessità di implementare le interfacce proprietarie o le API differenti per ciascun sistema.

In assenza di uno standard condiviso, infatti, ogni integrazione richiede la gestione dei formati e delle modalità operative diverse. Con l'MCP, invece, tutte le integrazioni seguono uno schema uniforme, riducendo la complessità dell' implementazione e il rischio degli errori.

In questo modo, dunque, l'MCP introduce un linguaggio comune tra i sistemi di IA e i servizi applicativi, superando la frammentazione tipica delle integrazioni tradizionali.

Modularità e flessibilità. Grazie all'architettura unificata del protocollo, ogni fonte di dati e ogni servizio esterno vengono incapsulati in un modulo indipendente, rappresentato da un server MCP. Tale approccio plug-and-play consente di combinare facilmente più integrazioni, senza influenzare il funzionamento complessivo del sistema.

In altre parole, diversi server MCP possono essere attivati simultaneamente, permettendo ai sistemi di IA di scoprirli e utilizzarli attraverso lo stesso protocollo.

La modularità, inoltre, facilita la manutenzione, ovvero ogni connettore rimane isolato e può essere modificato senza compromettere gli altri componenenti.

Sicurezza e controllo. Un ulteriore vantaggio significativo dell'MCP riguarda l'attenzione alla sicurezza. Il protocollo, infatti, supporta i meccanismi di autenticazione e la gestione dei permessi, al fine di definire le operazioni che il modello è in grado di eseguire.

Durante la fase di negoziazione tra client e server, ad esempio, è possibile definire se l'accesso ad una determinata risorsa deve essere limitato alla sola lettura oppure consentire anche operazioni di modifica. Ciò permette di controllare con precisione il livello di accesso concesso agli agenti AI riducendo, così, il rischio di incidenti o di utilizzi impropri.

Inoltre, l'utilizzo di un unico layer di integrazione rende più semplice monitorare le operazioni effettuate dal sistema e centralizzare attività di logging e di audit. Tale aspetto risulta particolarmente rilevante in contesti regolamentati, come quelli finanziari o sanitari, dove il rispetto delle politiche di sicurezza e della conformità risultano fondamentali.

Riusabilità e interoperabilità. L'MCP è stato progettato per essere indipendente dall'LLM utilizzato. Il protocollo, infatti, può essere usato con diversi LLM, inclusi modelli proprietari e open-source, senza vincolare l'integrazione a una specifica piattaforma. Tale caratteristica riduce il rischio di vendor lock-in e consente di cambiare il modello o il provider, senza dover riprogettare le integrazioni esistenti.

I server MCP, una volta realizzati, possono essere riutilizzati in molteplici applicazioni e in contesti differenti, aumentando l'efficienza e preservando, nel tempo, l'investimento tecnologico.

Inoltre, la natura aperta del protocollo incentiva la creazione di nuovi server MCP, al fine di arricchire il catalogo delle integrazioni disponibili.

2.2.5 I rischi e le buone pratiche di sicurezza

La capacità dell'MCP di consentire agli LLM l'interazione con i servizi esterni introduce, inevitabilmente, nuove superfici d'attacco che devono essere gestite con particolare attenzione. L'adozione di tale protocollo, infatti, permette l'accesso alle funzionalità avanzate che permettono ai modelli di eseguire azioni operative nel mondo reale ma, allo stesso tempo, comporta dei rischi specifici legati alla sicurezza.

Uno dei rischi più rilevanti è rappresentato dalla *prompt injection*, ovvero la possibilità che il modello venga indotto a eseguire i comandi o le chiamate API non previste. A tal proposito, se il modello interpreta in modo errato le istruzioni ricevute potrebbe attivare delle funzionalità sensibili o eseguire delle operazioni non autorizzate.

Tale rischio non riguarda esclusivamente scenari esplicitamente malevoli, ma può manifestarsi anche in modo accidentale. Ad esempio, un utente potrebbe formulare una richiesta ambigua e il modello interpretarla in maniera errata, attivando un'azione non desiderata, anche se apparentemente coerente con l'intento espresso.

Analogamente, un prompt condiviso da terzi potrebbe contenere delle istruzioni nascoste o offuscate che inducono il modello a eseguire delle operazioni non previste dal progettista di sistema.

In scenari più critici un prompt, apparentemente innocuo, potrebbe portare il server MCP a effettuare delle operazioni sensibili, come l'accesso ai dati riservati o la modifica di configurazioni di sistema.

Per mitigare tali rischi, dunque, è necessario adottare una serie di buone pratiche di sicurezza.

In primo luogo, le azioni eseguite tramite il server MCP dovrebbero essere soggette ai meccanismi di validazione e, quando opportuno, a una conferma esplicita prima dell'esecuzione delle operazioni critiche.

In secondo luogo, è fondamentale definire politiche di autorizzazione rigorose che limitino l'accesso alle sole funzionalità necessarie.

L'obiettivo di tali misure non è limitare le potenzialità offerte dal protocollo, ma garantire che l'autonomia operativa del modello rimanga entro i confini sicuri e controllabili. Solo attraverso un'attenta progettazione dei meccanismi di sicurezza, infatti, è possibile sfruttare a pieno le capacità dell'MCP preservando, al tempo stesso, l'integrità, la tracciabilità e l'affidabilità dell'ambiente operativo.

CAPITOLO 3

Contesto applicativo e obiettivi del progetto

Questo capitolo presenta il contesto aziendale in cui si è svolto il progetto oggetto della presente tesi. Successivamente, verrà analizzato l'ambiente tecnologico di riferimento, con particolare attenzione ai sistemi ERP e ai web service REST che ne supportano l'integrazione con l'ambiente esterno. Infine, verranno definiti gli obiettivi che si intendono raggiungere con lo sviluppo della PoC basata sulle tecniche di Generative AI per l'accesso in linguaggio naturale ai web service di un ERP.

3.1 Contesto aziendale

Il presente progetto è stato sviluppato durante il periodo di tirocinio presso l'azienda Centro Software, una società di servizi informatici specializzata nello sviluppo di software gestionali di ultima generazione. Il lavoro svolto consiste nella realizzazione di un modello semantico in grado di consentire a un LLM di ricevere istruzioni in linguaggio naturale e di tradurle automaticamente in chiamate API, al fine di estrarre i dati dal sistema ERP e restituire i risultati in forma comprensibile per l'utente finale.

3.1.1 Centro Software

Centro Software è un'azienda che sviluppa soluzioni digitali per la gestione delle imprese, tra le quali rientrano sistemi ERP multilingua tra i più avanzati presenti sul mercato italiano.

Fondata alla fine degli anni '80 con sede principale a San Pietro in Casale (BO), l'azienda si è affermata come un punto di riferimento nella produzione di soluzioni gestionali italiane, successivamente esportate anche in mercati internazionali. Il logo della società è riportato nella Figura 3.1.

Figura 3.1: Logo dell'azienda Centro Software

L'azienda opera con filiali dirette su tutto il territorio nazionale e si rivolge principalmente alle realtà manifatturiere e industriali di medie dimensioni.

Le principali aree di competenza dell'azienda includono la produzione, il CRM, la logistica, l'amministrazione e il controllo di gestione. Inoltre, Centro Software offre servizi di consulenza e personalizzazione dei software, con particolare attenzione all'automazione dei processi, al controllo della qualità e alla gestione delle filiali estere.

3.2 Contesto tecnologico e stato dell'arte

Una volta delineato il contesto aziendale di riferimento, risulta necessario analizzare l'ambiente tecnologico in cui si inserisce il progetto. In particolare, verranno presentati il sistema ERP adottato dall'azienda e i relativi web service che supportano l'integrazione con i sistemi esterni, al fine di fornire una visione complessiva dell'infrastruttura su cui si basa il lavoro svolto.

3.2.1 Panoramica sugli ERP

Un Enterprise Resource Planning (ERP) indica un sistema informativo integrato che consente di gestire, in maniera efficiente e coordinata, i processi operativi e decisionali di un'organizzazione.

Gli ERP, nati inizialmente come strumenti dedicati alla gestione dei processi dell'area logistico-produttiva, sono diventati gradualmente sistemi integrati e modulari, in grado di coprire tutte le principali aree che possono essere automatizzate e monitorate all'interno di un'azienda. Tra queste rientrano le principali funzioni aziendali, tra i quali gli acquisti, la progettazione, la contabilità, la gestione dei magazzini e quella finanziaria.

Un elemento distintivo di tali sistemi è la capacità di trattare l'impresa come un sistema unitario, basato sui flussi informativi condivisi tra più dipartimenti, al fine di favorire un coordinamento efficace dei processi. Tale gestione delle informazioni è resa possibile dall'utilizzo di un database unificato per la condivisione dei dati tra le diverse aree aziendali, riducendo, così, il rischio di duplicazioni e di errori.

Le versioni più recenti, definiti ERP di seconda generazione (ERP2), integrano ulteriori funzionalità che, sfruttando le moderne tecnologie di comunicazione, permettono di interagire, in tempo reale, con la propria filiera produttiva e commerciale. Ciò ha favorito l'emergere di nuovi modelli di gestione dell'impresa, nei quali l'organizzazione risulta fortemente integrata con il mondo esterno.

3.2.2 Il sistema ERP aziendale

Negli ultimi anni, accanto ai principali ERP sviluppati a livello internazionale, si è assistito alla crescita di soluzioni progettate per il contesto italiano. Tali sistemi si distinguono per la capacità di adattarsi alle normative fiscali e contabili nazionali e per una maggiore aderenza al modus operandi locale, risultando particolarmente idonei alle esigenze delle piccole e medie imprese.

In questo scenario, si inserisce l'ERP proprietario dell'azienda Centro Software, progettato per supportare, in modo integrato, le diverse attività aziendali e per rispondere alle esigenze di interoperabilità sia interna che verso l'ambiente esterno. In particolare, il sistema consente di gestire i flussi informativi tra i diversi reparti e di facilitare l'integrazione con i soggetti esterni all'impresa, tra i quali venditori, consulenti e partner della filiera.

Il sistema ERP adottato dall'azienda è SAM ERP2, un sistema ERP multi-lingua di ultima generazione progettato per supportare l'intero ecosistema aziendale (Figura 3.2).

Figura 3.2: Logo del software SAM ERP2

Il sistema consente la gestione automatizzata dei processi interni e l'interazione diretta con gli impianti produttivi in ottica *Industry 4.0*, nonché la comunicazione in tempo reale con la filiera produttiva e commerciale a livello globale.

Si tratta, quindi, di una piattaforma unificata, frutto di un progetto costantemente aggiornato, che integra le migliori pratiche industriali italiane con le tecnologie più avanzate.

Uno degli elementi distintivi di SAM ERP2 è la capacità di migliorare l'efficienza operativa e la produttività aziendale attraverso l'ottimizzazione dei flussi informativi e decisionali. Il sistema consente, ad esempio, di personalizzare le interfacce utente in base al ruolo dell'operatore, permettendo la visualizzazione in tempo reale dei principali indicatori di performance (KPI). Tali informazioni possono derivare sia dai dati interni al sistema che da fonti esterne, come i feed informativi e i dati di mercato.

SAM ERP2 è strutturato in diverse aree gestionali, ciascuna composta da specifici moduli funzionali (Figura 3.3). In particolare, tali aree sono:

Figura 3.3: Moduli funzionali di cui è composto SAM ERP2

- *Area AI e Workflow*: gestisce i processi aziendali integrando l'automazione e gli algoritmi di IA, al fine di fornire una base di dati condivisa e strumenti di monitoraggio avanzati.
- *Area CRM, marketing e servizi*: supporta la gestione delle relazioni con i clienti acquisiti e i prospect, coprendo le attività di pre-vendita, post-vendita e assistenza.
- *Area commerciale*: gestisce l'intero ciclo di vendita, dalla definizione dell'offerta fino all'evasione dell'ordine, integrando strumenti di pianificazione e di analisi.
- *Area acquisti, magazzini e logistica*: ottimizza i rapporti con i fornitori, la gestione delle scorte e la movimentazione delle merci.
- *Area produzione*: pianifica e controlla i processi produttivi, sia interni che esterni, con l'obiettivo di ridurre i tempi e i costi operativi.
- *Area qualità*: gestisce i controlli sui materiali e sui prodotti, includendo le procedure di collaudo e di valutazione dei fornitori.

- *Area amministrazione e gestione finanziaria*: gestisce le funzioni amministrative e finanziarie, garantendo precisione e conformità dei fornitori.
- *Area controllo di gestione*: fornisce strumenti avanzati di analisi, tra i quali la contabilità analitica, la Business Intelligence e il Data Mining, al fine di consentire ai quadri direttivi di prendere decisioni strategiche.
- *Area intercompany e comunicazioni*: abilita l'integrazione con i sistemi esterni e la comunicazione lungo tutta la filiera produttiva.
- *Area sicurezza e strumenti di sviluppo*: garantisce il controllo degli accessi, la protezione dei dati e la possibilità di estendere il sistema tramite strumenti di sviluppo e di personalizzazione.

3.2.3 Panoramica sui Web Service a supporto dell'ERP

Una volta illustrato il sistema SAM ERP2 e le sue principali caratteristiche, è utile analizzare le tecnologie di comunicazione che consentono di interagire con l'ambiente esterno. In tale contesto, un ruolo centrale è svolto dai Web Service, sistemi software progettati per supportare l'interazione tra le applicazioni distribuite attraverso uno standard di comunicazione.

Inizialmente, i Web Service erano basati sul protocollo SOAP (Simple Object Access Protocol), che utilizzava il formato XML per lo scambio di messaggi allo scopo di invocare i servizi remoti.

Con l'evoluzione delle architetture distribuite, si è progressivamente affermato l'approccio REST (Representational State Transfer), ispirato ai principi architetturali del web e basato sull'uso diretto del protocollo HTTP. In tale paradigma, le risorse remote sono identificate tramite URI (Uniform Resourse Identifiers), ovvero identificatori univoci che ne descrivono la struttura, e sono manipolate attraverso i metodi HTTP standard, quali GET, POST, PUT e DELETE, che corrispondono alle operazioni CRUD (Figura 3.4).

A differenza di SOAP, REST non impone formati rigidi per lo scambio dei dati, favorendo l'utilizzo di formati leggeri come JSON, e risultando, quindi, più flessibile, scalabile, ed efficiente.

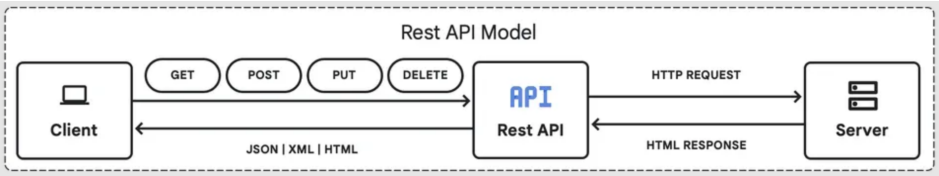


Figura 3.4: Funzionamento delle API REST

All'interno del sistema SAM ERP2, i Web Service REST costituiscono il principale strumento per esporre le funzionalità applicative verso l'esterno. Attraverso tali API RESTful il sistema rende disponibili, in modo strutturato, le proprie risorse ed espone i dati verso l'esterno, consentendo sia la comunicazione verticale tra i diversi livelli dell'organizzazione sia l'integrazione trasversale con i sistemi eterogenei lungo l'intera catena di supply-chain. Tale approccio permette di realizzare un'integrazione scalabile, indipendente dall'interfaccia utente e facilmente eseguibile in nuovi scenari applicativi.

L'insieme delle API REST è documentato nello Swagger del sistema SAM che descrive, in modo completo, la struttura e il funzionamento delle singole API. Nella Figura 3.5 è mostrata la schermata dello Swagger, tramite cui è possibile consultare tutti gli endpoint disponibili, i parametri richiesti e le strutture dei dati in input e output.

Tale documentazione, dunque, rappresenta un elemento fondamentale per l'utilizzo e l'integrazione dei servizi offerti dall'ERP.

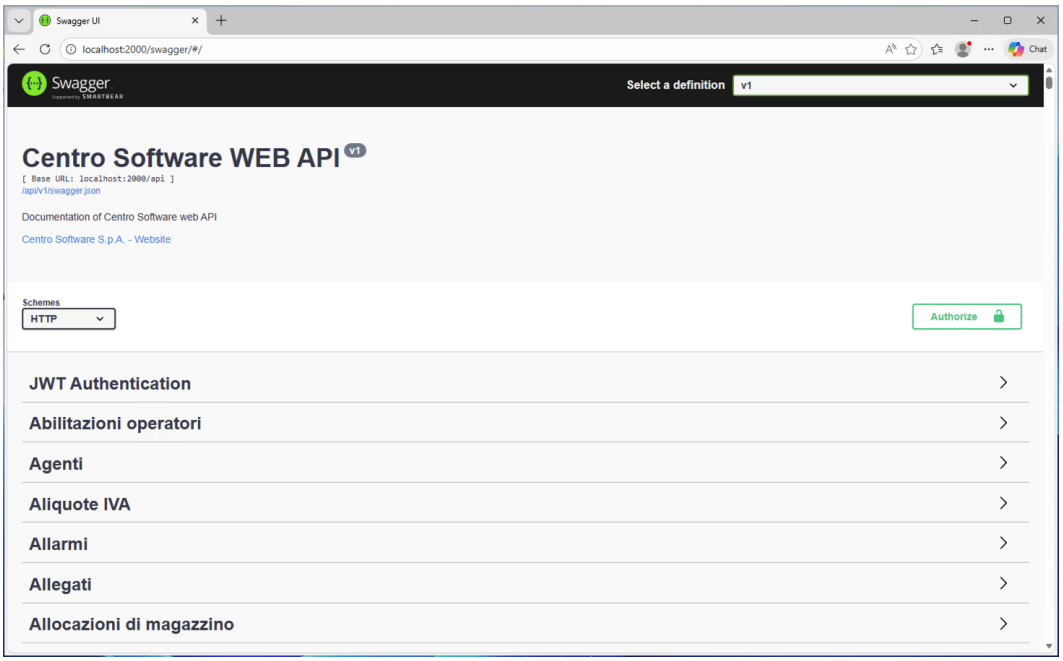


Figura 3.5: Schermata dello Swagger del sistema SAM ERP2

3.3 Motivazioni e obiettivi del progetto

L'evoluzione dei sistemi informativi aziendali e la crescente diffusione delle tecnologie di IA Generativa stanno aprendo nuove prospettive per l'interazione tra l'utente e i sistemi gestionali.

L'idea del progetto è quella di utilizzare un LLM come intermediario tra l'utente e il sistema informativo. In particolare, esso si propone di interfacciare un LLM con i Web Service esposti dal sistema SAM ERP2, con l'obiettivo di consentire agli utenti di interrogare il sistema utilizzando il semplice linguaggio naturale, senza la necessità di conoscere la struttura tecnica sottostante.

In uno scenario tipico, l'utente formula una richiesta in linguaggio naturale e l'LLM la traduce automaticamente in una richiesta tecnica eseguibile sull'ERP. Una volta eseguita la funzione, l'LLM elabora il risultato e lo restituisce all'utente in modo comprensibile. Tale approccio semplifica l'accesso alle informazioni e riduce la dipendenza dalle competenze tecniche specifiche.

Una delle principali motivazioni tecniche per l'uso dei Web Service riguarda i limiti dell'interrogazione diretta del database tramite query SQL. L'approccio con query SQL, infatti, garantisce un elevato grado di flessibilità, ma richiede una conoscenza approfondita della struttura del database, che può risultare complessa, soggetta a modifiche nel tempo e fortemente dipendente dall'implementazione specifica del sistema. Inoltre, molte informazioni all'interno di un ERP non derivano da semplici operazioni di lettura, ma sono il risultato di logiche di business articolate, replicabili attraverso query complesse.

Per superare tali criticità, il progetto si orienta verso l'utilizzo dei Web Service REST esposti dall'ERP, i quali incapsulano le logiche applicative e restituiscono i dati già elaborati. Tale approccio consente di astrarre dalla complessità del database sottostante e di garantire

maggiore stabilità rispetto a eventuali modifiche strutturali del sistema. Tuttavia, l'interrogazione delle API introduce un nuovo vincolo, ovvero le richieste devono rispettare i formati e i tracciati ben definiti, limitando la flessibilità rispetto all'uso diretto di SQL.

Per conciliare flessibilità e robustezza, il progetto introduce un modello semantico intermedio, il cui ruolo è quello di tradurre le richieste in linguaggio naturale in chiamate API strutturate. In particolare, il compito del modello è quello di identificare l'intento dell'utente, di estrarre i parametri rilevanti e di generare automaticamente la richiesta tecnica nel formato richiesto dall'ERP, tipicamente JSON. Ciò consente di separare il livello linguistico da quello tecnico, rendendo il sistema più modulare e adattabile.

Un aspetto chiave di tale modello è la capacità di essere addestrato o parametrizzato sulla base della struttura delle API disponibili, piuttosto che sul contenuto specifico dei dati. Ciò consente di ottenere un elevato grado di generalizzazione: il sistema può essere adattato a differenti basi di dati o, addirittura, a diversi ERP, semplicemente aggiornando le regole semantiche che governano la trasformazione delle richieste. In questo senso, il modello non si limita a effettuare una traduzione linguistica, ma costruisce una rappresentazione semantica del dominio applicativo.

Il flusso operativo, dunque, prevede che l'utente formuli una richiesta in linguaggio naturale, l'LLM ne interpreti il significato, il modello semantico la trasformi nella chiamata API corretta e, infine, l'ERP esegua la richiesta restituendo il risultato all'utente stesso.

Tale paradigma apre la strada a nuove modalità di interazione con i sistemi gestionali, rendendoli più accessibili, flessibili e orientati all'utente finale.

CAPITOLO 4

Analisi dei requisiti

In questo capitolo viene presentata l'analisi dei requisiti del sistema sviluppato, con l'obiettivo di definire le funzionalità attese e i principali vincoli progettuali.

In una prima fase, verranno approfonditi i web service REST a supporto dell'ERP, analizzandone la struttura e le modalità di invocazione.

Successivamente, verranno esaminati i principali limiti dell'utilizzo diretto delle API, soprattutto in relazione all'interazione da parte di utenti non tecnici.

In seguito, verranno illustrate le motivazioni alla base delle scelte tecnologiche adottate, con particolare riferimento al protocollo MCP e alla libreria Python FastMCP.

Infine, saranno formalizzati i requisiti funzionali e non funzionali del progetto, delineando gli obiettivi principali che si intendono raggiungere con lo sviluppo della PoC basata sulle tecniche di Generative AI per l'accesso in linguaggio naturale ai web service di un ERP.

4.1 Descrizione dei web service REST

Dopo aver delineato l'organizzazione generale del sistema ERP risulta opportuno soffermarsi sulle caratteristiche peculiari dei web service REST che lo supportano, con una particolare attenzione alla loro struttura e alle modalità di utilizzo.

I Web Service sono documentati secondo la specifica Swagger 2.0, un framework open source che facilita la progettazione, la realizzazione e l'utilizzo delle API REST in un contesto ordinato. In particolare, le operazioni esposte sono organizzate per categoria di risorsa e, per ognuna di esse, viene fornito un insieme di funzioni eseguibili attraverso endpoint HTTP. Ciascuna operazione, inoltre, è corredata da informazioni dettagliate sui parametri richiesti e sui vincoli di validità dei parametri coinvolti fornendo, quindi, una struttura chiara e facilmente interpretabile.

A titolo esemplificativo, in Figura 4.1 è mostrato un esempio di organizzazione delle API strutturate per risorsa, all'interno delle quali è presente l'elenco delle operazioni disponibili.

Da un punto di vista formale, la struttura dei dati scambiati è definita tramite schemi JSON, i quali specificano, in modo rigoroso, sia i corpi delle richieste (*request body*) sia quelli delle risposte (*response body*), contribuendo, così, a una descrizione completa delle interfacce esposte.

I Web Service, pertanto, non rappresentano soltanto un semplice punto di accesso ai dati, ma costituiscono un vero e proprio livello di astrazione delle logiche applicative. Essi, infatti, offrono operazioni coerenti con il dominio operativo dell'ERP, rendendolo più semplice da utilizzare e da integrare con i servizi esterni.

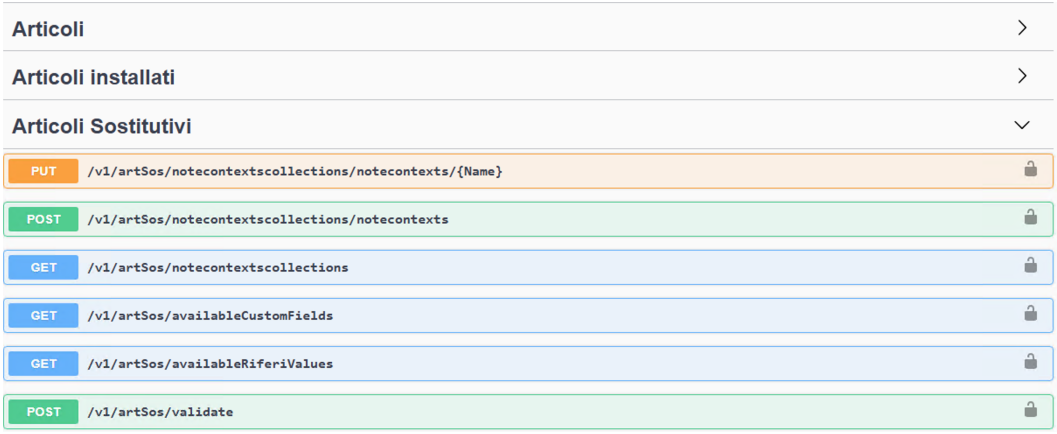


Figura 4.1: Esempio dell'organizzazione delle API per categoria di risorsa

4.1.1 Struttura delle API REST

L'analisi della struttura e del funzionamento delle API rappresenta un aspetto centrale del progetto, poichè costituisce la base su cui si fonda l'intero processo di generazione delle chiamate API e delle relative risposte.

In particolare, lo studio preliminare delle modalità con cui le API sono organizzate, descritte e invocate si è rivelato un passaggio essenziale per affrontare il problema progettuale, al fine di definire una soluzione coerente con gli obiettivi prefissati. La capacità di tradurre una richiesta espressa in linguaggio naturale in una chiamata API corretta presuppone, infatti, una conoscenza accurata della struttura dei servizi disponibili e dei vincoli imposti dai parametri coinvolti.

In generale, quando si vuole interagisce con un'API REST, è necessario seguire alcuni passaggi essenziali:

- definire l'endpoint, ovvero l'indirizzo della risorsa a cui si vuole accedere;
- specificare i parametri richiesti, ovvero le informazioni necessarie per eseguire la richiesta;
- inviare la richiesta tramite il metodo HTTP appropriato.

A seguito della richiesta, il sistema restituisce una risposta strutturata contenente i dati richiesti, spesso già elaborati secondo le logiche proprie dell'ERP. Tale approccio consente, dunque, di ottenere le giuste informazioni evitando, al tempo stesso, sia l'accesso diretto al database che l'implementazione manuale delle logiche applicative.

Esempio applicativo A titolo esemplificativo, si consideri l'operazione *GetSingleArticoli*, che consente di recuperare le informazioni relative a un singolo articolo a partire dalla definzione del suo identificativo univoco. Lo schema JSON dell'API di riferimento è riportato nel Listato 6.1.

```
  "paths":
  {
    "\/v1\/artico\/{Id}":
    {
      "get":
      {
        "tags":
        [
          "Articoli"
        ]
,
```

```
        "description":"Restituisce il record di Articoli corrispondente all'Id passato come parametro",
        "operationId":"GetSingleArticoli",
        "produces":
        [
          "application\/json"
        ]
,
        "parameters":
        [
                {
           "in":"path",
           "name":"Id",
           "description":"Id Articoli",
           "required":true,
           "type":"integer"
          }
        ]
,
        "responses":
        {
          "200":
          {
            "description":"OK",
            "schema":
            {
              "$ref":"#\/definitions\/ArticoModel"
            }
            }
            }



,
        "security":
        [
                {
           "bearer":
           [
                     ]
          }
        ]
      }
    }
    }
"definitions":
{
   "ArticoModel":
   {
     "type":"object",
     "required":
     [
       "id",
       "codice",
       "descr",
       "staart",
       "tipart"
     ]
,
     "properties":
     {
       "id":
       {
         "type":"integer",
         "description":"Identificatore articolo"
       }
,
       "codice":
       {
         "type":"string",
         "description":"Codice"
       }
,
       "descr":
       {
         "type":"string",
         "description":"Descrizione"
       }
,
       "staart":
       {
         "type":"string",
         "description":"Stato dell'articolo"
       }
,
       "tipart":
       {
         "type":"string",
         "description":"Tipo articolo"
       }
       }
       }
```

Listato 4.1: Struttura dell'API

Per invocare tale API, è necessario utilizzare il metodo HTTP GET e specificare il parametro obbligatorio opportuno, ovvero l'identificatore univoco dell'articolo da recuperare, espresso come valore intero.

In caso di esito positivo, il servizio restituisce un oggetto in formato JSON conforme allo schema *ArticoModel* definito all'interno del Listato 6.1. Tale modello rappresenta, dunque, la struttura dell'oggetto restituito e include un insieme di campi obbligatori e facoltativi che descrivono le principali caratteristiche dell'articolo. Tra i campi obbligatori si distinguono:

- *id*: identificatore dell'articolo;
- *codice*: codice identificativo;
- *descr*: descrizione dell'articolo;
- *tipart*: tipologia dell'articolo;
- *staart*: stato dell'articolo.

4.1.2 Limiti dell'utilizzo diretto delle API

Nonostante i molteplici vantaggi offerti dai Web Service REST, il loro utilizzo diretto presenta alcune difficoltà rilevanti, soprattutto quando l'interazione coinvolge utenti non tecnici.

Come evidenziato in precedenza, per interagire con le API è necessaria una buona conoscenza dello loro struttura. A tal proposito, risulta essenziale conoscere l'endpoint corretto da utilizzare, gli eventuali parametri da fornire e il corrispettivo formato, oltre ad essere in grado di interpretare correttamente la risposta restituita dal sistema. Risulta chiaro che tutto ciò implica una certa familiarità con la documentazione e con i meccanismi tecnici propri delle API REST che non sono sempre accessibili a tutti gli utenti.

Un ulteriore aspetto riguarda la rigidità del modello di interazione. Le API, infatti, espongono un insieme finito di operazioni, definite secondo tracciati ben precisi. Le richieste, quindi, devono aderire strettamente a tali strutture e non è possibile discostarsi dal formato previsto, né formulare interrogazioni libere, come avverrebbe nel linguaggio naturale. In tal senso, l'utilizzo diretto delle API risulta meno intuitivo e maggiormente vincolato.

A ciò si aggiunge un limite legato alla scalabilità funzionale: se un utente formula una richiesta che non trova una corrispondenza diretta in uno degli endpoint disponibili, il sistema non è in grado di fornire una risposta. La capacità espressiva dell'interazione, quindi, resta confinata all'interno delle operazioni previste dall'API, senza possibilità di un adattamento dinamico.

Infine, la presenza di un elevato numero di endpoint e di operazioni può rendere difficile anche trovare l'API corretta per soddisfare una determinata esigenza, soprattutto se non si ha una buona conoscenza del dominio applicativo.

Nel complesso, se da un lato tale rigidità garantisce coerenza e affidabilità nell'accesso ai dati, dall'altro rappresenta un limite significativo quando si desidera un'interazione più naturale e immediata.

Da qui nasce l'esigenza di introdurre un livello intermedio capace di mediare tra il linguaggio dell'utente e le specifiche tecniche delle API, traducendo automaticamente le richieste in chiamate API strutturate.

4.2 Scelte tecnologiche

4.2.1 Adozione del protocollo MCP

Le limitazioni nell'utilizzo diretto delle API evidenziano la necessità di introdurre un livello di astrazione intermedio, capace di colmare il divario tra il linguaggio naturale utilizzato dagli utenti e la struttura tecnica richiesta dai servizi esposti. In tale contesto, l'adozione dell'MCP si rileva essere una scelta particolarmente efficace.

Uno degli aspetti più rilevanti dell'MCP è la capacità di standardizzare l'interazione tra l'LLM e gli strumenti esterni. Ciò significa che non è necessario implementare manualmente la logica per invocare le singole API, in quanto il protocollo consente di descrivere le funzionalità disponibili in modo strutturato e uniforme, rendendole direttamente accessibili al modello di linguaggio.

Un ulteriore vantaggio è la semplificazione nello sviluppo applicativo. In assenza di un protocollo come MCP, l'integrazione tra l'LLM e le API richiederebbe la gestione esplicita di diversi aspetti, tra cui il mapping tra il linguaggio naturale e i parametri tecnici, la selezione degli endpoint e la gestione delle possibili eccezioni. L'introduzione dell'MCP, invece, permette di centralizzare tali responsabilità, rendendo l'architettura più chiara e modulare.

Dal punto di vista della manutenibilità, tale approccio si rileva particolarmente efficace. Eventuali modifiche alle API, come l'aggiunta di nuovi endpoint o la variazione dei parametri, possono essere gestite semplicemente aggiornando la relativa descrizione, senza richiedere modifiche alla logica del modello; ciò rende il sistema più flessibile e meglio adattabile all'evoluzione del contesto applicativo.

Infine, l'adozione dell'MCP consente di superare uno dei principali limiti dell'interazione diretta con le API, ovvero la loro rigidità strutturale. Sebbene le API impongono vincoli stringenti sui formati e sui tracciati delle richieste, il protocollo introduce un livello intermedio che permette di preservare la naturalezza dell'interazione garantendo, al contempo, la correttezza delle chiamate tecniche.

In tale prospettiva, l'MCP può essere interpretato come un collegamento tra il dominio linguistico dell'utente e quello operativo del sistema ERP, offrendo una soluzione scalabile e coerente per l'integrazione tra modelli di IA e sistemi informativi aziendali.

4.2.2 Utilizzo del framework FastMCP

Da un punto di vista implementativo, è stato scelto di utilizzare il framework FastMCP, una libreria pensata per semplificare la creazione di server MCP e l'esposizione di tool utilizzabili da un LLM.

Uno dei principali vantaggi offerti da FastMCP è la capacità di semplificare la complessità legata all'implementazione del protocollo. Il protocollo, infatti, è dotato di un'interfaccia ad alto livello che consente di definire le funzioni che possono essere invocate dall'LLM. In questo modo, le API REST esposte dal sistema ERP possono essere facilmente mappate in strumenti MCP, senza dover gestire manualmente tutti i dettagli di integrazione.

Un aspetto particolarmente rilevante è la possibilità di integrare specifiche OpenAPI; ciò consente di importare automaticamente la descrizione delle API e di trasformarla in un insieme di funzioni direttamente utilizzabili dal modello. Di conseguenza, si riduce lo sforzo necessario per l'implementazione, evitando la necessità di definire manualmente ciascun endpoint e garantendo una maggiore coerenza tra la documentazione delle API e il comportamento del sistema.

L'utilizzo di FastMCP, quindi, si inserisce in modo naturale nell'architettura di riferimento, poichè consente di realizzare rapidamente un'infrastruttura in cui le API dell'ERP diventano

strumenti direttamente utilizzabili dall'LLM. Inoltre, tale approccio favorisce la modularità e la manutenibilità del sistema e ne semplifica l'evoluzione, rendendo più facile l'estensione a nuovi servizi o l'adattamento a eventuali modifiche delle API.

4.3 Vincoli tecnici

4.3.1 Compatibilità delle specifiche API

Nel corso dell'analisi dei requisiti, è stato rilevato un vincolo tecnico importante legato al modo in cui sono documentate le API REST del sistema ERP. In particolare, tali API sono descritte secondo lo standard Swagger 2.0, mentre il framework FastMCP richiede specifiche nel formato OpenAPI 3.x.

Sebbene i due standard siano concettualmente simili, esistono differenze strutturali che non permettono di utilizzarli in modo intercambiabile. Ad esempio, il modo in cui vengono descritte le richieste, le risposte e i parametri è diverso, e questo ha delle ripercussioni sulla compatibilità con lo strumento MCP.

Tale discrepanza introduce un vincolo progettuale significativo: la documentazione delle API non può essere utilizzata direttamente come base per l'integrazione con l'LLM, ma deve essere dapprima adattata al formato previsto dagli strumenti utilizzati.

Dunque, già in fase di analisi dei requisiti, risulta evidente la necessità di garantire che le specifiche delle API siano compatibili con le tecnologie utilizzate.

4.4 Definizione dei requisiti

Dopo aver analizzato il contesto applicativo e le caratteristiche dei Web Service REST, è possibile individuare le funzionalità attese e i vincoli che lo sviluppo del sistema deve rispettare. In particolare, i requisiti vengono distinti in requisiti funzionali, relativi ai comportamenti e alle funzionalità del sistema, e in requisiti non funzionali, che riguardano, invece, gli aspetti qualitativi e prestazionali

4.4.1 Requisiti funzionali

Il progetto ha come obiettivo la realizzazione di una PoC (Proof of Concept) basata sulle tecniche di Generative AI per l'accesso in linguaggio naturale ai web service di un ERP.

In primo luogo, il sistema deve consentire agli utenti di formulare richieste in linguaggio naturale, senza dover conoscere necessariamente la struttura tecnica delle API o del database sottostante.

Successivamente, il sistema deve essere in grado di interpretare tali richieste, identificando correttamente l'intento dell'utente e le informazioni rilevanti, come parametri, entità e vincoli.

A partire dall' interpretazione della richiesta, il sistema deve generarare automaticamente una chiamata API conforme alle specifiche del sistema ERP, rispettando i formati e i tracciati previsti.

Un ulteriore requisito fondamentale è la capacità di selezionare l'endpoint più appropriato tra quelli disponibili, sulla base della richiesta dell'utente.

Il sistema deve, inoltre, essere in grado di gestire l'invocazione delle API REST, ricevere la risposta e restituirla all'utente in forma comprensibile, eventualmente rielaborando i dati, al fine di migliorarne la leggibilità.

Infine, esso deve essere in grado di estendere le funzionalità a nuovi endpoint, senza richiedere modifiche sostanziali alla logica di funzionamento.

4.4.2 Requisiti non funzionali

Dal punto di vista tecnico, per soddisfare i requisiti funzionali, sono stati definiti i vincoli e le caratteristiche qualitative che il sistema deve rispettare per risultare efficace all'interno del contesto aziendale.

Un primo aspetto riguarda l'usabilità, ovvero il sistema deve offrire un'interazione semplice e intuitiva, permettendo all'utente di accedere alle informazioni senza competenze tecniche specifiche. In tale contesto, la qualità dell'interpretazione del linguaggio naturale rappresenta un elemento cruciale.

Un ulteriore requisito riguarda la modularità e la manutenibilità del sistema. L'architettura, infatti, deve essere progettata in modo da separare il livello linguistico da quello tecnico, consentendo aggiornamenti indipendenti delle API o dei modelli senza impatti significativi sull'intero sistema.

Inoltre, il sistema deve garantire tempi di risposta adeguati, compatibili con un utilizzo interattivo; ciò implica una gestione efficiente delle chiamate API e dei processi di elaborazione delle richieste.

Infine, è necessario garantire la scalabilità e l'estensibilità della soluzione, in modo da poter supportare un numero crescente di endpoint e scenari applicativi, nonché l'eventuale integrazione con altri sistemi ERP o fonti di dati.

CAPITOLO 5

Progettazione dell'ecosistema

Nel presente capitolo verranno illustrate le scelte progettuali adottate per la realizzazione dell'ecosistema in grado di integrare un modello linguistico con i web service esposti dal sistema ERP aziendale tramite il protocollo MCP. A partire da un'iniziale visione architetturale ad alto livello, verranno progressivamente introdotti e approfonditi i componenti che costituiscono il sistema. Tra questi rientrano l'LLM, responsabile dell'interpretazione semantica delle richieste dell'utente, e il server MCP, che funge da livello di orchestrazione tra il dominio linguistico e quello applicativo. Successivamente verrà descritta la strategia di integrazione dell' API REST all'interno del sistema, con particolare attenzione alla generazione dinamica dei tool a partire dalla specifica OpenAPI e alla progettazione di un'architettura basata su macro-tool, pensata per ridurre la complessità esposta all'LLM. Inoltre, verranno trattati gli aspetti legati all'autenticazione per l'accesso all'ERP e al processo di validazione e testing delle API tramite Postman. Nella parte finale verrà mostrato il flusso operativo dell'intero ecosistema, descrivendo come la richiesta dell'utente venga tradotta in un'azione concreta sul sistema ERP e come il risultato dell'esecuzione dell'API venga trasformato e restituito in forma conversazionale.

5.1 Architettura dell'ecosistema

Una volta individuati i requisiti e gli obiettivi che il sistema deve soddisfare, l'attenzione si è focalizzata sulla progettazione di un'architettura che integri un LLM con i web service offerti dal sistema ERP.

L'obiettivo principale è consentire agli utenti di interrogare il sistema ERP tramite linguaggio naturale, senza dover necessariamente conoscere la struttura tecnica delle API esposte dall'ERP e i relativi vincoli di utilizzo. Al fine di soddisfare tale esigenza, è stata progettata un'architettura organizzata in più livelli, ciascuno caratterizzato da una responsabilità ben definita.

In Figura 5.1 è riportato uno schema architetturale allo stato embrionale, all'interno del quale è possibile individuare quattro componenti principali:

- *Utente*: formula interrogazioni in linguaggio naturale al fine di ricevere informazioni provenienti dall'ERP.
- *LLM*: ha il compito di interpretare a livello semantico la richiesta dell'utente, identificandone l'intento e le informazioni rilevanti.

- *Server MCP*: costituisce il livello di orchestrazione del sistema, fungendo da livello intermedio tra il modello linguistico e i servizi applicativi.
- *Web Service REST*: rappresenta il livello applicativo, responsabile dell'accesso ai dati e dell'esecuzione delle API.

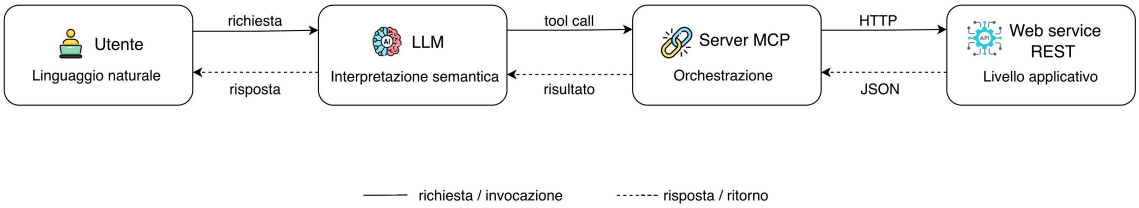


Figura 5.1: Architettura ad alto livello del sistema

Tale suddivisione architetturale consente di isolare la complessità tecnica intrinseca del sistema ERP e di evitare che i web service vengano esposti direttamente all'utente o all'LLM.

Un elemento chiave dell'architettura è rappresentato dal server MCP che svolge il ruolo di orchestrazione del sistema. Esso, infatti, non si limita a essere un semplice canale di comunicazione, ma introduce un vero e proprio livello di astrazione tra il dominio linguistico e quello tecnico e applicativo. In particolare, il server MCP ha il compito di gestire l'interazione con le API e di garantire che le richieste generate dall'LLM in seguito all'interpretazione semantica delle interrogazioni siano conformi con le specifiche tecniche imposte dalle stesse API. Tale aspetto risulta di fondamentale importanza, in quanto consente di vincolare l'operatività dell'LLM a un insieme controllato di funzionalità, riducendo il rischio di generare chiamate non valide o semanticamente scorrette.

Dal punto di vista architetturale, l'introduzione del server MCP permette di separare nettamente i compiti degli attori che compongono il sistema. In effetti, l'LLM si occupa esclusivamente della comprensione e dell'interpretazione della richiesta dell'utente, mentre il server MCP gestisce la traduzione operativa in chiamate API e l'interazione con il sistema ERP.

Un ulteriore aspetto rilevante riguarda il controllo sul comportamento del modello. L'LLM, infatti, non ha un accesso diretto alle API né alla loro struttura interna, ma interagisce con esse esclusivamente attraverso gli strumenti messi a disposizione dal server MCP. In questo modo, è possibile guidare il processo decisionale del modello e limitare il suo spazio di azione a operazioni ben definite e coerenti con il dominio applicativo.

Inoltre, la presenza di un livello intermedio rappresenta un punto strategico per il progresso del sistema. Grazie alla creazione di un server MCP, infatti, è possibile integrare facilmente nuovi servizi o apportare modifiche alla struttura delle API disponibili, senza richiedere interventi diretti sull'LLM o sull'interfaccia utente. Allo stesso modo, il sistema può essere esteso ad altri contesti applicativi, mantenendo, di fatto, invariata la sua struttura generale.

L'architettura proposta, quindi, consente un'interazione naturale e intuitiva da parte dell'utente, garantendo, al contempo, il rispetto dei vincoli tecnici imposti dal sistema ERP.

5.1.1 Integrazione con i web service a supporto dell'ERP

Una volta definita la struttura architetturale ad alto livello, è possibile analizzare più nel dettaglio i componenti che costituiscono l'ecosistema.

In particolare, l'integrazione con i web service rappresenta il punto di collegamento tra il livello di orchestrazione e il sistema ERP. Tuttavia, come già anticipato nella sezione precedente, tali servizi non vengono utilizzati direttamente dall'utente o dall'LLM. Al contrario, essi

sono incapsulati all'interno del server MCP che, in qualche modo, ne gestisce l'accessibilità e ne controlla l'utilizzo.

Avendo a disposizione i web service raccolti all'interno del server MCP, è possibile astrarre la complessità tecnica associata alle API REST. Pertanto, l'LLM non deve gestire gli aspetti quali la definizione degli endpoint, la formattazione dei parametri o il rispetto dei vincoli imposti dai tracciati JSON. Tali responsabilità, invece, sono delegate al server MCP, che funge da orchestratore capace di garantire coerenza nell'esecuzione delle operazioni. Tra l'altro, ciò semplifica ulteriormente il ruolo dell'LLM, che può concentrarsi esclusivamente sull'interpretazione semantica della richiesta dell'utente, senza doversi occupare della complessità tecnica sottostante.

Inoltre, tale approccio consente di mantenere un corretto allineamento tra le specifiche tecniche delle API e il loro effettivo utilizzo all'interno del sistema. Poiché le operazioni disponibili derivano direttamente dalla documentazione formale dei web service, ogni chiamata risulta coerente con le logiche applicative e con le regole di business definite nell'ERP. In tal senso, l'integrazione con i web service dell'ERP non introduce ambiguità, piuttosto rafforza la consistenza del sistema.

Dunque, sebbene i web service mantengano il loro ruolo centrale nell'accesso ai dati e alle funzionalità operative, essi risultano completamente trasparenti all'utente finale, rendendo l'interazione con il sistema più naturale ed intuitiva.

5.1.2 Generazione dinamica delle API

Durante la fase di progettazione, particolare attenzione è stata dedicata alle modalità di integrazione dei web service a supporto dell'ERP all'interno del sistema. In particolare, si è reso necessario comprendere come il server MCP potesse gestire e rendere effettivamente utilizzabile un insieme molto ampio di API, garantendo, al tempo stesso, coerenza con i requisiti individuati e una buona capacità di adattamento all'evoluzione del contesto applicativo.

In quest'ottica, anziché adottare un approccio statico basato sulla definizione manuale dei singoli endpoint, si è scelto di orientarsi verso una generazione dinamica delle API, in grado di adattarsi automaticamente alla struttura dei servizi disponibili. Nello specifico, sfruttando la documentazione formale dei web service in standard OpenAPI, è stato possibile generare automaticamente gli strumenti disponibili. In altre parole, ciascun endpoint descritto nella specifica viene interpretato e trasformato in un tool invocabile.

In Figura 5.2 è riportato uno schema che illustra come i diversi endpoint vengano trasformati automaticamente in tool MCP invocabili, evitando, così, una gestione manuale delle singole API.

Tale scelta è motivata, in primo luogo, dall'elevato numero di endpoint esposti dal sistema ERP e, soprattutto, dalla loro natura dinamica, soggetta a frequenti variazioni nel tempo. Di conseguenza, un'approccio statico comporterebbe continui interventi di manutenzione, risultando non solo oneroso, ma anche potenzialmente soggetto a errori. Al contrario, la generazione dinamica consente di superare tali limitazioni, garantendo al sistema di allinearsi automaticamente alle modifiche introdotte nella documentazione delle API.

Inoltre, tale soluzione si dimostra particolarmente vantaggiosa anche in termini di portabilità. Qualora, infatti, si rendesse necessario integrare un diverso ERP, caratterizzato da una differente struttura delle API, sarebbe sufficiente aggiornare la specifica OpenAPI di riferimento per ottenere, in modo automatico, un nuovo insieme di strumenti coerenti con il contesto operativo.

Allo stesso tempo, la generazione automatica dei tool consente all'LLM di operare su un insieme di tool sempre aggiornato allo stato reale dell'ERP. In particolare, il modello

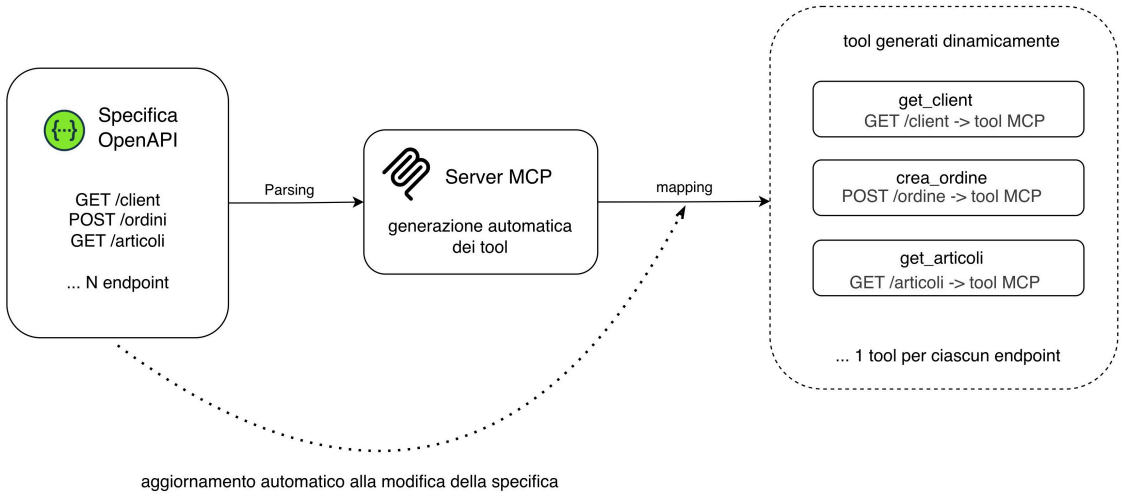


Figura 5.2: Generazione dinamica delle API

linguistico può fare riferimento alla descrizione strutturata delle API per individuare, di volta in volta, l'operazione più pertinente rispetto alla richiesta dell'utente.

L'adozione di un meccanismo di generazione automatica delle API, quindi, è particolarmente efficace in scenari complessi aziendali, nei quali l'elevato numero di servizi disponibili renderebbe impraticabile una gestione manuale.

5.1.3 Definizione dei tool MCP

Una volta chiarite le basi della progettazione, l'attenzione si è focalizzata sulla definizione dei tool, elementi centrali dell'architettura.

In tale contesto, il protocollo MCP svolge un ruolo fondamentale, poiché agisce come intermediario tra il modello linguistico e le funzionalità offerte dai sistemi esterni. In particolare, il compito dell'MCP consiste nel rendere comprensibili e utilizzabili dagli LLM le operazioni che, nella loro forma originaria, sarebbero accessibili soltanto attraverso interfacce tecniche.

Per raggiungere tale obiettivo, si è reso necessario un meccanismo di incapsulamento all'interno di un server MCP. In pratica, al fine di trasformare le API in tool utilizzabili dall'LLM, viene predisposto un server dedicato, ovvero un programma che si occupa di interagire con le API stesse e di esporne le funzionalità secondo il linguaggio e le regole previste dal protocollo MCP.

Il cuore di tale trasformazione risiede nella definizione dei tool. All'interno del server MCP, infatti, ogni endpoint viene mappato in un tool formalizzato secondo uno schema standard. Ciascun tool, infatti, è caratterizzato da tre elementi fondamentali:

- *nome*, ovvero un identificativo univoco che consente al modello linguistico di individuarlo correttamente;
- *descrizione* in linguaggio naturale, utile a spiegare all'LLM lo scopo e il contesto in cui deve essere utilizzato;
- *schema dei parametri*, generalmente espresso tramite strutture compatibili con JSON Schema. Tale schema definisce con precisione i dati di input che il modello deve fornire, il loro formato e le eventuali regole di validazione.

In tale scenario, il server MCP non si limita soltanto a inoltrare le richieste, ma svolge una vera e propria funzione di mediazione semantica. Da un lato, assume il ruolo di registro, organizzando i tool derivanti dalle API e comunicando all'LLM quali strumenti siano disponibili; dall'altro ne descrive le modalità di utilizzo, specificando i parametri richiesti, il tipo di dati attesi e la struttura delle risposta.

Ne deriva che la complessità tecnica delle API viene astratta e ricondotta a un'interfaccia più strutturata e interpretabile; ciò permette all'LLM di orientarsi tra le diverse operazioni disponibili.

Da un punto di vista formale, l'accesso a tali strumenti avviene secondo un'architettura di tipo client-server, nella quale l'LLM si comporta come un client MCP, mentre il server MCP rappresenta il punto di accesso alle risorse esterne.

5.1.4 Progettazione dei tool MCP

La possibilità di generare automaticamente e rendere disponibile un tool distinto per ciascun endpoint dell'ERP rappresenta, senza dubbio, un grande vantaggio. Tuttavia, secondo un'approccio iniziale più naive, l'esposizione diretta di tutti gli endpoint al modello linguistico non costituisce una scelta efficace. Sebbene il server MCP possa contenere un numero elevato di tool, non è opportuno renderli tutti disponibili contemporaneamente all'LLM.

In effetti, un LLM non ragiona bene quando viene posto di fronte a un numero eccessivo di azioni possibili, con il rischio, tra l'altro, di ridurre la capacità di individuare l'operazione più pertinente rispetto alla richiesta ricevuta.

Per risolvere il problema, è stata adottata un'architettura basata su macro-tool, nella quale il numero di tool esposti direttamente all'LLM viene drasticamente ridotto, mantenendo, al contempo, la possibilità di accedere all'intero insieme delle funzionalità disponibili.

L'obiettivo, infatti, non è quello di far conoscere all'LLM l'intero catalogo delle API e le relative strutture, bensì permettere ad esso di scoprire dinamicamente le funzionalità disponibili e di richiamare solo quelle necessarie nel momento opportuno.

Su tale base è stata definita una progettazione orientata a due tool principali, concepiti come punti di accesso ad alto livello verso l'intero ecosistema di servizi. Essi sono:

- *tool di discovery*, incaricato di fornire al modello una rappresentazione sintetica delle operazioni disponibili. In questo modo, l'LLM può esplorare il dominio applicativo senza essere esposto all'intera complessità sottostante e comprendere quale operazione risulti semanticamente più coerente con la richiesta dell'utente.
- *Tool di invocazione*, utilizzato per richiedere al server MCP l'esecuzione dell'API selezionata, specificando gli eventuali parametri di input necessari.

La Figura 5.3 sintetizza visivamente il passaggio dall'approccio naive, caratterizzato dall'esposizione diretta di un elevato numero di endpoint al modello linguistico, alla soluzione basata su macro-tool adottata nel sistema progettato.

Tale approccio, dunque, consente di costruire un percorso decisionale più ordinato. A tal proposito, l'LLM esplora dapprima l'elenco delle funzionalità disponibili, individua l'azione più pertinente e, infine, procede con la richiesta di esecuzione vera e propria.

5.1.5 Definizione del client MCP e dell'LLM adottato

Una volta definita la struttura dei tool del server MCP, è stato necessario individuare gli strumenti incaricati di gestire l'interazione con il sistema e l'interpretazione delle richieste da parte dell'utente.

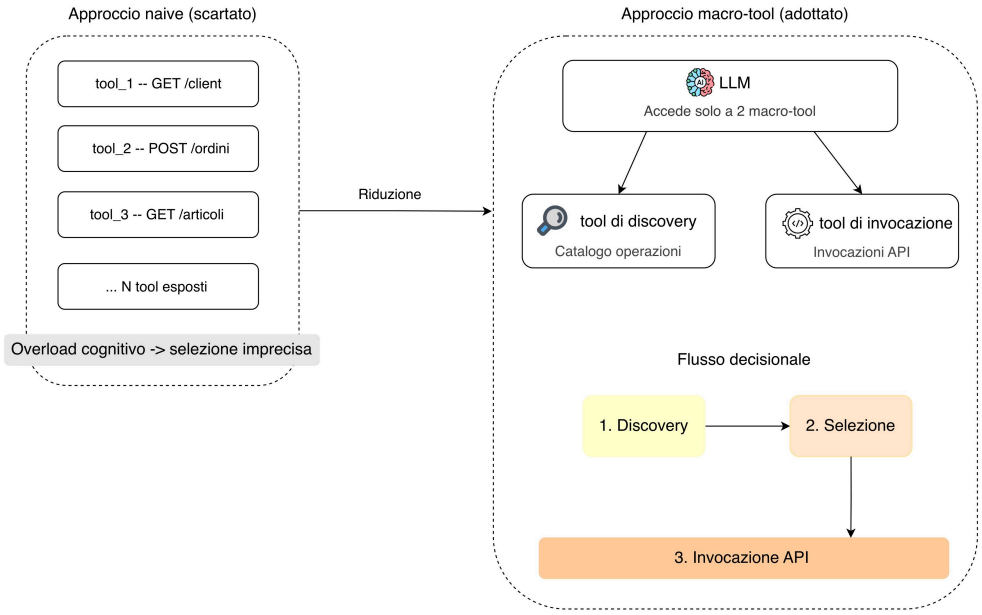


Figura 5.3: Passaggio dall'approccio naive alla soluzione basata su macro-tool

In tale contesto è stato scelto di utilizzare Gemini CLI come componente host e, allo stesso tempo, come client MCP, in linea con il modello architetturale dell'MCP descritto nei capitoli precedenti. Tale scelta si inserisce in modo naturale all'interno dell'architettura proposta, considerando che, come già evidenziato, host e client MCP risultano spesso strettamente integrati e, in alcuni casi, coincidono in un'unica entità applicativa.

Nello specifico, Gemini CLI è un'interfaccia a riga di comando che consente di interagire con i modelli della famiglia Gemini, offrendo, al tempo stesso, la possibilità di integrare gli strumenti esterni all'interno dello stesso ambiente operativo. Per tale ragione, esso si è rivelato una soluzione particolarmente efficace, in quanto consente di concentrare in un unico contesto sia la gestione del dialogo con l'utente sia la comunicazione strutturata con il server MCP.

In effetti, da una parte, Gemini CLI rappresenta il punto di accesso attraverso cui l'utente formula le proprie richieste e riceve le risposte; dall'altra, mette a disposizione i meccanismi necessari per stabilire la connessione con il server MCP, esplorare i tool disponibili e coordinarne l'utilizzo secondo le regole del protocollo. In questo modo, viene concretamente realizzata quella sovrapposizione tra host e client già introdotta a livello teorico.

5.1.6 Processo di autenticazione per l'accesso alle API REST

Per concludere la fase di progettazione del server MCP, è stato importante considerare il flusso di autorizzazione necessario per accedere ai servizi esposti dal sistema ERP.

In particolare, il processo di autorizzazione si articola in due fasi distinte ma strettamente collegate tra loro, il cui flusso operativo è rappresentato in Figura 5.4. Le principali fasi previste sono:

- *Basic Authentication*: in una prima fase vengono richiesti alcuni parametri applicativi per ottenere l'autorizzazione ad effettuare la richiesta POST all'endpoint per generare il JSON Web Token (JWT). Tale token è un codice univoco e valido per un periodo limitato, che consente di interagire con le API operative.

- *JWT Bearer*: una volta acquisito il token, esso deve essere incluso nell'header di ciascuna chiamata API al fine di garantire l'accesso alle risorse protette. Nel caso in cui il token risulti scaduto o non valido, le API restituiscono un errore di autorizzazione negata.

Inoltre, poiché il JWT ha una durata limitata, è stato necessario prevedere una logica di rinnovo automatico, al fine di garantire la continuità operativa senza interrompere il flusso di comunicazione.

Tale processo di refresh, quindi, consente al server MCP di avere un token valido e di evitare errori dovuti a scadenze inattese.

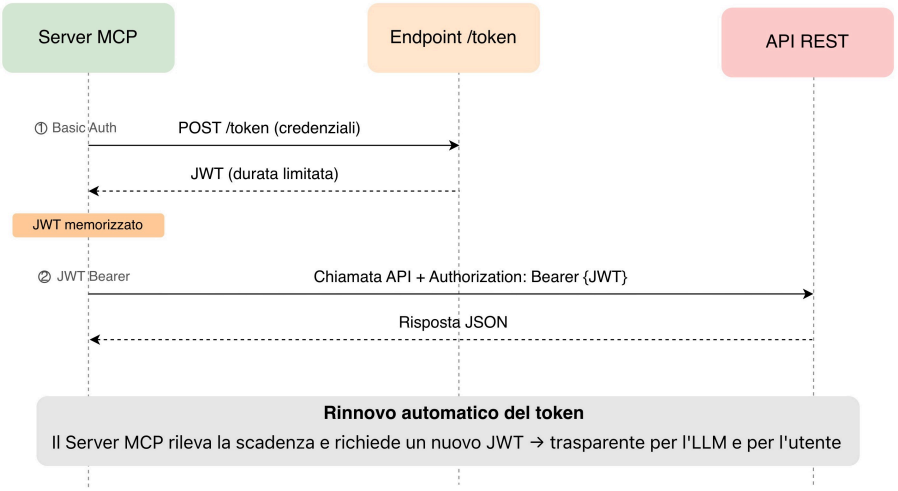


Figura 5.4: Flusso di autenticazione e di autorizzazione per l'accesso all'ERP

Inoltre, il processo di autenticazione viene gestito direttamente dal server MCP e risulta completamente trasparente sia per l'utente finale che per l'LLM. Essi, quindi, non devono conoscere i dettagli dell'accesso alle API. La gestione del token, invece, viene delegata al livello di orchestrazione, che esegue automaticamente i passaggi necessari e restituisce all'LLM soltanto il risultato dell'operazione richiesta.

5.2 Validazione manuale delle API e supporto al testing tramite Postman

Nel processo di integrazione dei web service all'interno dell'architettura progettata è emersa, fin da subito, la necessità di uno strumento in grado di analizzare il comportamento delle API e verificarne il funzionamento prima di esporle tramite il server MCP. In tale contesto, un ruolo centrale è stato svolto da Postman, una piattaforma molto utilizzata per testare le API grazie alla sua interfaccia grafica intuitiva che consente di inviare richieste HTTP e di ispezionare le risposte restituite dalle API stesse.

Al tal fine, sono state importate in Postman le collection aziendali, ovvero insiemi organizzati di richieste API precompilate raccolte secondo una logica funzionale che permettono di facilitare sia l'esecuzione dei test che la navigazione tra i diversi endpoint disponibili. In Figura 5.5 è mostrata la schermata iniziale di Postman.

L'utilizzo di Postman ha consentito di verificare il corretto funzionamento dei web service e, allo stesso tempo, di analizzare i parametri richiesti, la struttura delle risposte e le eventuali condizioni di errore, il tutto in un ambiente visivo e facilmente interpretabile. Ciò ha favorito

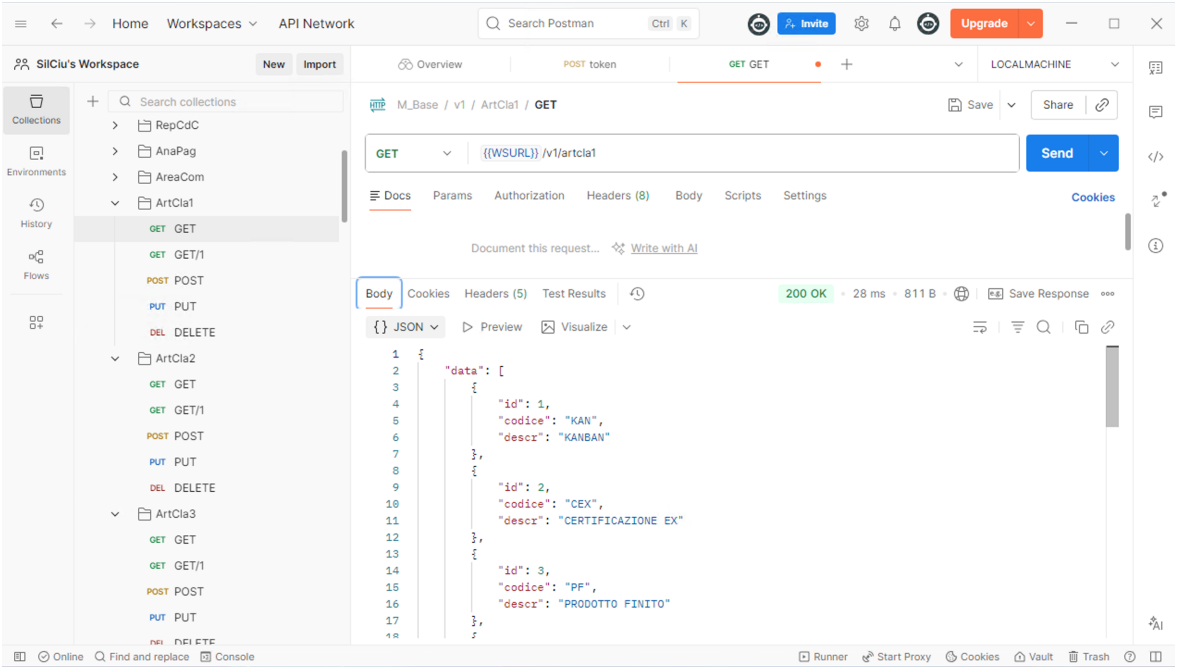


Figura 5.5: Schermata iniziale di Postman

una comprensione concreta dei servizi, un aspetto essenziale per la successiva modellazione dei tool all'interno dell'ecosistema MCP.

Un aspetto interessante è che Postman recentemente ha introdotto un supporto diretto ai server MCP esterni, rendendo possibile il testing e il debug direttamente dall'interfaccia della piattaforma. Una volta connesso al server MCP esterno, infatti, esso consente di visualizzare e utilizzare tutti i tool esposti, simulandone il comportamento come farebbe un client MCP.

Di conseguenza, Postman non è stato solo uno strumento usato per la validazione preliminare delle API, ma anche un supporto concreto per l'integrazione con il server MCP. In particolare, ha permesso di confrontare le risposte ottenute tramite chiamate dirette ai web service con quelle restituite attraverso il livello di orchestrazione MCP e, successivamente, con quelle generate nell'interazione mediata da un LLM.

5.3 Progettazione del flusso di interazione

Una volta completata la progettazione dei singoli componenti del sistema, il passo successivo è stato definire come questi interagiscono tra loro, in modo da descrivere chiaramente il funzionamento dell'intero ecosistema.

A tale scopo, è stata realizzata una pipeline end-to-end che accompagna l'intero processo, dalla richiesta iniziale dell'utente in linguaggio naturale fino alla sua traduzione in un'azione concreta sui servizi aziendali, con la successiva restituzione del risultato all'utente finale (Figura 5.6).

Nel dettaglio, l'interazione ha origine dall'utente, che formula una richiesta in linguaggio naturale attraverso l'interfaccia conversazionale. Essa viene interpretata dall'LLM, il quale ne analizza il contenuto semantico e valuta se sia necessario ricorrere a strumenti esterni per soddisfarla. Qualora sia richiesto richieda l'accesso alle informazioni dell'ERP, l'LLM individua i tool più adatti tra quelli messi a disposizione dal server MCP, sulla base delle descrizioni e dei parametri associati.

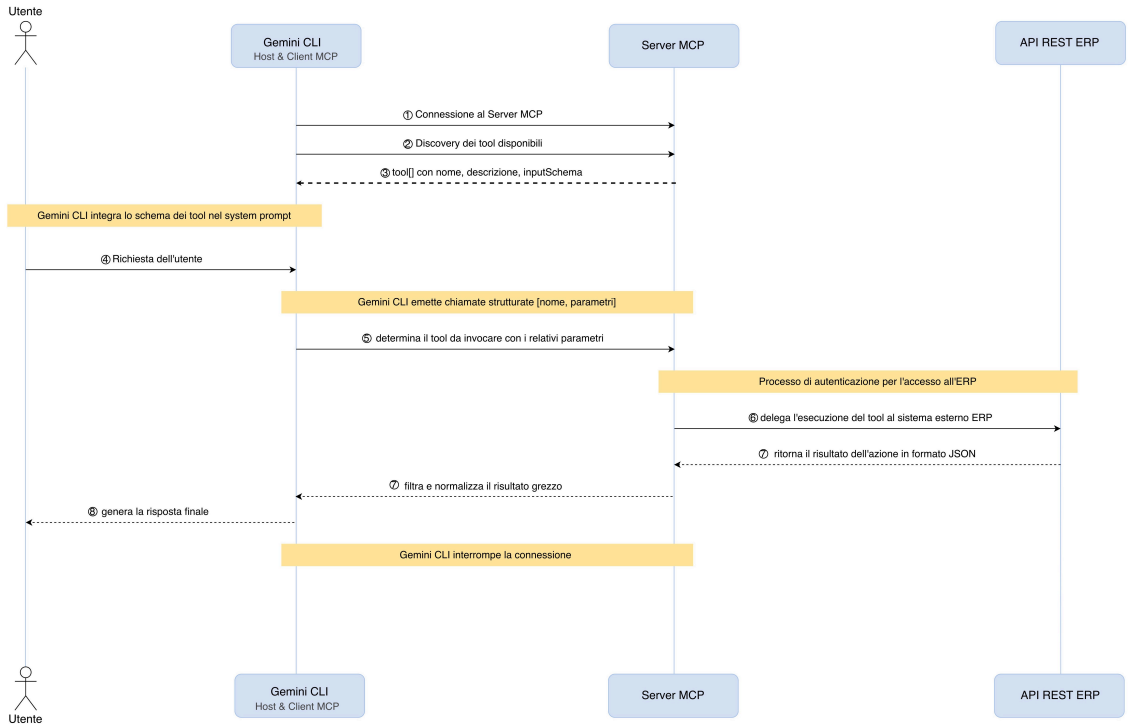


Figura 5.6: Flusso operativo end-to-end

A questo punto, l'LLM invia una richiesta strutturata al server MCP, specificando l'operazione da eseguire e gli eventuali parametri necessari. Il server MCP, quindi, entra in gioco come livello intermedio; esso prende in carico la richiesta, la traduce in una chiamata concreta verso le API del sistema ERP e ne gestisce l'esecuzione.

Una volta elaborata l'operazione, l'ERP restituisce il risultato al server MCP, che si occupa di normalizzarlo e renderlo nuovamente comprensibile dal modello linguistico. A partire da tali informazioni, L'LLM completa il processo e genera una risposta chiara e comprensibile che viene, infine, restituita all'utente finale.

Ciò che emerge da tale flusso di lavoro è, sicuramente, una netta distribuzione dei ruoli: l'utente esprime un'esigenza, l'LLM la interpreta e decide come agire, il server MCP gestisce l'interazione tecnica con i servizi esterni, mentre l'ERP esegue concretamente l'azione desiderata. Una struttura di questo tipo permette di mantenere il sistema modulare e scalabile, favorendo, al tempo stesso, una maggiore robustezza dell'intero sistema.

5.3.1 Ruolo del prompt nella guida dell'LLM

Un aspetto importante del flusso operativo è il ruolo del prompt, ovvero l'insieme di istruzioni che vengono fornite all'LLM per orientarne il comportamento. In effetti, non basta mettere a disposizione dell'LLM i tool esposti dal server MCP, ma è necessario anche guidarlo nell'utilizzo corretto di tali strumenti.

In quest'ottica, il prompt assume una funzione di guida, poiché consente di fornire al modello indicazioni su come interpretare le richieste dell'utente e su come tradurre queste ultime in operazioni coerenti con le funzionalità offerte dal sistema. In particolare, attraverso il prompt, è possibile orientare il modello nella selezione dei tool, nella gestione dei parametri di input e nell'utilizzo dei dati restituiti dalle API.

In questo modo, il modello non agisce in modo completamente libero, ma segue un percorso più strutturato. Esso, infatti, in primo luogo, analizza la richiesta, successivamente individua lo strumento più pertinente e, infine, utilizza i dati ottenuti per costruire la risposta.

Il prompt diventa, quindi, un elemento fondamentale per rendere il comportamento del modello più controllabile e coerente con gli obiettivi del sistema. In questo modo, il modello può fornire risposte più precise, grazie alle istruzioni e alle indicazioni fornire dal prompt.

5.3.2 Trasformazione e presentazione dei dati restituiti dalle API

Per concludere la progettazione del flusso operativo del sistema, è stato preso in considerazione anche il modo in cui le informazioni vengono presentate all'utente finale. Le risposte dai web service dell'ERP, infatti, sono generalmente strutturate in formato JSON e, oltre ai dati utili, possono includere anche informazioni tecniche poco significative in un contesto conversazionale.

Per tale ragione, è stato previsto un meccanismo di trasformazione delle risposte, con l'obiettivo di rendere i dati più chiari e semplici da interpretare. In particolare, gli elementi non significativi vengono filtrati, mentre i contenuti utili vengono riorganizzati in una forma più leggibile e coerente con la richiesta formulata dall'utente.

Di conseguenza, il sistema non si limita a mostrare i risultati grezzi delle API, bensì elabora le informazioni ottenute al fine di presentarle in maniera più comprensibile e naturale all'interno della conversazione.

CAPITOLO 6

Implementazione dell'ecosistema

Nel presente capitolo verranno messi in luce gli aspetti salienti, a livello implementativo, dell'ecosistema basato su Generative AI per l'accesso in linguaggio naturale ai web service di un ERP.

In primo luogo, verrà analizzata la struttura del progetto, mostrando l'organizzazione dei file e delle cartelle.

Successivamente, sarà illustrata la preparazione della specifica OpenAPI, necessaria per rendere le API esposte dall'ERP compatibili con il framework FastMCP.

In seguito, sarà approfondita l'implementazione del server MCP, con particolare attenzione alla definizione dei tool e alla gestione dell'autenticazione tramite token JWT.

Il capitolo affronterà, inoltre, le attività di adattamento della specifica OpenAPI per garantire la compatibilità con il sistema di validazione di FastMCP e la trasformazione delle risposte JSON in una forma più leggibile per il modello linguistico.

Infine, verrà descritta l'integrazione del server MCP con Gemini CLI e la definizione delle istruzioni fornite al modello Gemini per orientarne il comportamento nell'utilizzo dei tool disponibili.

6.1 Struttura del progetto

Conclusa la fase di progettazione, si è passati all'implementazione dell'ecosistema. In particolare, si è scelto di adottare un'organizzazione modulare del codice, con l'obiettivo di separare le diverse responsabilità applicative e di facilitare lo sviluppo delle funzionalità descritte in fase progettuale. A tal proposito, il codice sorgente è stato organizzato in moduli Python distinti, ciascuno dedicato a una specifica funzionalità del sistema.

La struttura principale del progetto comprende i seguenti file e le seguenti directory:

- `mcp/`: directory che aggrega i principali componenti legati al protocollo MCP. Al suo interno, il file `server.py` si occupa della configurazione e dell'inizializzazione del server, mentre il file `tools.py` definisce i macro tool esposti all'LLM e la logica di invocazione dinamica delle API. A questi, si affianca il prompt che contiene le istruzioni fornite all'LLM per guidarlo nell'utilizzo dei tool disponibili.
- `auth.py`: dedicato alla gestione dell'autenticazione, contenente la logica necessaria per la generazione e il rinnovo del token JWT utilizzato per accedere ai servizi ERP.
- `openapi/`: directory che raccoglie le funzionalità necessarie per adattare la specifica OpenAPI al funzionamento del framework FastMCP.
- `formatter.py`: si occupa di riorganizzare le risposte JSON restituite dalle API in una forma più leggibile e coerente con il contesto conversazionale.

- `settings.py`: file che centralizza il caricamento delle variabili applicative necessarie per il funzionamento del server MCP.
- `.env`: file che raccoglie le variabili d'ambiente per la configurazione del sistema, come le credenziali di autenticazione, la chiave API per l'interfacciamento con l'LLM e l'URL dei web service con i relativi parametri di accesso.
- `requirements.txt`: file che elenca tutte le dipendenze Python necessarie per l'esecuzione del progetto.

Tale organizzazione consente di mantenere il codice pulito e manutenibile e, inoltre, facilita eventuali sviluppi futuri del sistema, come l'aggiunta di nuovi tool, l'integrazione di ulteriori servizi ERP o l'estensione delle funzionalità del server MCP.

6.2 Preparazione della specifica OpenAPI

La prima fase dello sviluppo software si è concentrata sulla preparazione della specifica OpenAPI utilizzata dal protocollo MCP per la generazione dinamica dei tool. Essa si è rivelata un elemento centrale, poiché il sistema sviluppato basa la propria logica operativa sulla descrizione formale dei web service a supporto del sistema ERP.

Come già discusso in fase progettuale, il protocollo MCP non interagisce direttamente con gli endpoint definiti manualmente, ma utilizza la rappresentazione strutturata delle API per individuare le operazioni disponibili, i parametri richiesti e le corrette modalità di invocazione. Per tale ragione, è stato necessario predisporre una specifica aggiornata e coerente con gli standard OpenAPI, al fine di renderla interpretabile dai componenti responsabili della generazione dei tool.

Originariamente la documentazione dei web service risultava disponibile in formato Swagger 2.0. Sebbene tale standard rappresenti ancora una soluzione ampiamente diffusa, esso presenta alcune limitazioni in termini di compatibilità con gli strumenti più recenti e con le librerie orientate all'ecosistema OpenAPI 3.x. Di conseguenza, prima di procedere con il caricamento della documentazione, si è reso necessario introdurre un processo di conversione, così da renderla integrabile con gli strumenti adottati nel progetto.

6.2.1 Conversione da Swagger 2.0 a OpenAPI 3.x

Il processo di conversione della specifica inizia con il download automatico della documentazione Swagger, effettuato mediante una richiesta HTTP autenticata tramite *Basic Authentication* con le credenziali applicative configurate per l'accesso ai servizi aziendali.

Una volta ottenuto il documento JSON originale, è stato necessario eseguire una fase preliminare di pulizia dei dati, poiché la specifica Swagger conteneva caratteri di controllo non validi e caratteri ASCII non stampabili che causavano errori durante il processo di conversione.

Terminata la fase di pulizia, la documentazione viene convertita automaticamente dalla versione Swagger 2.0 allo standard OpenAPI 3.x tramite Swagger Editor, uno strumento che consente di aggiornare, in modo guidato, le specifiche API da versioni precedenti a standard più moderni.

Ad ogni modo, i dettagli implementativi sono riportati nel Listato 6.1.

Al termine della conversione, la specifica OpenAPI viene salvata localmente all'interno della directory `openapi`, così da poter analizzare più facilmente eventuali problemi legati alla struttura delle API e alla generazione dei tool MCP durante le attività di testing e debugging. Inoltre, il salvataggio locale riduce i tempi di inizializzazione del server, evitando

di dover scaricare e convertire nuovamente la specifica durante ciascuna esecuzione, almeno finché la documentazione dei web service non subisca modifiche.

```
# ======================================================
# Funzione per escapare caratteri di controllo dentro le stringhe JSON
# ======================================================
def escape_control_chars_in_json(raw: str) -> str:
   string_regex = re.compile(r'"(\\.|[^"\\])*"', re.DOTALL)

   def escape_match(match):
      s = match.group(0)
      s_escaped = s.replace("\n", "\\n").replace("\r", "\\r")
      s_escaped = re.sub(r'[\x00-\x08\x0B\x0C\x0E-\x1F]', '', s_escaped)
      return s_escaped

   return string_regex.sub(escape_match, raw)

# ======================================================
# Funzione per scaricare e convertire Swagger 2.0  OpenAPI 3.0
# ======================================================
def convert_swagger_online(swagger_url: str, output_file_path: str = None):
   logger.info(f"Scarico Swagger da: {swagger_url}")

   try:
      resp = requests.get(swagger_url, auth=(BASIC_AUTH_USERNAME, BASIC_AUTH_PASSWORD))
      resp.raise_for_status()
   except Exception as e:
      logger.error(f"Errore nel download dello Swagger: {e}")
      raise

   raw = resp.text
   raw_cleaned = escape_control_chars_in_json(raw)

   with open("swagger_cleaned_debug.json", "w", encoding="utf-8") as f:
      f.write(raw_cleaned)
   logger.info("Swagger pulito salvato in swagger_cleaned_debug.json")

   try:
      swagger_data = json.loads(raw_cleaned)
   except json.JSONDecodeError as e:
      logger.error(f"Errore parsing JSON dopo pulizia: {e}")
      raise

   logger.info("Converto Swagger  OpenAPI 3 tramite converter.swagger.io")
   converter_url = "https://converter.swagger.io/api/convert"

   try:
      response = requests.post(converter_url, json=swagger_data)
      response.raise_for_status()
      openapi_data = response.json()
   except Exception as e:
      logger.error(f"Errore durante la conversione online: {e}")
      raise

   if output_file_path:
      with open(output_file_path, "w", encoding="utf-8") as f:
         json.dump(openapi_data, f, indent=2)
      logger.info(f"Specifica OpenAPI salvata in: {output_file_path}")

   return openapi_data


   # 1) Scarica e converte Swagger
   openapi_spec = convert_swagger_online(SWAGGER_JSON, "./converted_openapi.json")
```

Listato 6.1: Conversione della specifica Swagger 2.0 in OpenAPI 3.x

6.3 Implementazione del server MCP

Una volta completata la preparazione della specifica OpenAPI, l'attenzione si è concentrata sull'implementazione del server MCP, componente responsabile della gestione delle chiamate ai web service a supporto del sistema ERP. All'interno dell'architettura progettata, infatti, il server svolge il ruolo di orchestratore capace di mettere in comunicazione l'LLM con i servizi esposti dall'ERP. In particolare, esso si occupa di astrarre la complessità tecnica delle API e di esporne le funzionalità tramite tool interpretabili dal modello linguistico.

Come già anticipato nei capitoli precedenti, per lo sviluppo del server MCP è stato utilizzato il framework FastMCP, una libreria Python progettata per facilitare la creazione di server conformi al protocollo MCP. Uno degli aspetti più interessanti di FastMCP è la possibilità di concentrarsi principalmente sulla logica applicativa e sulla definizione dei tool,

senza dover gestire direttamente i dettagli più complessi legati al protocollo di comunicazione. In questo modo, gran parte degli aspetti infrastrutturali vengono demandati al framework stesso, semplificando, dunque, lo sviluppo del sistema.

Inoltre, FastMCP consente sia la generazione automatica dei tool a partire da una specifica OpenAPI, sia la definizione manuale dei tool personalizzati in base alle esigenze del progetto.

A tal proposito, nel progetto si è scelto di non esporre direttamente l'insieme degli endpoint all'LLM. É stata, invece, adottata una soluzione più controllata basata su macro-tool, in linea con quanto discusso nella fase di progettazione, con l'obiettivo di ridurre il numero di tool disponibili e guidare, in modo più efficace, il comportamento del modello linguistico.

6.3.1 Inizializzazione del server

Fatte tali premesse, l'inizializzazione del server MCP è partita da una configurazione base proposta dal framework FastMCP.

Come mostrato nel Listato 6.2, la creazione del server prevede il caricamento della specifica OpenAPI, la definizione di un client HTTP asincrono per effettuare le chiamate API e la successiva inizializzazione del server tramite il metodo `FastMCP.from_openapi()`. Con queste poche righe di codice, dunque, gli endpoint definiti nella documentazione vengono automaticamente trasformati in tool MCP utilizzabili dall'LLM.

```
import httpx
from fastmcp import FastMCP

# Create an HTTP client for your API
client = httpx.AsyncClient(base_url="https://api.example.com")

# Load your OpenAPI spec
openapi_spec = httpx.get("https://api.example.com/openapi.json").json()

# Create the MCP server
mcp = FastMCP.from_openapi(
   openapi_spec=openapi_spec,
   client=client,
   name="My API Server"
)

if __name__ == "__main__":
   mcp.run()
```

Listato 6.2: Configurazione di base di un server MCP tramite FastMCP

Tuttavia, sebbene questa soluzione rappresenti un ottimo punto di partenza, durante lo sviluppo del progetto sono emerse alcune limitazioni legate soprattutto alla necessità di controllare, in maniera più precisa, il comportamento dell'LLM e la modalità di esposizione delle API.

Per tale ragione, l'implementazione del server si è progressivamente allontanata dalla configurazione standard proposta dal framework, introducendo una gestione personalizzata dei tool e una logica di orchestrazione più coerente con le esigenze dell'ecosistema progettato.

6.3.2 Definizione dei tool

Una volta inizializzato il server MCP, il passo successivo è stata la costruzione di una struttura di mapping tra le `operationId` definite nella specifica OpenAPI e i relativi endpoint ERP. In pratica, il sistema attraversa automaticamente la sezione `paths` della documentazione OpenAPI ed estrae, per ciascuna operazione, le informazioni quali il metodo HTTP, il path e la relativa specifica associata. Una volta ricavate, tali informazioni vengono memorizzate all'interno di una struttura dati centralizzata, utilizzata poi dal server per individuare l'endpoint corretto da invocare. I principali dettagli implementativi di tale meccanismo sono riportati nel Listato 6.3.

```
    operations = {}
    for path, methods in openapi_spec.get("paths", {}).items():
        for method, spec in methods.items():
            if method.startswith("x-"):
                continue

            op_id = spec.get("operationId")
            if not op_id:
                continue

            operations[op_id] = {
                "method": method.upper(),
                "path": path,
                "spec": spec
            }

    logger.info(f"Caricate {len(operations)} operationId")
```

Listato 6.3: Costruzione del mapping tra le `operationId` e gli endpoint dell'ERP

Come discusso nella fase di progettazione, l'obiettivo non era esporre direttamente all'LLM migliaia di endpoint distinti, ma introdurre un livello di astrazione capace di guidare il modello linguistico nell'utilizzo delle API disponibili.

Per tale ragione, sono stati implementati due macro-tool principali, definiti per mezzo del decoratore `@mcp.tool()`, che consente al framework FastMCP di registrare automaticamente le funzioni come tool invocabili dall'LLM.

Il primo macro-tool, `list_operations`, mostrato nel Listato 6.4, ha il compito di fornire al modello linguistico una panoramica sintetica delle operazioni disponibili. Nello specifico, esso restituisce l'elenco delle `operationId` estratte dalla specifica OpenAPI insieme alle relative informazioni descrittive, come il metodo HTTP, il path dell'endpoint e una breve descrizione dell'operazione. In questo modo, l'LLM può esplorare dinamicamente le funzionalità offerte dal sistema senza conoscere direttamente l'intera struttura tecnica delle API.

```
@mcp.tool()
async def list_operations():
    """
    Restituisce l'elenco completo delle operazioni API disponibili,
    ciascuna con il proprio operation_id, una descrizione sintetica,
    il metodo HTTP e il percorso dell'endpoint.
    Deve essere invocato SEMPRE come primo passo, prima di qualsiasi
    chiamata API, per identificare l'operationId semanticamente più
    coerente con la richiesta dell'utente.
    """
    return [
        {
            "operation_id": op_id,
            "summary": op["spec"].get("summary"),
            "path": op["path"],
            "method": op["method"]
        }
        for op_id, op in operations.items()
    ]
```

Listato 6.4: Implementazione del macro-tool `list_operations` per la consultazione dei tool disponibili

Il secondo macro-tool, `openapi_call`, riportato nel Listato 6.5, rappresenta, invece, il vero punto di accesso operativo verso i servizi ERP. Esso riceve in ingresso l'identificativo dell'operazione da eseguire con gli eventuali parametri necessari alla chiamata API, come i parametri del path, i parametri della query, l'header o il body della richiesta.

```
@mcp.tool()
async def openapi_call(
    operation_id: str,
    path_params: dict | None = None,
    query_params: dict | None = None,
    headers: dict | None = None,
    body: dict | list | str | None = None
):
    """
    Esegue una chiamata verso un endpoint API identificato dall'operation_id
```

```
    ottenuto tramite list_operations.
    Accetta parametri opzionali: path_params per i valori da sostituire
    nel percorso URL, query_params per i filtri nella query string,
    body per il contenuto della richiesta.
    Restituisce la risposta già trasformata in testo leggibile: il risultato
    deve essere utilizzato come base per formulare una risposta chiara e
    contestuale all'utente, senza mostrare mai i dati tecnici grezzi.
    """
    if operation_id not in operations:
       raise ValueError(f"operationId sconosciuto: {operation_id}")

    op = operations[operation_id]
    method = op["method"]
    path = op["path"]

    if path_params:
       for k, v in path_params.items():
          path = path.replace(f"{{{k}}}", str(v))

    response = await client.request(
       method=method,
       url=path,
       params=query_params,
       json=body,
       headers=headers
    )

    data = response.json() if response.content else {}
    return format_json_readable(data)
```

Listato 6.5: Implementazione del macro-tool `openapi_call` per l'invocazione dinamica delle API dell'ERP

Da evidenziare, infine, come il tool verifichi innanzitutto che l'operazione richiesta sia presente tra quelle caricate dalla specifica OpenAPI. Una volta individuato l'endpoint corretto, il sistema ricostruisce dinamicamente il metodo HTTP e il path della richiesta, sostituendo, eventualmente, i parametri presenti nell'URL. A questo punto, il server MCP può effettuare la chiamata verso il sistema ERP tramite il client HTTP autenticato.

6.4 Gestione dell'autenticazione

I servizi esposti dal sistema ERP non risultano direttamente accessibili, ma richiedono un processo di autorizzazione basato sull'utilizzo di un token JWT. Alla luce di ciò, una volta definito il meccanismo di invocazione delle API tramite i macro-tool MCP, è stata implementata la gestione dell'autenticazione all'interno del server.

In particolare, il token JWT viene generato mediante una chiamata verso l'endpoint di autenticazione messo a disposizione dall'ERP e, successivamente, utilizzato per autorizzare tutte le richieste API effettuate dal server MCP.

Nel dettaglio, la gestione dell'autenticazione si articola in tre aspetti principali:

- la generazione del token JWT;
- l'utilizzo del token nelle chiamate API;
- il rinnovo automatico del token in caso di scadenza o di errore di autorizzazione.

Ciascuno di questi aspetti verrà descritto, dal punto di vista implementativo, nelle prossime sezioni.

6.4.1 Generazione del token JWT

La richiesta del token JWT viene gestita tramite la funzione `get-jwt-token`, i cui principali dettagli implementativi sono riportati nel Listato 6.6.

```
async def get_jwt_token():
   async with httpx.AsyncClient(
```

```
        base_url=SWAGGER_URL,
        auth=(BASIC_AUTH_USERNAME, BASIC_AUTH_PASSWORD)
    ) as client:

        data = {
            "jwtusername": jwtusername,
            "jwtpassword": jwtpassword,
            "companyid": idcompany
        }

        logger.info("Richiesta token JWT...")

        resp = await client.post("/token", data=data)
        resp.raise_for_status()

        token = resp.json().get("token")

        logger.info("Token JWT ottenuto correttamente")

        return token
```

Listato 6.6: Funzione asincrona per la richiesta del token JWT

La funzione crea un client HTTP dedicato alla fase iniziale di autenticazione tramite *Basic Authentication* e lo utilizza per inviare una richiesta `POST` verso l'endpoint `/token`, responsabile della generazione del token JWT. In particolare l'endpoint `/token` riceve in input i parametri applicativi necessari all'autenticazione, ovvero username, password e identificativo aziendale, trasmessi nel corpo come `form data`, secondo il formato atteso dall'ERP.

Successivamente, la funzione verifica l'esito della richiesta ed, eventualmente, interrompe il flusso in presenza di errori HTTP. In caso di autenticazione completata correttamente, il token JWT viene estratto dalla risposta JSON, così da poter essere utilizzato per l'autorizzazione delle successive chiamate API.

Da evidenziare, infine, che il client utilizzato in questa fase è temporaneo, nel senso che esso viene creato esclusivamente per la richiesta del token e chiuso al termine dell'operazione. In tal caso, quindi, l'utilizzo è limitato a una singola operazione e non richiede di mantenere la connessione nel tempo.

6.4.2 Utilizzo del token nelle chiamate API

Una volta ottenuto il token JWT, esso viene inserito nell'header HTTP `Authorization` in modo tale da autenticare tutte le richieste successive verso i servizi ERP.

A tale scopo, è stato definito un client HTTP "persistente" condiviso tra i tool MCP, la cui implementazione è riportata nel Listato 6.7.

```
token = await get_jwt_token()

client = httpx.AsyncClient(
    base_url=SWAGGER_URL,
    headers={"Authorization": f"Bearer {token}"}
)
```

Listato 6.7: Creazione del client HTTP autenticato per le chiamate alle API REST

A differenza del client temporaneo utilizzato per la generazione del token JWT, il client "persistente" è stato introdotto per gestire tutte le comunicazioni continue con i web service esposti dall'ERP. A tal proposito, mantenere attiva la connessione evita la creazione di un nuovo client per ciascuna richiesta e, inoltre, riduce il sovraccarico legato all'apertura e alla chiusura ripetuta delle connessioni HTTP.

Poiché il token JWT viene configurato direttamente negli header del client "persistente", tutte le chiamate verso i servizi dell'ERP risultano automaticamente autenticate. In questo modo, quindi, l'LLM non deve occuparsi esplicitamente dei meccanismi di autenticazione, poiché questi rimangono incapsulati all'interno del server MCP.

6.4.3 Rinnovo automatico del token JWT

Dal momento che il token JWT ha una validità temporale limitata, si è reso necessario introdurre un meccanismo di rinnovo automatico per esso, al fine di garantire la continuità operativa del sistema anche durante sessioni prolungate di utilizzo.

In assenza di un meccanismo di refresh automatico, la scadenza del token causerebbe errori durante l'invocazione delle API REST. Tali errori verrebbero propagati fino all'LLM, che potrebbe interpretarli come problemi di autenticazione arrivando, potenzialmente, a richiedere nuovi dati all'utente. Per evitare tale comportamento, la gestione del token è stata resa completamente trasparente sia per l'LLM che per l'utente finale.

Chiarita la necessità di introdurre una logica di rinnovo del token, non rimane che descrivere come essa sia stata implementata. I principali dettagli implementativi della funzione adibita a ciò sono riportati nel Listato 6.8.

```
async def authenticated_request(method, url, **kwargs):

    global client

    response = await client.request(method, url, **kwargs)

    # Token scaduto o non valido
    if response.status_code == 401:

        logger.warning("Token JWT scaduto. Avvio refresh automatico...")

        # Nuova autenticazione
        new_token = await get_jwt_token()

        # Aggiornamento header Authorization
        client.headers["Authorization"] = f"Bearer {new_token}"

        # Retry automatico della richiesta originaria
        response = await client.request(method, url, **kwargs)

    return response
```

Listato 6.8: Rinnovo automatico del token JWT in caso di errore di autorizzazione

Il token ottenuto durante la fase iniziale di autenticazione viene mantenuto in memoria dal server MCP tramite il client HTTP condiviso tra i tool. Poiché il sistema ERP non fornisce informazioni esplicite sulla durata del token JWT, il server non è in grado di conoscere preventivamente il momento esatto della sua scadenza. Di conseguenza, la validità del token JWT viene verificata indirettamente durante l'esecuzione delle chiamate API.

Nello specifico, il server rileva la scadenza del token quando una chiamata ai servizi ERP restituisce un errore `401 Unauthorized`. Quando ciò accade, il server MCP avvia una nuova procedura di autenticazione per generare un token JWT valido; il token viene aggiornato nell'header `Authorization` e la richiesta API che aveva generato l'errore viene ripetuta.

Il rinnovo del token, dunque, avviene in modo trasparente, senza interrompere il flusso operativo del sistema. Di fatto, la gestione dell'autenticazione non ricade né sull'utente né sull'LLM, ma rimane completamente incapsulata nel server MCP.

6.5 Compatibilità tra la specifica OpenAPI e il framework FastMCP

Una volta completata l'implementazione del server MCP e definiti i relativi tool, è emersa la necessità di comprendere più a fondo il modo in cui FastMCP interpreta e utilizza internamente la specifica OpenAPI.

Il framework, infatti, non si limita a generare dinamicamente i tool a partire dalla documentazione, ma sfrutta anche gli schemi JSON definiti nella specifica durante l'esecuzione delle chiamate API, sia per costruire correttamente le richieste HTTP, sia per verificare la coerenza delle risposte restituite dai servizi ERP rispetto a quanto dichiarato.

Tale comportamento ha sicuramente un vantaggio in termini di affidabilità, poiché riduce il rischio di incongruenze tra i dati attesi e quelli realmente ricevuti. Tuttavia, nel corso dello sviluppo, sono emerse alcune limitazioni legate alla rigidità del sistema di validazione adottato da FastMCP. In diversi casi, di fatto, il framework applicava controlli più stringenti rispetto a quelli previsti dallo standard OpenAPI, generando errori anche in presenza di specifiche formalmente corrette.

Per questo motivo, si è resa necessaria una fase di adattamento della documentazione OpenAPI, con l'obiettivo di colmare tali incongruenze e garantire una maggiore compatibilità con il framework.

6.5.1 Controllo di conformità delle risposte ricevute dalle API in FastMCP

Una volta approfondito il ruolo della specifica OpenAPI all'interno di FastMCP, l'analisi si è concentrata sul modo in cui il framework verifica la conformità delle risposte API rispetto agli schemi JSON definiti nella documentazione.

Quando un tool invoca un endpoint, FastMCP riceve la risposta HTTP dell'ERP e la confronta con lo schema associato all'endpoint nella specifica OpenAPI. Durante questa fase, il framework verifica che i tipi dei campi siano coerenti con quanto dichiarato, che le proprietà obbligatorie siano presenti e che la struttura complessiva del file JSON rispetti lo schema previsto.

Se la risposta risulta conforme allo schema, il risultato viene restituito normalmente al tool MCP. In caso contrario, FastMCP genera un errore di validazione all'interno del quale indica il campo non conforme, il tipo atteso e il valore effettivamente ricevuto.

La Figura 6.1 mostra un esempio reale di errore di validazione generato da FastMCP durante l'invocazione di un endpoint ERP tramite Postman. In questo caso, il framework segnala una mancata conformità tra il valore restituito dall'API e il tipo definito nella specifica OpenAPI. Il campo ricevuto, infatti, contiene il valore numerico `0`, mentre lo schema associato all'endpoint prevede un valore di tipo booleano.

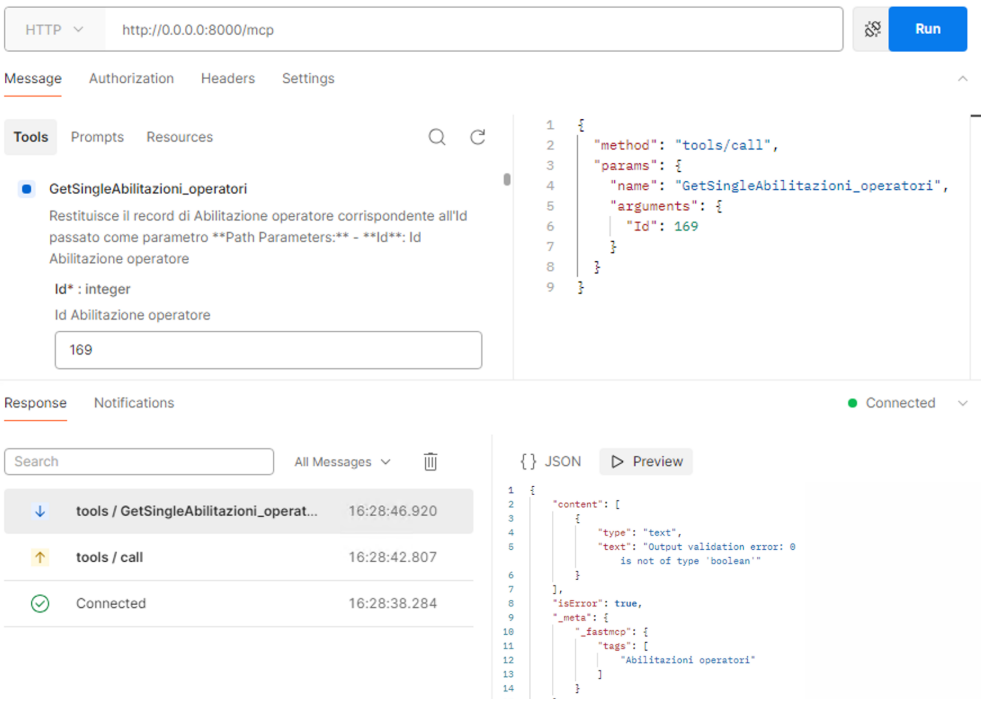


Figura 6.1: Esempio di errore di validazione generato da FastMCP durante l'invocazione di un'API REST

L'esempio evidenzia come le criticità non dipendano da errori sintattici nella specifica, ma principalmente dal modo in cui FastMCP applica le proprie regole di validazione interna che, in alcuni casi, si sono rivelate più restrittive rispetto a quanto previsto dallo standard OpenAPI.

6.5.2 Adattamento della specifica per la compatibilità con FastMCP

La rigidità del processo di validazione adottato da FastMCP sulle risposte provenienti dalle API REST ha reso necessario introdurre una fase aggiuntiva di adattamento della specifica OpenAPI. L'obiettivo è stato quello di migliorarne la compatibilità con il framework, senza, però, modificare il significato funzionale delle API descritte.

Per tale ragione, è stato sviluppato uno script di normalizzazione della specifica, riportato nel Listato 6.9, eseguito al termine della conversione da Swagger 2.0 a OpenAPI 3.x. Lo script interviene su alcune strutture della documentazione che, pur essendo formalmente valide, non risultavano pienamente compatibili con i controlli applicati da FastMCP. Ad ogni modo, le principali modifiche apportate riguardano sostanzialmente tre aspetti.

```
def convert_boolean_to_oneof(prop_data):
   if prop_data.get("type") == "boolean":
      return {
         "oneOf": [
            {"type": "boolean"},
            {"type": "integer", "enum": [-1, 0]}
         ],
         "description": prop_data.get("description", "")
      }
   return prop_data


def add_nullable_to_all_fields(schema):
   if "properties" not in schema:
      return

   for prop_name, prop_data in list(schema["properties"].items()):
      new_prop = convert_boolean_to_oneof(prop_data)
      if "nullable" not in new_prop:
         new_prop["nullable"] = True
      schema["properties"][prop_name] = new_prop


def is_paginated_endpoint(entry):
   """
   Controlla se l'endpoint ha parametri pageno e pagesize.
   """
   parameters = entry.get("parameters", [])
   names = [p.get("name") for p in parameters]
   return "pageno" in names and "pagesize" in names


def fix_endpoint_schemas(data):
   paths = data.get("paths", {})

   for path, methods in paths.items():
      for method, entry in methods.items():
         if "responses" not in entry:
            continue

         paginated = is_paginated_endpoint(entry)

         for code, response in entry["responses"].items():
            content = response.get("content", {})

            for mime, mime_data in content.items():
               schema = mime_data.get("schema")
               if not isinstance(schema, dict):
                  continue

               # Se l'endpoint èpaginato  forziamo type "object"
               if paginated:
                  if schema.get("type") == "array":
                     schema["type"] = "object"
                     # Rimuoviamo items perch object non ha items
                     schema.pop("items", None)
                     print(f"    Endpoint paginato rilevato: {method.upper()} {path}  type forzato a object
                           ")
               else:
                  # Altrimenti se èobject + items senza properties  convertiamo in array
                  if schema.get("type") == "object" and "items" in schema and "properties" not in schema:
                     schema["type"] = "array"

   return data
```

```
def process_openapi(data):
    # Aggiorna schemi globali
    components = data.get("components", {})
    schemas = components.get("schemas", {})

    for schema_name, schema_def in schemas.items():
        if schema_def.get("type") == "object":
            add_nullable_to_all_fields(schema_def)

    # Aggiorna schemi nei response degli endpoint
    fix_endpoint_schemas(data)

    return data
```

Listato 6.9: Normalizzazione della specifica OpenAPI per la compatibilità con FastMCP

La prima modifica riguarda la gestione dei campi booleani. Durante la fase di debug su Postman è emerso che alcuni attributi dichiarati come `boolean` nella specifica venivano restituiti dalle API con valori numerici, tipicamente `0` e `-1`, invece dei valori `true` e `false` previsti dal JSON Schema. Poiché FastMCP applica una validazione piuttosto rigida sui tipi di dato, questa differenza generava errori durante il controllo della struttura delle risposte. Per superare il problema, la funzione `convert_boolean_to_oneof()` trasforma tali campi in una struttura `oneOf`, consentendo di accettare sia valori booleani sia valori interi appartenenti all'insieme `{-1, 0}`.

La seconda modifica riguarda la gestione dei valori nulli. In alcuni casi, infatti, le API potevano restituire valori `null` per campi che nella specifica non erano esplicitamente indicati come `nullable`. Tale comportamento, però, causava problemi di validazione, poiché FastMCP richiede che la possibilità di ricevere un valore nullo sia dichiarata chiaramente nello schema. Per questa ragione, la funzione `add_nullable_to_all_fields()` attraversa le proprietà degli schemi di tipo `object` e aggiunge automaticamente il flag `nullable` ai campi che ne sono privi.

La Figura 6.2 mostra l'effetto dell'adattamento della specifica sullo stesso endpoint che nella Figura 6.1 generava un errore di validazione. Come si evince dalla figura, dopo l'aggiunta del flag `nullable` la risposta restituita dall'API viene accettata correttamente da FastMCP anche in presenza di campi valorizzati a `null`, come nel caso del campo `reg_name`.

La terza modifica, invece, riguarda la correzione degli schemi di risposta associati agli endpoint che implementano un meccanismo di paginazione. A tal proposito, la funzione `fix_endpoint_schemas()` individua gli endpoint che utilizzano parametri come `pageno` e `pagesize` e corregge eventuali incongruenze nella struttura della risposta. In particolare, quando la risposta di un endpoint con paginazione risulta dichiarata come `array`, il tipo di dato nello schema di risposta viene convertito in `object` e il campo `items` viene rimosso, poiché FastMCP si aspetta che le risposte paginate siano rappresentate come `object`. Al contrario, in alcuni casi, la specifica dichiarava come tipo di dato `object` le risposte che presentavano il campo `items` che, invece, è tipico degli `array`. In queste situazioni, la funzione riconduce lo schema al tipo `array`.

6.6 Trasformazione delle risposte restituite dalle API

Una volta implementato il meccanismo di invocazione delle API, l'attenzione si è focalizzata sul tema della presentazione dei dati restituiti dai servizi ERP.

Le risposte delle API, infatti, sono generalmente strutturate in formato JSON e possono contenere campi tecnici, informazioni annidate e valori non sempre facilmente interpretabili in un contesto conversazionale.

Inoltre, restituire all'LLM i file JSON grezzi avrebbe reso più difficile la generazione di una risposta chiara per l'utente finale.

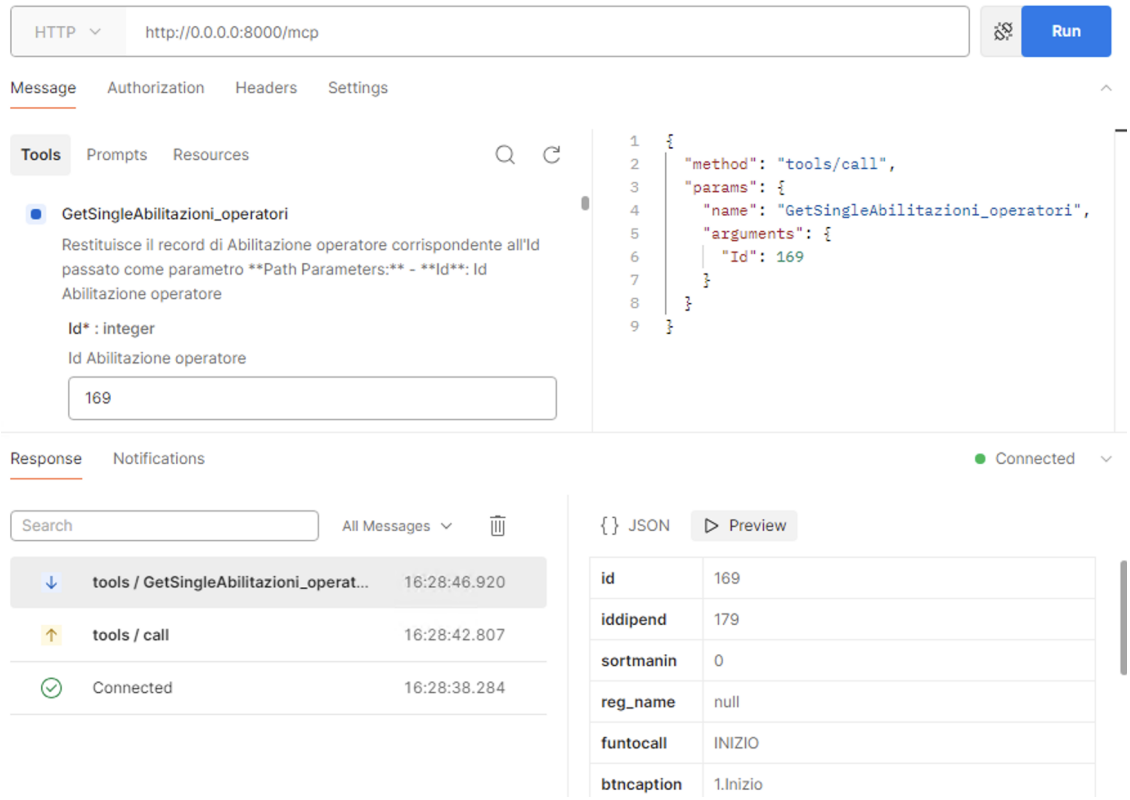


Figura 6.2: Esempio di risposta API accettata da FastMCP in seguito alla gestione dei valori `null`

Sebbene l'LLM sia in grado di rielaborare autonomamente una risposta JSON, si è scelto di introdurre una fase preliminare di trasformazione direttamente all'interno del server MCP. Tale scelta consente di fornire al modello dati già organizzati in una forma più leggibile e coerente, riducendo il rischio che vengano riportati campi tecnici non rilevanti o strutture poco adatte a un'interazione in linguaggio naturale.

Per questo motivo, si è scelto di implementare una funzione di trasformazione, denominata `format_json_readable()` e riportata nel Listato 6.10, che, tramite una visita ricorsiva, converte strutture dati arbitrariamente complesse in testo leggibile.

```
def format_json_readable(data, indent=0):
    """Trasforma qualsiasi JSON in testo leggibile"""
    testo = ""
    prefix = " " * indent

    if isinstance(data, dict):
        for k, v in data.items():
            if isinstance(v, (dict, list)):
                testo += f"{prefix}{k}:\n{format_json_readable(v, indent+1)}"
            else:
                # Trasforma flag/booleani comuni in Si/No
                if isinstance(v, int) and v in (-1, 0, 1):
                    v_str = "Si" if v == 1 else "No" if v in (0, -1) else str(v)
                else:
                    v_str = str(v)
                testo += f"{prefix}{k}: {v_str}\n"
    elif isinstance(data, list):
        for i, item in enumerate(data, start=1):
            testo += f"{prefix}- [{i}]\n{format_json_readable(item, indent+1)}"
    else:
        testo += f"{prefix}{str(data)}\n"

    return testo
```

Listato 6.10: Trasformazione delle risposte JSON in testo leggibile

Nel dettaglio, la funzione percorre la struttura JSON distinguendo tra dizionari, liste e valori scalari, applicando, a ciascuno, una rappresentazione testuale indentata che preserva

la gerarchia originale.

Un caso particolare riguarda i valori booleani restituiti dall'ERP come valori interi. Essi, infatti, vengono automaticamente convertiti nelle etichette `Sì` e `No`, rendendo il testo prodotto interpretabile da un LLM, senza richiedere alcuna conoscenza interna del sistema gestionale. Il risultato, infine, viene restituito all'LLM come output del tool `openapi_call`, che lo utilizza come base per formulare una risposta conversazionale all'utente.

6.7 Integrazione del server MCP con Gemini CLI

Terminata l'implementazione del server MCP, l'attenzione si è spostata sulla sua integrazione con Gemini CLI, già individuato in fase progettuale come componente host e client MCP. Gemini CLI rappresenta, infatti, l'ambiente attraverso cui l'utente interagisce con il modello Gemini di Google e, allo stesso tempo, consente all'LLM di accedere ai tool esposti dal server MCP.

Per rendere possibile il collegamento, il server MCP, innanzitutto, è stato avviato da terminale e reso raggiungibile tramite endpoint HTTP, come mostrato in Figura 6.3. Una volta in esecuzione, infatti, il server rimane in ascolto sull'host e sulla porta configurati.

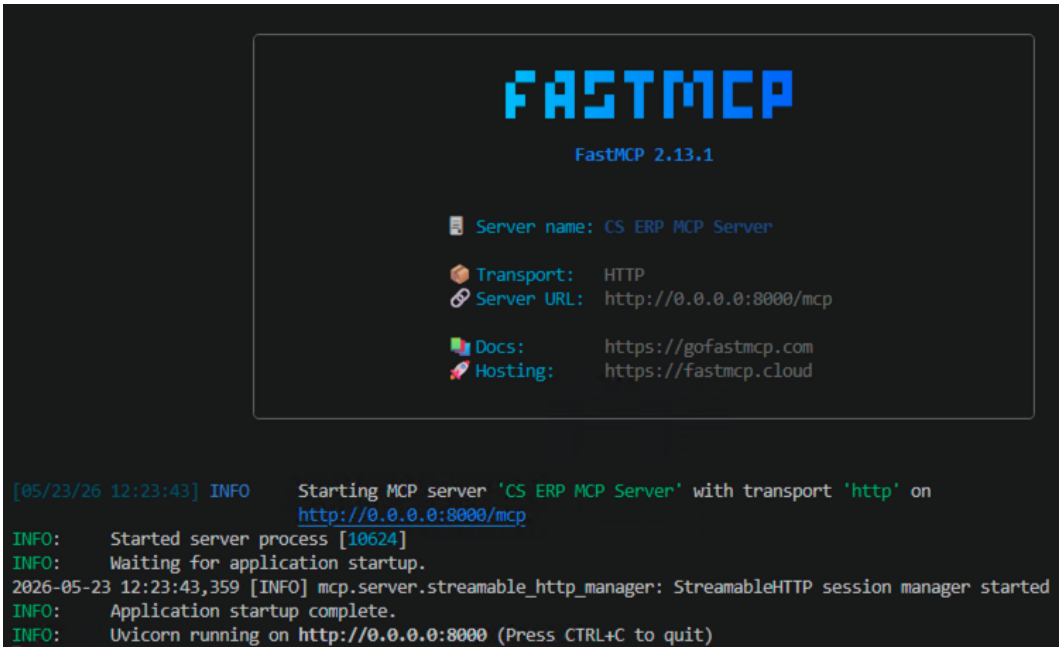


Figura 6.3: Avvio del server MCP da terminale

Dopo aver configurato l'API key necessaria per l'accesso ai modelli Gemini, è stato impostato il collegamento con il server MCP remoto tramite il file `settings.json` di Gemini CLI, riportato nel Listato 6.11. In particolare, all'interno della sezione `mcpServers` è indicato l'endpoint HTTP del server, così da permettere a Gemini CLI di raggiungerlo durante l'interazione con il modello linguistico.

```
{
  "security": {
    "auth": {
      "selectedType": "gemini-api-key"
    }
  },

  "mcpServers": {
    "CS ERP MCP Server": {
      "httpUrl": "http://localhost:8000/mcp"
    }
```

```
    }
  }
```

Listato 6.11: Configurazione del server MCP remoto nel file `settings.json` di Gemini CLI

Una volta completata la configurazione, Gemini CLI è in grado di rilevare automaticamente il server MCP e di rendere disponibili all'LLM i tool esposti. La corretta connessione può essere verificata tramite il comando `/mcp`, che consente di visualizzare lo stato dei server MCP configurati e l'elenco dei tool disponibili.

La Figura 6.4 mostra la schermata di Gemini CLI, in cui il server MCP risulta correttamente riconosciuto, confermando che l'integrazione con esso sia andata a buon fine.

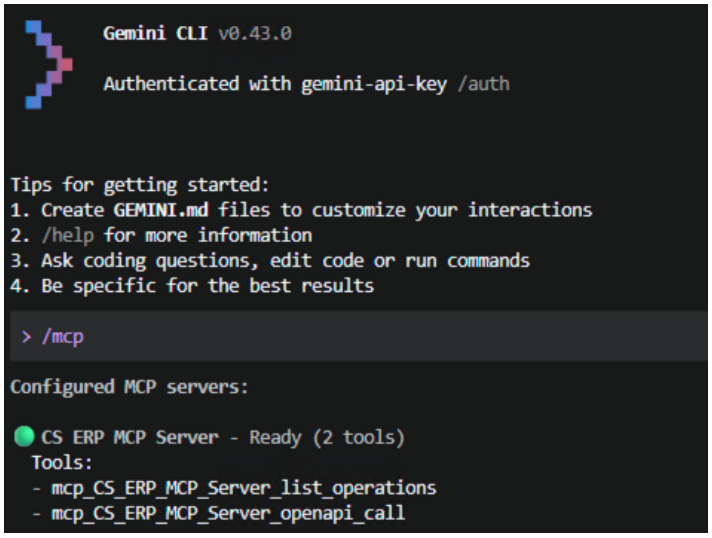


Figura 6.4: Verifica della connessione al server MCP tramite Gemini CLI

Da questo momento, infatti, il modello Gemini può interrogare i servizi ERP tramite i tool messi a disposizione dal server MCP, senza conoscere direttamente la struttura delle API REST, i dettagli degli endpoint o il processo di autenticazione sottostante.

6.7.1 Definizione delle istruzioni per il modello Gemini

Particolarmente interessante e stimolante, nell'ottica dell'integrazione con Gemini CLI, si è rivelata la definizione delle istruzioni da fornire al modello Gemini, affinché potesse utilizzare, in modo appropriato i tool messi a disposizione dal server MCP. La sola disponibilità dei tool, infatti, non è sufficiente a garantire che il modello li utilizzi correttamente; è necessario, piuttosto, guidarlo nella scelta dei tool e nella gestione dei dati restituiti dalle API.

A tal fine, è stato definito un prompt dedicato tramite il decoratore `@mcp.prompt`, riportato nel Listato 6.12, con l'obiettivo di orientare il comportamento di Gemini durante l'interazione con il sistema ERP. Tale prompt viene esposto dal server MCP e fornisce al modello una serie di istruzioni operative da seguire quando riceve una richiesta in linguaggio naturale da parte dell'utente.

```
@mcp.prompt
def erp_assistant(user_request: str) -> PromptMessage:
    """Guida il modello linguistico nell'utilizzo corretto dei tool MCP per interrogare i servizi ERP."""

    prompt_text = f"""
    Sei un assistente AI integrato con un server MCP che espone funzionalità aziendali provenienti da un
        sistema ERP.

    Il tuo compito ècomprendere la richiesta dell'utente in linguaggio naturale e, quando necessario,
        utilizzare i tool MCP disponibili per recuperare informazioni o eseguire operazioni sui servizi
        ERP.
```

```
Segui sempre queste regole operative:

1. Analizza la richiesta dell'utente
  - Comprendi l'intento della richiesta.
  - Individua eventuali informazioni rilevanti, come identificativi, codici, nomi, date, filtri o altri
       parametri utili.
  - Se la richiesta èambigua o mancano parametri indispensabili, chiedi chiarimenti all'utente prima di
       invocare i tool.

2. Scopri le operazioni disponibili
  - Per sapere quali API sono disponibili, utilizza sempre prima il tool `list_operations`.
  - Non cercare file locali, specifiche Swagger, file JSON o documentazione OpenAPI.
  - Non inventare operationId: scegli l'operazione più adatta solo tra quelle restituite da `
      list_operations`.

3. Seleziona l'operationId più coerente
  - Confronta la richiesta dell'utente con le descrizioni, i path e i metodi HTTP restituiti dal tool `
      list_operations`.
  - Individua l'operationId semanticamente più coerente con l'azione richiesta.
  - Se più operazioni sembrano simili, scegli quella più specifica rispetto alla richiesta dell'utente.
  - Se nessuna operazione disponibile ècoerente con la richiesta, non invocare `openapi_call` e
      comunica chiaramente all'utente che non èdisponibile un'API adatta a soddisfare la richiesta.

4. Invoca l'API tramite il tool corretto
  - Dopo aver individuato l'operationId, utilizza il tool `openapi_call`.
  - Passa sempre il parametro `operation_id`.
  - Usa `path_params`, `query_params` o `body` solo quando sono realmente necessari.
  - Se la richiesta dell'utente richiede informazioni provenienti da più API, puoi invocare più volte `
      openapi_call`, purch ogni chiamata sia basata su una operationId restituita da `list_operations`
       e sia coerente con la richiesta.
  - Non gestire direttamente endpoint, token, autenticazione o header di autorizzazione: questi aspetti
       sono gestiti dal server MCP.
  - Non effettuare paginazione automatica per scaricare tutti i record, a meno che l'utente lo richieda
       esplicitamente.

5. Gestisci i dati restituiti dall'API
  - Usa il risultato dell'API come informazione interna per costruire la risposta.
  - Non mostrare mai all'utente il JSON grezzo o dettagli tecnici non necessari.
  - Non riportare campi nulli, vuoti o puramente tecnici se non sono utili alla richiesta.
  - Riorganizza le informazioni in modo chiaro, sintetico e leggibile.

6. Formula la risposta finale
  - Rispondi sempre nella lingua dell'utente.
  - Fornisci una risposta naturale e comprensibile.
  - Se hai invocato più API, integra i risultati in un'unica risposta coerente, evitando ripetizioni e
      separando chiaramente le informazioni provenienti dalle diverse operazioni quando necessario.
  - Se l'API non restituisce dati utili, spiegalo in modo chiaro senza esporre dettagli tecnici.
  - Se si verifica un errore, comunica il problema in modo semplice, evitando messaggi grezzi
      provenienti dal sistema.
  - Non menzionare i passaggi interni, gli endpoint o gli operationId, salvo che l'utente lo chieda
      esplicitamente.

Il flusso corretto da seguire èquindi:
1. Comprendere la richiesta dell'utente.
2. Chiamare `list_operations`.
3. Individuare l'operationId o le operationId più adatte.
4. Chiamare `openapi_call` una o più volte con i parametri necessari.
5. Rielaborare il risultato, o i risultati ottenuti, in una risposta chiara e conversazionale.

Richiesta dell'utente:
---
{user_request}
---
"""

return PromptMessage(
   role="user",
   content=TextContent(
      type="text",
      text=prompt_text
   )
)
```

Listato 6.12: Prompt MCP per guidare il modello Gemini nell'utilizzo dei tool ERP

Nel dettaglio, il prompt guida il modello a partire dall'analisi della richiesta dell'utente. A tal proposito, Gemini deve, innanzitutto, comprendere l'intento e individuare eventuali informazioni utili all'invocazione dei tool, come codici identificativi, date, filtri o altri parametri rilevanti. Nel caso in cui la richiesta risulti ambigua o incompleta, il modello viene istruito a chiedere chiarimenti prima di procedere con qualsiasi operazione.

Un punto centrale delle istruzioni riguarda il modo in cui Gemini deve accedere alle API. Il modello, infatti, non deve cercare autonomamente nei file locali o nella documentazione OpenAPI. Al contrario, esso deve seguire un flusso preciso, ovvero utilizzare sempre prima il tool `list_operations` per consultare le operazioni esposte dal server MCP e, solo dopo aver individuato l' `operationId` più coerente con la richiesta dell'utente, in-

vocare `openapi_call` passando gli eventuali parametri necessari. Nel caso in cui tra le operazioni disponibili non sia presente alcuna API adatta a soddisfare la richiesta, il modello deve comunicarlo chiaramente all'utente, evitando di forzare l'invocazione di un tool non pertinente.

Le istruzioni chiariscono, inoltre, che il modello non deve occuparsi direttamente dei dettagli tecnici, come token, header di autorizzazione o meccanismi di autenticazione, poiché tali aspetti sono gestiti dal server MCP.

In questo modo, dunque, viene mantenuta una netta separazione tra il livello conversazionale, affidato all'LLM, e il livello tecnico-operativo, gestito, invece, dal server.

Infine, il prompt dedica particolare attenzione alla gestione dei risultati restituiti dalle API. A tal proposito, i dati ottenuti devono essere utilizzati per costruire la risposta finale ed evitare di mostrare all'utente file JSON grezzi, campi nulli o dettagli tecnici non rilevanti. Il risultato, dunque, deve essere riorganizzato in una forma chiara, sintetica e coerente con la richiesta formulata dall'utente.

CAPITOLO 7

Testing dell'ecosistema

In questo capitolo verrà presentata la fase di testing dell'ecosistema implementato, con l'obiettivo di osservare il comportamento dell'integrazione tra Gemini CLI e il server MCP per l'accesso in linguaggio naturale ai web service dell'ERP.

Nella prima sezione verranno introdotte le funzionalità di Gemini CLI considerate più rilevanti per seguire l'esecuzione dei test, come la visualizzazione delle chiamate ai tool e il meccanismo di approvazione Human-in-the-Loop.

Nella seconda sezione verranno mostrati alcuni scenari rappresentativi di utilizzo, scelti per evidenziare le capacità del sistema di gestire tipologie di richieste differenti. Inoltre, verranno riportate le chiamate manuali eseguite tramite l'interfaccia grafica di Postman, così da confrontare le risposte generate da Gemini con i risultati restituiti direttamente dai servizi ERP e valutare la coerenza dei dati ottenuti nei due casi.

Infine, il capitolo si concluderà con una discussione sui risultati ottenuti e sui principali limiti emersi.

7.1 Ambiente di test

Una volta effettuata l'integrazione tra Gemini CLI e il server MCP, una fase altrettanto importante del lavoro ha riguardato il testing dell'ecosistema implementato, con l'obiettivo di verificare che tutto funzionasse come previsto. In particolare, l'attenzione si è concentrata sul comportamento complessivo del flusso operativo, dalla richiesta formulata dall'utente all'interrogazione dei servizi ERP tramite il server MCP, fino alla restituzione della risposta finale.

I test sono stati svolti a partire da richieste formulate in linguaggio naturale, così da riprodurre un'interazione il più possibile vicina a un utilizzo reale. In questo modo, è stato possibile osservare come Gemini interpreta la richiesta dell'utente, consulta le operazioni disponibili tramite i tool MCP e individua l'operazione più coerente con la richiesta formulata. Una volta selezionata l'operazione, il modello invia al server MCP una richiesta strutturata, specificando l'operazione da eseguire e gli eventuali parametri necessari. Il server MCP, quindi, gestisce la parte tecnica dell'interazione con i servizi ERP, mentre Gemini utilizza i dati restituiti per generare una risposta comprensibile per l'utente.

A supporto delle verifiche, sono stati eseguiti anche test manuali sulle API tramite Postman. Questo passaggio ha permesso di confrontare le risposte ottenute passando attraverso il server MCP con quelle ottenute mediante chiamate dirette ai web service dell'ERP, al fine di verificare la coerenza dei dati restituiti nei due casi.

7.1.1 Funzionalità principali

Durante la fase di testing sono state prese in considerazione alcune funzionalità fornite dall'ambiente di interazione, utili per osservare e per valutare il comportamento dell'ecosistema. Esse sono:

- *Chat con streaming in tempo reale*: Gemini CLI consente di visualizzare la risposta mentre viene generata, senza attendere il completamento dell'intera elaborazione; questo rende più immediata l'interazione con Gemini e permette di seguire l'avanzamento della risposta durante l'esecuzione del test. Tale caratteristica offre un feedback immediato all'utente, migliorando la percezione di reattività del sistema.
- *Visualizzazione delle chiamate ai tool*: durante l'interazione, Gemini CLI mostra nello schermo le chiamate ai tool MCP che vengono effettuate nel corso della conversazione, insieme ai parametri necessari. Questa funzionalità rende più trasparente l'esecuzione del flusso e permette di verificare che vengano utilizzati correttamente il tool per consultare le API disponibili e quello per invocare l'API selezionata.
- *Gestione delle interruzioni Human-in-the-Loop*: Gemini CLI, prima di eseguire una chiamata verso le API, richiede sempre l'approvazione dell'operatore attraverso un pannello dedicato dal quale è possibile confermare o rifiutare l'azione proposta. Inoltre, in presenza di richieste ambigue, incomplete o non pienamente coerenti con le API disponibili, il modello può richiedere un chiarimento all'utente prima di procedere. Questo comportamento riduce il rischio di chiamate non pertinenti e introduce un maggiore controllo umano nell'interazione con il sistema ERP.

7.1.2 Flusso di esecuzione di una richiesta

Prima di entrare nel dettaglio dei singoli casi di test, è utile richiamare il percorso seguito da una richiesta all'interno dell'ecosistema implementato. In questo modo, gli esempi analizzati nelle sezioni successive possono essere letti con maggiore chiarezza, tenendo conto non solo della risposta finale, ma anche dei passaggi che la precedono.

L'interazione ha inizio quando l'utente formula una richiesta in linguaggio naturale nella chat di Gemini CLI. Il modello ne interpreta il contenuto e, quando per rispondere è necessario accedere ai dati dell'ERP, si affida al server MCP che espone le API del sistema gestionale come tool invocabili. In particolare, il primo passaggio consiste nella consultazione delle operazioni disponibili tramite il macro-tool `list_operations`, così da individuare l'`operationId` più coerente con l'esigenza espressa dall'utente.

Individuata l'operazione da eseguire, Gemini richiama il macro-tool `openapi_call`, passando l'`operationId` selezionato e gli eventuali parametri necessari. Da questo momento in poi, la parte tecnica dell'interazione viene gestita dal server MCP, che costruisce la richiesta HTTP verso il servizio ERP e riceve la risposta restituita dall'API in formato JSON.

Il risultato ottenuto, quindi, viene trasformato dal server MCP in una forma più leggibile prima di essere restituito al modello. Gemini utilizza tali informazioni come base per costruire la risposta finale, evitando di esporre all'utente dettagli tecnici, file JSON grezzi o informazioni non rilevanti.

Questo flusso rappresenta, dunque, la chiave di lettura per l'analisi degli scenari di utilizzo presentati nelle sezioni successive. Per ciascun caso, infatti, verranno considerati sia la richiesta dell'utente sia i passaggi intermedi che consentono al modello di generare la risposta finale.

7.2 Scenari di utilizzo e analisi delle conversazioni

Per dimostrare le capacità operative dell'ecosistema implementato, in questa sezione vengono presentati alcuni scenari rappresentativi di utilizzo. Gli esempi selezionati illustrano differenti tipologie di richieste che il sistema è in grado di gestire, dalle semplici interrogazioni informative fino a casi in cui è necessario gestire parametri specifici o combinare informazioni provenienti da più API dell'ERP.

7.2.1 Esempio di richiesta del recupero del dettaglio di un articolo tramite ID

Il primo scenario riguarda il recupero del dettaglio di un articolo a partire dal suo identificativo. Anche se si tratta di una richiesta semplice dal punto di vista dell'utente, l'esempio permette di osservare come un'informazione presente nella domanda, in questo caso l'ID dell'articolo, venga riconosciuta e passata al server MCP per eseguire la chiamata concreta verso l'API dell'ERP.

La conversazione viene avviata con una richiesta in linguaggio naturale nella quale l'utente indica l'articolo da consultare e chiede esplicitamente di ricevere un riepilogo delle sole informazioni principali, evitando campi tecnici, valori nulli o dati non rilevanti. Nello specifico, la richiesta può essere formulata nel seguente modo:

> *Recupera il dettaglio dell'articolo con ID 25 e restituisci un riepilogo chiaro delle informazioni principali. Mostra solo i dati utili per identificare l'articolo, evitando campi tecnici, valori nulli o informazioni non rilevanti.*

A partire da questa richiesta, Gemini interpreta l'intento dell'utente e riconosce che l'obiettivo è recuperare le informazioni relative a un singolo articolo. Poiché tali informazioni risiedono nei servizi ERP, il modello si affida ai tool messi a disposizione dal server MCP. Come previsto dalle istruzioni definite nel prompt MCP, inizialmente viene utilizzato il macro-tool `list_operations` attraverso il quale è possibile consultare le API rese disponibili dal server MCP sotto forma di tool invocabili.

Tra i tool disponibili, il modello individua `GetSingleArticoli`, coerente con la richiesta dell'utente, poiché permette di recuperare il record di un articolo a partire dal suo identificativo. A questo punto, Gemini prepara l'invocazione del macro-tool `openapi_call`, passando l'`operationId` selezionato e il parametro necessario, cioè l'ID dell'articolo indicato nella richiesta.

Prima dell'esecuzione effettiva della chiamata, Gemini CLI mostra sullo schermo i dettagli dell'invocazione proposta. In particolare, viene visualizzato il macro-tool che sarà utilizzato, cioè `openapi_call`, insieme all'`operationId` dell'API selezionata e al parametro individuato nella richiesta dell'utente. Questo passaggio consente all'operatore di verificare la coerenza dell'azione proposta prima di approvarla o rifiutarla. Nel caso analizzato, l'invocazione del tool viene approvata e il server MCP si interfaccia con il sistema ERP per eseguire la chiamata all'API.

Prima di analizzare la risposta prodotta dal modello, la stessa API è stata verificata manualmente tramite Postman, utilizzato come interfaccia di test collegata al server MCP.

Come mostrato in Figura 7.1, Postman consente di visualizzare il tool invocato e il parametro passato in input. Nella stessa schermata, inoltre, è riportato un estratto del file JSON restituito dall'API, sufficiente per evidenziare la struttura estesa della risposta. L'oggetto restituito, infatti, contiene numerosi campi tecnici, identificativi interni e valori nulli che, pur essendo utili dal punto di vista applicativo, non risultano rilevanti per la richiesta dell'utente.

A partire dal file JSON grezzo restituito dall'API, si osserva il passaggio dalla risposta tecnica dell'API alla risposta finale destinata all'utente. A tal proposito, il server MCP effettua una prima trasformazione della risposta dell'ERP, rendendola più leggibile e riducendo la

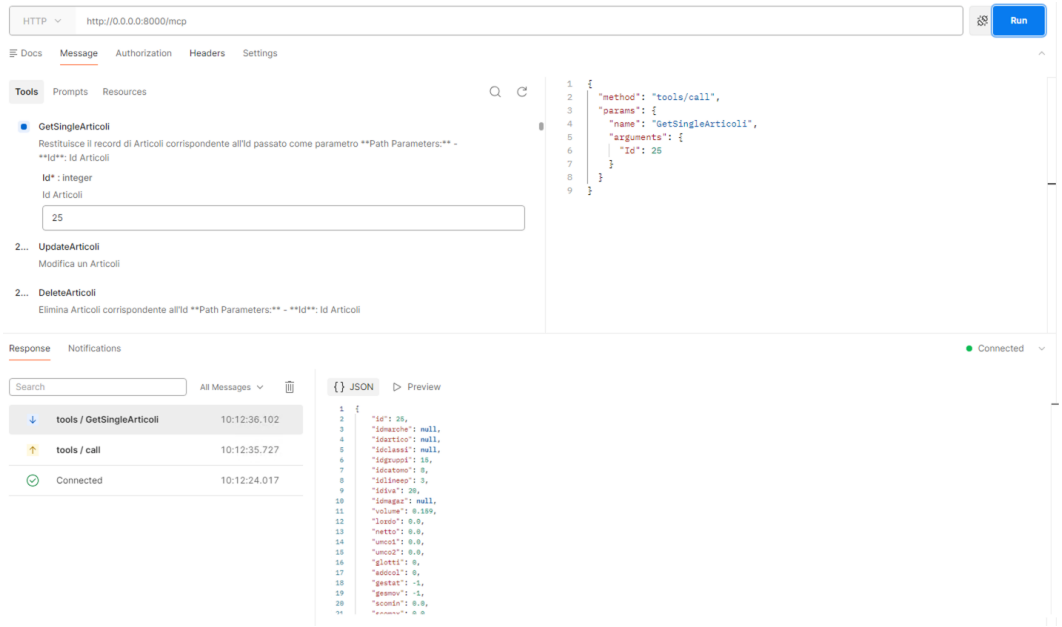


Figura 7.1: Verifica tramite Postman del tool `GetSingleArticoli` per il recupero del dettaglio dell'articolo con ID 25

presenza di campi non valorizzati. Successivamente Gemini utilizza i dati ricevuti per costruire un riepilogo sintetico, selezionando le informazioni più rilevanti rispetto alla richiesta iniziale e organizzandole in linguaggio naturale. La risposta finale restituita all'utente è riportata nel Listato7.1.

```
L'articolo con ID 25 corrisponde al codice ALIM00IMNOTR002, descritto come "ALIM IM-NOTR COLLA PER CARTONI
      ASTUCCI". L'unità di misura principale èKG e l'articolo risulta associato al gruppo 15, alla
      categoria 8, alla linea prodotto 3 e all'aliquota IVA 20. Sono inoltre disponibili alcune
      informazioni gestionali: il volume pari a 0,159, il costo standard 2,25, il tempo di consegna
      acquisto di 30 giorni e il lotto minimo di acquisto 1000. Per quanto riguarda le quantità, risultano
      una giacenza iniziale pari a 1997,85, una quantità impegnata pari a 0,85 e una quantità ordinata pari
       a 1020.
```

Listato 7.1: Risposta generata da Gemini a partire dai dati restituiti dall'API

Tale esempio mette in evidenza l'importanza delle descrizioni associate ai campi della risposta. Molti attributi restituiti dall'ERP utilizzano abbreviazioni o codifiche interne, che non sempre risultano immediatamente comprensibili. La presenza di descrizioni chiare nella specifica OpenAPI diventa, quindi, fondamentale per aiutare il modello a interpretare correttamente i dati e a selezionare le informazioni rilevanti per l'esigenza dell'utente.

Infine, il confronto tra la risposta grezza dell'API e quella restituita da Gemini mostra il ruolo di mediazione svolto sia dal modello linguistico sia dal server MCP. L'utente, di fatto, non viene esposto alla complessità della struttura del file JSON originale, ma riceve una risposta focalizzata sui dati più significativi, coerente con la richiesta iniziale e più adatta a un'interazione in linguaggio naturale.

7.2.2 Esempio di richiesta della disponibilità di un articolo a partire da una sua descrizione

Il secondo scenario riguarda la verifica della disponibilità di un articolo in una determinata data, a partire da una descrizione fornita dall'utente in linguaggio naturale. Rispetto al caso precedente, la richiesta è più articolata poiché l'utente non indica direttamente l'identificativo

dell'articolo, ma descrive semplicemente il prodotto di interesse. Prima di poter verificare la disponibilità, il sistema deve ricondurre tale descrizione a un articolo presente nell'ERP.

L'utente inizia la conversazione con la seguente richiesta:

"Verifica la disponibilità della resistenza da 10 OHM alla data del 30 giugno 2026".

A partire da tale richiesta, Gemini interpreta l'intento dell'utente e riconosce che, per fornire una risposta corretta, è necessario interrogare i servizi ERP tramite i tool messi a disposizione dal server MCP. Tuttavia, poiché l'endpoint di disponibilità richiede l'identificativo dell'articolo, occorre, innanzitutto, individuare il record corrispondente alla descrizione fornita.

Per fare ciò, Gemini utilizza il macro-tool `list_operations` per individuare, tra le operazioni disponibili, quella relativa alla consultazione dell'elenco degli articoli, ovvero `GetAllArticoli` che, successivamente, viene invocata tramite il macro-tool `openapi_call`. Poiché l'elenco degli articoli viene restituito in forma paginata, i record vengono analizzati progressivamente, confrontando le descrizioni degli articoli restituiti con quella indicata dall'utente. In questo modo, dunque, il sistema prova a individuare il prodotto più coerente con la richiesta, pur non partendo da un identificativo già noto.

Questo passaggio mette nuovamente in evidenza l'importanza della qualità delle descrizioni presenti nei dati ERP. Se le descrizioni degli articoli sono poco esaustive, abbreviate o basate su codifiche interne, il modello può incontrare maggiori difficoltà nel ricondurre correttamente la richiesta dell'utente al record corrispondente. Al contrario, descrizioni più chiare e complete facilitano il confronto tra il linguaggio naturale e i dati gestionali, rendendo più affidabile l'individuazione dell'articolo.

In ogni caso, una volta ricavato l'ID dell'articolo, l'informazione viene restituita a Gemini. A questo punto, però, il modello riconosce che la richiesta dell'utente non è ancora soddisfatta, poiché l'obiettivo non è individuare l'identificativo dell'articolo, ma verificarne la disponibilità nella data indicata.

Per completare il flusso, il modello utilizza nuovamente il macro-tool `list_operations` per individuare l'operazione dedicata alla disponibilità di un articolo e seleziona quella associata a `GetItemAvailability`. Successivamente, viene effettuata una seconda invocazione tramite il macro-tool `openapi_call`, passando l'`operationId` selezionato e i parametri necessari, ovvero l'id dell'articolo e la data di riferimento.

Prima dell'esecuzione effettiva della chiamata, Gemini CLI mostra sullo schermo l'invocazione proposta, permettendo all'operatore di controllare l'operazione selezionata e i parametri trasmessi al server MCP. Solo dopo l'approvazione dell'utente viene eseguita la chiamata verso il servizio ERP.

Anche per questo esempio, la verifica della disponibilità è stata osservata tramite Postman. Come mostrato in Figura 7.2, l'interfaccia consente di seguire il flusso delle chiamate effettuate: nella parte sinistra sono visibili prima la consultazione dell'elenco degli articoli e, successivamente, l'invocazione del servizio di disponibilità; nella parte destra sono, invece, riportati i parametri passati al tool e la risposta restituita dall'API.

Nel caso analizzato, il valore più rilevante per l'utente è rappresentato dal campo `availability`, pari a 1595 unità. Gli altri campi presenti nella risposta, come `trafficlight` e `idweb_fdispcla`, rappresentano, invece, indicatori interni del sistema ERP. Poiché essi non dispongono di una descrizione sufficientemente esplicativa e non sono necessari per rispondere alla richiesta formulata, non vengono esposti direttamente nella risposta finale.

La risposta generata dal modello è riportata nel Listato 7.2.

```
La resistenza da 10 OHM risulta disponibile alla data del 30 giugno 2026 con una quantità pari a 1595 unit
    à.
```

Listato 7.2: Risposta generata da Gemini per la verifica della disponibilità

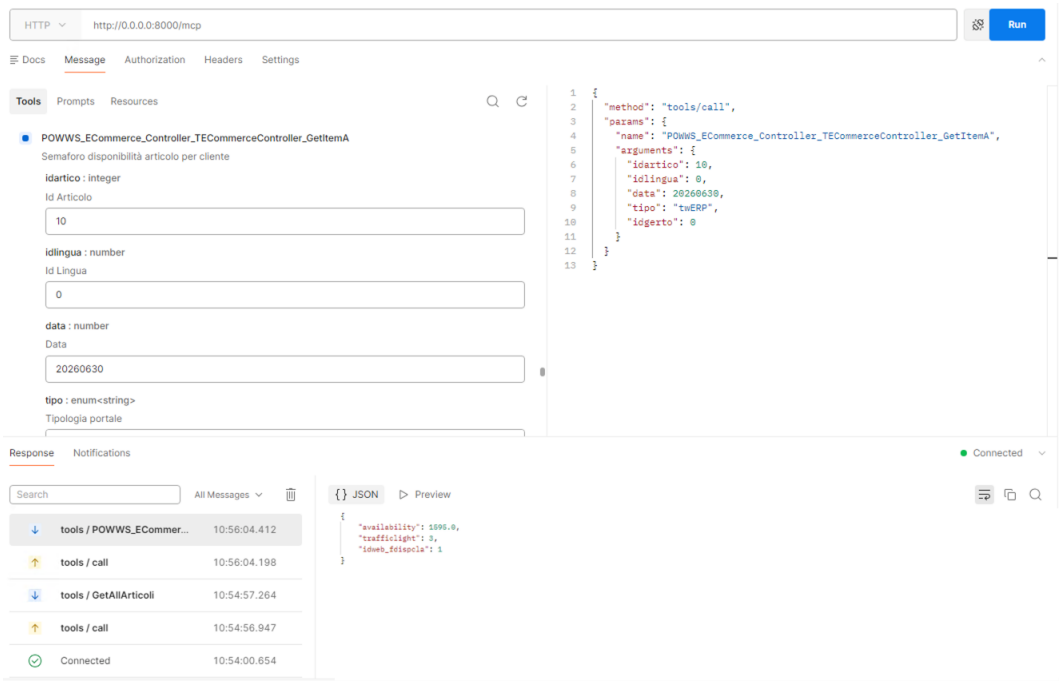


Figura 7.2: Verifica tramite Postman della ricerca dell'articolo e della sua disponibilità

Questo scenario evidenzia che Gemini non si ferma al primo risultato ottenuto, cioè l'identificazione dell'articolo, ma riconosce che tale informazione rappresenta solo un passaggio intermedio rispetto alla richiesta iniziale. Per questo motivo, il modello prosegue con una seconda invocazione API, utilizzando l'ID ricavato per interrogare il servizio di disponibilità.

L'esempio analizzato mostra, quindi, come l'ecosistema possa combinare più operazioni all'interno dello stesso flusso, trasformando una richiesta formulata in termini descrittivi in una risposta gestionale concreta e comprensibile per l'utente.

7.2.3 Esempio di richiesta di inserimento di un articolo

Il terzo scenario riguarda la gestione di una richiesta relativa all'inserimento di un nuovo articolo. A differenza degli esempi precedenti, incentrati sulla sola consultazione dei dati dell'ERP, questo caso permette di osservare il comportamento dell'ecosistema davanti a un'operazione che modifica i dati aziendali.

In particolare, si consideri il caso di un utente che non conosce nel dettaglio i parametri richiesti dall'API per procedere con l'inserimento. Prima di formulare la richiesta vera e propria, quindi, l'utente chiede quali informazioni siano necessarie per costruirla correttamente. Questo passaggio è utile perché consente di verificare come il sistema gestisca una domanda preliminare sui dati richiesti dall'API, senza eseguire immediatamente alcuna operazione di scrittura.

Per iniziare la conversazione, l'utente pone la seguente domanda:

"È possibile inserire un nuovo articolo? Se Sì, quali parametri sono necessari?".

A partire da questa richiesta, Gemini interpreta l'intento dell'utente e consulta le funzionalità disponibili tramite il macro-tool `list_operations`. Tra le API convertite dal server

MCP in tool invocabili, viene individuata quella relativa all'inserimento di un articolo, ovvero `InsertArticoli`. Poiché l'utente non sta ancora chiedendo di eseguire l'inserimento, ma vuole conoscere quali dati siano necessari per farlo, il modello utilizza le informazioni presenti nell'`inputSchema` dell'operazione individuata per spiegare quali parametri devono essere forniti, in quale formato e quali sono i valori ammessi.

Nello specifico, la risposta restituita dal modello è riportata nel Listato 7.3.

```
Sì, èpossibile inserire un nuovo articolo secondo lo schema ArticoModel. Per farlo, sono necessari i
      seguenti parametri obbligatori:

- codice: una stringa che identifica il codice dellarticolo;
- descr: una stringa che contiene la descrizione dellarticolo;
- staart: una stringa che indica lo stato dellarticolo. I valori ammessi sono:
   - G: gestito a magazzino;
   - S: solo anagrafico;
   - P: prodotto;
- tipart: una stringa che indica il tipo di articolo. I valori ammessi sono:
   - I: inventariale;
   - S: servizio;
   - C: costo;
   - F: fittizio.

Se vuoi procedere con l'inserimento di un articolo, sono necessari almeno questi valori.
```

Listato 7.3: Risposta di Gemini sui parametri richiesti per l'inserimento di un articolo

Dopo aver ricevuto tali informazioni, l'utente prova a fornire i parametri per l'inserimento del nuovo articolo. Il sistema, quindi, prepara l'invocazione dell'operazione `InsertArticoli`, passando nel body della richiesta i valori indicati dall'utente. Tuttavia, durante la validazione lato ERP, alcuni dati risultano incompleti rispetto alle regole richieste dal sistema gestionale.

In particolare, l'ERP segnala che, per l'articolo da inserire, è necessario specificare anche l'unità di misura inventariale. In ogni caso, l'errore non viene mostrato all'utente soltanto nella sua forma tecnica, ma viene rielaborato da Gemini in una risposta più comprensibile, come riportato nel Listato 7.4.

```
Invocazione del tool:
openapi_call{ "operation_id": "InsertArticoli","body": {"staart": "P","descr": "Condensatore ceramico 100
     nF 50V 0805","codice": "LET0CAP0CER010","tipart": "I"}}

Errore restituito dal sistema ERP:
Classname: ECSValidateResponseException
Items:
[1]:
   - Key: GESMOV_UM
   - Value: Per larticolo gestito a magazzino deve essere specificata lunità di misura inventariale.

Risposta breve:
L'inserimento dellarticolo non èstato completato. Il sistema richiede di specificare anche lunità di
      misura inventariale quando larticolo viene gestito a magazzino. Per procedere con l'inserimento, è
      quindi necessario indicare un valore valido per tale campo.
```

Listato 7.4: Tentativo di inserimento con parametri incompleti

Questo passaggio mostra come il sistema non si limiti a inoltrare la richiesta all'ERP, ma sia anche in grado di intercettare l'errore restituito dal servizio e di trasformarlo in un messaggio più chiaro per l'utente. In questo modo, l'utente viene guidato nella correzione dei dati mancanti prima di procedere con un nuovo tentativo di inserimento.

Alla luce dell'errore ricevuto, l'utente fornisce anche l'unità di misura inventariale richiesta. Giunti a questo punto, viene preparata una nuova invocazione dell'operazione `InsertArticoli`, includendo nel body il campo mancante. Prima dell'esecuzione effettiva, Gemini CLI mostra nuovamente sullo schermo i dettagli della chiamata proposta, così che l'operatore possa controllare i parametri e scegliere se approvare o rifiutare l'azione.

Nel caso analizzato, l'utente approva la chiamata e il server MCP inoltra la richiesta al servizio ERP. L'operazione viene completata con successo e il sistema restituisce l'identificativo assegnato al nuovo articolo, come mostrato nel Listato 7.5.

```
Invocazione del tool:
openapi_call{ "operation_id": "InsertArticoli","body": {"staart": "P","descr": "Condensatore ceramico 100
     nF 50V 0805","codice": "LET0CAP0CER010","umi": "KG","tipart": "I"}}

Id: 118774
Location: /api/avi/articolo/118774

Risposta breve:
L'articolo èstato creato con successo con ID 118774.
```

Listato 7.5: Tentativo di inserimento completato con successo

A completamento del test, è stata eseguita una successiva interrogazione tramite Postman. Come mostrato in Figura 7.3, è stato invocato il tool `GetSingleArticoli` passando l'identificativo restituito in seguito all'inserimento. La risposta conferma la presenza del nuovo articolo nel sistema ERP e riporta i dati principali forniti in fase di creazione, tra cui il codice e la descrizione.

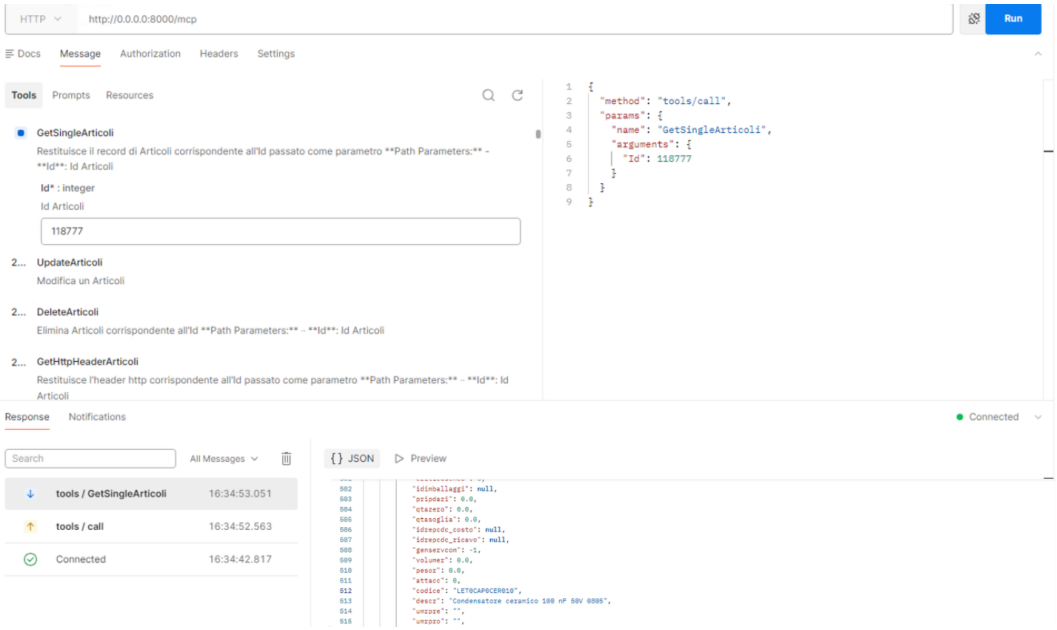


Figura 7.3: Verifica manuale tramite Postman dell'avvenuto inserimento dell'articolo

L'aspetto più rilevante di questo scenario non è soltanto l'inserimento effettivo dell'articolo, ma il processo di supporto alla compilazione della richiesta. In tal senso, il sistema aiuta l'utente a comprendere quali dati siano necessari, segnala eventuali informazioni mancanti e consente di correggere la richiesta prima di riprovare l'inserimento.

Inoltre, la presenza del controllo Human-in-the-loop prima dell'esecuzione della chiamata consente all'operatore di verificare i parametri prima che venga effettuata un'operazione modificativa sui dati aziendali.

7.2.4 Esempio di richiesta non supportata dalle API disponibili

L'ultimo scenario prende in considerazione una richiesta che, pur essendo plausibile dal punto di vista dell'utente, non può essere soddisfatta con le API esposte dal server MCP. In particolare, l'utente chiede di conoscere la posizione fisica del camion incaricato della consegna di un ordine di vendita. Il caso è utile perché consente di osservare il comportamento del sistema quando la richiesta supera le informazioni effettivamente disponibili nell'ERP.

La richiesta formulata dall'utente è la seguente:

"Dove si trova fisicamente il camion che sta consegnando l'ordine di vendita 155?".

A partire da questa richiesta, Gemini interpreta l'intento dell'utente e riconosce che la domanda riguarda un'informazione dinamica e aggiornata in tempo reale, come la posizione GPS del mezzo o il tracking fornito da un corriere.

Dalla consultazione delle operazioni disponibili emerge che il server MCP non espone funzionalità pensate per restituire la posizione geografica del mezzo o per collegarsi a servizi esterni di tracking. Per questo motivo, la richiesta non può essere soddisfatta direttamente. Gemini prova, quindi, a verificare almeno il dato di partenza fornito dall'utente, cioè l'ordine di vendita indicato, individuando l'operazione che permette di recuperarne la testata.

L'invocazione dell'operazione selezionata non restituisce però alcun ordine corrispondente all'identificativo indicato dall'utente. Il sistema ERP, infatti, risponde con un errore di risorsa non trovata, come riportato nel Listato 7.6.

```
Riscontro dalle API:
classname: ECSResourceNotFoundException
message: Resource TComTesRT (155) not found

Risposta restituita all'utente:
L'ordine di vendita con ID 155 non èstato trovato nel sistema ERP.

Inoltre, anche nel caso in cui lordine fosse presente, le API disponibili non permettono di conoscere la
      posizione fisica del camion in tempo reale. Il sistema ERP può fornire dati gestionali relativi all'
      ordine o alla spedizione, ma non espone un servizio di geolocalizzazione del mezzo o un'integrazione
      con sistemi di tracking dei corrieri.
```

Listato 7.6: Risposta a una richiesta di tracking logistico non supportata

Dalla risposta emerge un doppio limite. Da un lato, l'ordine di vendita indicato dall'utente non risulta presente nel sistema ERP; dall'altro, anche nel caso in cui l'ordine fosse stato trovato, le API disponibili non avrebbero comunque permesso di conoscere la posizione fisica del camion in tempo reale. L'ERP, infatti, avrebbe potuto restituire soltanto dati statici, come informazioni sull'ordine, eventuali riferimenti alla spedizione, vettore o data prevista, ma non dati GPS o informazioni di tracking aggiornate.

Tale comportamento è significativo perchè mostra che il sistema non tenta di forzare una risposta non supportata dalle API esposte dall'ERP. Al contrario, comunica il limite riscontrato in modo esplicito, distinguendo tra il mancato ritrovamento dell'ordine e l'assenza di un servizio di tracking logistico in tempo reale tra le API disponibili.

Inoltre, l'esempio evidenzia, ancora una volta, l'importanza delle istruzioni fornite al modello tramite il prompt MCP. A tal proposito, quando non esiste un'operazione coerente con la richiesta, il comportamento corretto non consiste nel cercare una risposta a tutti i costi, ma nel riconoscere il limite delle funzionalità disponibili e comunicarlo, in modo trasparente, all'utente. In questo modo, si riduce, dunque, il rischio di generare informazioni non verificabili o non supportate dal sistema ERP.

7.3 Discussione

Gli scenari descritti nelle sezioni precedenti hanno permesso di osservare il comportamento dell'ecosistema in situazioni diverse, passando da richieste semplici fino a casi più articolati, nei quali è stato necessario combinare più API in cascata o gestire eventuali condizioni non previste direttamente dalle API disponibili.

Nel complesso, i test hanno mostrato che l'integrazione tra Gemini CLI e il server MCP consente di gestire semplici richieste formulate in linguaggio naturale e di tradurle in strutture tecniche volte a richiamare le API dell'ERP. L'aspetto più rilevante riguarda il modo in cui il sistema accompagna la richiesta lungo l'intero flusso, dall'interpretazione della richiesta espressa dall'utente fino alla costruzione di una risposta finale comprensibile e coerente con i dati restituiti dall'ERP.

In questo senso, la fase di testing ha permesso di valutare l'ecosistema non solo dal punto di vista tecnico, ma anche rispetto alla sua capacità di rendere più accessibili i dati e le funzionalità normalmente esposte attraverso interfacce strutturate e poco leggibili per un utente non tecnico.

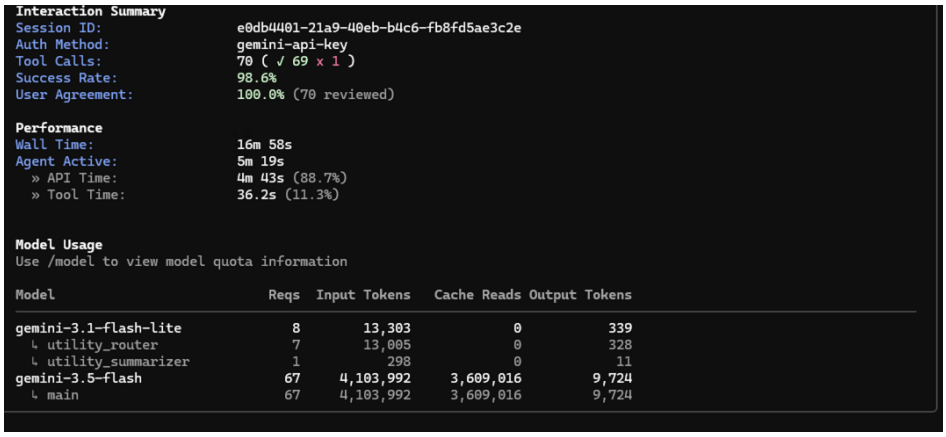
7.3.1 Osservazioni sui risultati ottenuti

Gli scenari analizzati hanno mostrato diversi livelli di complessità, dall'utilizzo diretto di parametri forniti dall'utente ai casi in cui è stato necessario combinare più invocazioni dei tool per soddisfare la richiesta o gestire errori restituiti dal sistema gestionale.

Un aspetto rilevante riguarda la capacità del sistema di gestire richieste che non possono essere soddisfatte con una singola invocazione di un tool. In questi casi, il flusso può proseguire in modo progressivo, utilizzando le informazioni ottenute da un primo passaggio per orientare le operazioni successive. In questo modo, la risposta finale viene costruita attraverso una sequenza di passaggi coerenti con l'obiettivo espresso dall'utente.

È risultato significativo anche il comportamento osservato nei casi meno lineari, ad esempio in presenza di errori di validazione o di richieste non supportate. In tali situazioni, il sistema non tenta di produrre una risposta non verificabile; al contrario, esso segnala il problema in modo comprensibile, indicando eventuali dati mancanti oppure dichiarando il limite delle API disponibili. Ciò contribuisce a rendere l'interazione più controllata e riduce il rischio di risposte fuorvianti.

Per completare l'analisi, sono state considerate le statistiche fornite da Gemini CLI al termine della sessione di test, riportate in Figura 7.4. È da precisare che tali dati non sono stati utilizzati come misura diretta della qualità delle risposte generate, bensì come indicazione del carico complessivo della sessione e dell'effettivo utilizzo dei tool MCP durante gli scenari analizzati.



Interaction Summary

Session ID:	e0db4401-21a9-40eb-b4c6-fb8fd5ae3c2e
Auth Method:	gemini-api-key
Tool Calls:	70 (✓ 69 x 1)
Success Rate:	98.6%
User Agreement:	100.0% (70 reviewed)

Performance

Wall Time:	16m 58s
Agent Active:	5m 19s
» API Time:	4m 43s (88.7%)
» Tool Time:	36.2s (11.3%)

Model Usage

Use /model to view model quota information

Model	Reqs	Input Tokens	Cache Reads	Output Tokens
gemini-3.1-flash-lite	8	13,303	0	339
↳ utility_router	7	13,005	0	328
↳ utility_summarizer	1	298	0	11
gemini-3.5-flash	67	4,103,992	3,609,016	9,724
↳ main	67	4,103,992	3,609,016	9,724

Figura 7.4: Le statistiche fornite da Gemini CLI

Un elemento rilevante delle statistiche è la possibilità di osservare come si distribuisca il tempo della sessione. A fronte di una durata complessiva pari a 16 minuti e 58 secondi, il tempo di attività effettiva dell'agente è pari a 5 minuti e 19 secondi. All'interno di tale intervallo, il tempo associato all'esecuzione dei tool risulta pari a 36,2 secondi, mentre la parte più consistente riguarda le chiamate al modello. Ciò suggerisce che, nei test svolti, il carico principale non è legato tanto all'invocazione tecnica dei tool MCP, quanto alla gestione del contesto, alla selezione delle operazioni più adatte e alla generazione delle risposte da parte del modello.

Per quanto riguarda l'utilizzo dei modelli, la sessione è stata gestita principalmente da `gemini-3.5-flash`, affiancato da componenti di supporto di `gemini-3.1-flash-lite`.

L'elevato numero di token in input è legato al tipo di interazione prevista da una sessione basata sull'MCP. In questo caso, infatti, il modello non elabora soltanto le richieste dell'utente, ma deve tenere conto anche del contesto tecnico necessario per interagire con il server MCP. In particolare, rientrano nel conteggio il prompt MCP, le descrizioni delle operazioni disponibili, gli schemi dei parametri, la cronologia della conversazione, le invocazioni dei tool e le risposte restituite dalle API dell'ERP.

Al contempo, la presenza di numerose letture della cache indica che una parte significativa del contesto è stata riutilizzata durante la sessione. Questo aspetto è utile soprattutto durante le lunghe sessioni caratterizzate da molte chiamate ai tool, poiché riduce la necessità di rielaborare continuamente le stesse informazioni.

Tuttavia, è importante precisare che le statistiche di Gemini CLI descrivono il comportamento della sessione, ma non permettono di stabilire se le risposte generate siano corrette dal punto di vista applicativo. Per questo motivo, la correttezza delle risposte è stata controllata tramite Postman, utilizzato come interfaccia grafica per eseguire chiamate dirette alle API e confrontare i risultati con quelli ottenuti passando attraverso il server MCP.

7.3.2 Limiti emersi

Nonostante i risultati ottenuti siano complessivamente positivi, la fase di testing ha messo in evidenza alcuni limiti dell'ecosistema realizzato.

Un primo aspetto riguarda la documentazione OpenAPI utilizzata come base per esporre le API al server MCP. Nel contesto del progetto, infatti, è stato necessario lavorare su una specifica ottenuta tramite conversione da un editor online, poiché la documentazione disponibile nell'infrastruttura aziendale non era inizialmente in un formato direttamente compatibile con FastMCP. Da un lato, ciò ha permesso di convertire automaticamente le API in tool invocabili; dall'altro, non si può escludere che tale passaggio abbia introdotto alcune imprecisioni nella struttura della specifica, nelle descrizioni dei campi o nella definizione degli schemi di input e di output.

Tale aspetto, inoltre, si collega direttamente alla qualità della documentazione OpenAPI di riferimento. Il server MCP, infatti, rende disponibili le API dell'ERP a partire dalle informazioni contenute nella specifica stessa; di conseguenza, descrizioni poco chiare o campi abbreviati possono rendere più difficile l'interpretazione dei dati da parte del modello. Tale criticità è emersa soprattutto nei casi in cui alcuni campi restituiti dall'ERP risultavano comprensibili dal punto di vista applicativo, ma non immediatamente interpretabili senza una descrizione più dettagliata. La qualità della documentazione, quindi, incide non solo sulla costruzione dei tool, ma anche sulla capacità del modello di selezionare le informazioni più rilevanti e di produrre una risposta esplicativa per l'utente.

Un'ulteriore criticità riguarda l'approccio adottato per rendere disponibili le API all'LLM. In tale contesto, la conversione automatica della specifica OpenAPI in tool invocabili tramite MCP si è rivelata adeguata nella realizzazione di una PoC, perché ha consentito di esporre rapidamente le funzionalità dell'ERP e di testarne l'utilizzo tramite Gemini CLI. Tuttavia, una conversione automatica delle API REST non può essere considerata, da sola, una soluzione definitiva per un contesto produttivo. Le API REST, infatti, sono progettate principalmente per essere invocate da applicazioni software o da sviluppatori che conoscono la struttura degli endpoint, i parametri richiesti e il significato delle risposte restituite. Tale modello di interazione, però, non coincide necessariamente con le esigenze di un agente linguistico, che, invece, deve selezionare l'operazione più adatta a partire da una richiesta formulata in linguaggio naturale e interpretare correttamente il significato dei dati restituiti.

Questa criticità, inoltre, è strettamente collegata al consumo di token osservato nelle statistiche di Gemini CLI. Le specifiche OpenAPI, soprattutto in contesti aziendali complessi, possono contenere molti endpoint, ciascuno con descrizioni, parametri e schemi associati.

Anche se nel lavoro presentato tale criticità è stata parzialmente mitigata attraverso l'utilizzo dei due macro-tool generici `list_operations` e `openapi_call`, il modello deve comunque gestire una quantità significativa di contesto tecnico per comprendere le operazioni disponibili e il modo in cui esse devono essere utilizzate.

In una prospettiva produttiva, dunque, questo aspetto può incidere sia sull'efficienza sia sulla capacità del modello di concentrarsi sulle informazioni rilevanti per il task da svolgere.

CAPITOLO 8

Conclusioni e sviluppi futuri

Il presente lavoro di tesi ha avuto come obiettivo la progettazione e l'implementazione di un ecosistema basato sulla Generative AI per l'accesso in linguaggio naturale ai web service di un sistema ERP. In particolare, l'attenzione si è focalizzata sulla capacità del sistema di interpretare le richieste formulate dall'utente, di estrarre i parametri rilevanti e di generare automaticamente la richiesta tecnica nel formato previsto dall'ERP.

Per garantire una totale chiarezza del caso di studio, nelle fasi iniziali del lavoro è stato presentato un *excursus* sulle principali tappe che hanno portato alla diffusione dei Large Language Model e sui loro principali limiti. Tra questi, è stata approfondita soprattutto la natura statica della conoscenza appresa in fase di addestramento, che rende necessario collegare tali modelli a dati e servizi esterni per ottenere informazioni aggiornate e coerenti con lo stato effettivo dei sistemi interrogati.

A partire da queste considerazioni, è stato approfondito il ruolo del function calling e, successivamente, del Model Context Protocol, individuando nell'MCP un approccio utile per rendere più uniforme e modulare l'interazione tra i modelli linguistici e i sistemi applicativi esterni.

Terminata la fase introduttiva, lo studio si è, poi, concentrato sul contesto aziendale di riferimento, rappresentato dal sistema SAM ERP2 e dai web service RESTful esposti a supporto delle sue funzionalità. In questa fase sono state analizzate le caratteristiche del sistema e sono state individuate le esigenze operative da soddisfare, in coerenza con gli obiettivi del progetto.

In seguito è stata definita l'architettura dell'ecosistema, scegliendo il framework low-code FastMCP, per la realizzazione del server MCP, e Gemini CLI, come ambiente di interazione con il modello. Una parte rilevante del lavoro è consistita nella preparazione della specifica OpenAPI, con particolare attenzione alla sua conversione in un formato compatibile con FastMCP, e nella progettazione dei tool attraverso cui rendere invocabili le API dell'ERP. A tal proposito, per evitare di esporre un numero eccessivo di operazioni direttamente al modello, è stata adottata una soluzione basata su due macro-tool; il primo consente di consultare le operazioni disponibili, mentre il secondo permette di invocare quella più coerente con la richiesta dell'utente. Dopodichè, sono stati implementati il processo di autorizzazione basato sull'utilizzo del token JWT e la trasformazione delle risposte restituite dalle API, con l'obiettivo di rendere i dati più leggibili e adatti alla successiva rielaborazione da parte dell'LLM.

Infine, la fase di testing ha permesso di osservare il comportamento dell'ecosistema in differenti scenari di utilizzo. I risultati ottenuti hanno mostrato che l'integrazione tra Gemini CLI, server MCP e servizi ERP consente di trasformare semplici richieste formulate in linguaggio naturale in interazioni operative con il sistema gestionale, riducendo la necessità per l'utente di conoscere gli endpoint, i parametri tecnici o la struttura delle risposte JSON. Al contempo, il lavoro ha messo in evidenza alcuni limiti importanti. In primo luogo, è emersa la forte dipendenza dalla qualità della documentazione OpenAPI, poiché descrizioni incomplete, campi abbreviati o schemi poco esplicativi possono rendere più difficile l'interpretazione delle operazioni disponibili e dei dati restituiti dall'ERP. In secondo luogo, è stato messo in evidenza il consumo di contesto per il modello, dovuto alla necessità di gestire un'enorme quantità di informazione, quali le descrizioni delle operazioni, i parametri e i file JSON grezzi provenienti dall'ERP.

In questa prospettiva, il sistema implementato, nella sua configurazione attuale, rappresenta un Proof of Concept utile a dimostrare la fattibilità dell'approccio, pur lasciando spazio a ulteriori miglioramenti e ottimizzazioni.

In virtù di quanto emerso, un primo ambito di sviluppo riguarda la progettazione del livello MCP in modo più mirato rispetto a quanto realizzato nel PoC. Nel lavoro svolto, infatti, il server MCP è stato generato automaticamente a partire dalla specifica Swagger dell'ERP, così da rendere rapidamente disponibili le API come tool invocabili dall'LLM. Questo approccio si è rivelato utile nella fase di prototipazione, poiché ha permesso di verificare, in tempi piuttosto rapidi, la fattibilità dell'integrazione tra l'LLM, il server MCP e il sistema ERP. Tuttavia, nel momento in cui si dovesse pensare alla fase di produzione, una conversione automatica della documentazione potrebbe non rappresentare la soluzione più adatta.

Una possibile evoluzione consisterebbe, quindi, nel progettare tool MCP più orientati alle funzionalità effettive dell'azienda, piuttosto che esporre direttamente le singole API REST. In questo modo, più chiamate API necessarie per completare una determinata operazione potrebbero essere incapsulate all'interno di un unico tool, riducendo la complessità del flusso e guidando meglio il modello nella scelta delle azioni da eseguire. In questa direzione, un eventuale supporto potrebbe essere rappresentato da Arazzo, una specifica rilasciata dall'OpenAPI Initiative, pensata per descrivere workflow applicativi composti da più chiamate API finalizzate al raggiungimento di un risultato specifico. Rispetto a una documentazione basata soltanto sui singoli endpoint, Arazzo consente di rappresentare, in modo più esplicito, l'ordine delle operazioni, le dipendenze tra i diversi passaggi e il modo in cui i dati ottenuti da una chiamata API possono essere utilizzati in quelle successive.

Tale aspetto diventa particolarmente rilevante se si considera il ruolo, sempre più centrale, degli agenti linguistici come consumatori diretti di API. Un agente che può scoprire e interpretare un workflow ben descritto risulta, infatti, più affidabile rispetto a un agente costretto a ricostruire il comportamento atteso a partire dai singoli endpoint isolati. In questo senso, Arazzo permette di rendere i workflow un elemento documentale esplicito e leggibile, sufficientemente strutturato per supportare l'automazione ma, allo stesso tempo, comprensibile anche per gli utenti finali.

Un ulteriore ambito di miglioramento riguarda l'introduzione di una metodologia di valutazione più strutturata. A tal proposito, sarebbe utile definire casi di test ripetibili e criteri di valutazione espliciti, così da analizzare, in modo più sistematico, la correttezza delle risposte e la coerenza dei tool selezionati. A supporto di questa fase, inoltre, potrebbero essere utilizzati approcci come LLM-as-a-judge, oppure strumenti di valutazione specifici per gli ecosistemi MCP, in grado di considerare non solo la risposta finale, ma anche il flusso di interazione tra l'LLM, il server MCP e i tool invocati.

Infine, nel momento in cui si dovesse pensare alla fase di produzione, sarebbe necessario

approfondire gli aspetti legati alla sicurezza, introducendo meccanismi di autorizzazione più granulari e una distinzione più rigorosa tra le operazioni di sola lettura e le operazioni di scrittura.

Il sistema proposto, in virtù di ciò, può costituire una base solida per future evoluzioni dell'ecosistema, orientate alla progettazione di tool più robusti, sicuri e maggiormente aderenti ai processi aziendali. Il lavoro svolto evidenzia, quindi, il potenziale degli LLM come interfaccia intelligente verso i sistemi ERP, aprendo la strada a modalità di accesso ai dati aziendali più naturali, controllate e coerenti con le esigenze operative degli utenti.

Bibliografia

ACERBI, A. e STUBBERSFIELD, J. M. (2023), «Large language models show human-like content biases in transmission chain experiments», .

ALTO, V. (2024), *Building LLM Powered Applications*, Packt Publishing.

BRUCKS, M. e TOUBIA, O. (2025), «Prompt architecture induces methodological artifacts in large language models», .

DENG, Z., GUO, Y., HAN, C., MA, W., XIONG, J., WEN, S. e XIANG, Y. (2025), «AI Agents Under Threat: A Survey of Key Security Challenges and Future Pathways», .

FLEURET, F. (2023), *The Little Book of Deep Learning*, Wild Acres Books.

FRANKLIN, M., TOMAŠEV, N., JACOBS, J., LEIBO, J. Z. e OSINDERO, S. (2026), «AI Agent Traps», .

GULLÍ, A. (2025), *Agentic Design Patterns: A Hands-On Guide to Building Intelligent Systems*, Springer Nature Switzerland.

GUO, Y., GUO, M., SU, J., andMENGQIU ZHU, Z. Y., LI, H. e andSHUO SHUO LIU, M. Q. (2024), «Bias in Large Language Models: Origin, Evaluation, and Mitigation», *arXiv:2411.10915*.

HAMMOND, L., CHAN, A., CLIFTON, J., HOELSCHER-OBERMAIER, J., KHAN, A., MCLEAN, E., SMITH, C., BARFUSS, W. e OTHERS (2025), «Multi-Agent Risks from Advanced AI», *arXiv:2502.14143*.

HOU, X., ZHAO, Y., WANG, S. e WANG, H. (2026), «Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions», *ACM Computing Surveys*.

HUA, Q., YE, L., FU, D., XIAO, Y., CAI, X., WU, Y., andJUNFEI WANG, J. L. e LIU, P. (2026), «Context Engineering 2.0: The Context of Context Engineering», *arXiv:2510.26493*.

LUMER, E., NIZAR, F., GULATI, A., BASAVARAJU, P. H. e SUBBIAH, V. K. (2025), «Tool-to-Agent Retrieval: Bridging Tools and Agents for Scalable LLM Multi-Agent Systems», *arXiv:2511.01854*.

MOHAN, R., AGRAWAL, S. e SUNIL, S. (2025), *Multi-Agent Systems with AutoGen: Explore Agentic AI and Build LLM-Powered Applications*, Orange Education.

PARK, J. S., O'BRIEN, J. C., CAI, C. J., MORRIS, M. R., LIANG, P. e BERNSTEIN, M. S. (2023), «Generative Agents: Interactive Simulacra of Human Behavior», *arXiv:2304.03442*.

PHOENIX, J. e TAYLOR, M. (2024), *Prompt Engineering for Generative AI: Future-proof Inputs for Reliable AI Outputs at Scale*, O'Reilly Media.

PRINCE, S. J. (2023), *Understanding Deep Learning*, The MIT Press.

SAYFAN, G. (2026), *Design Multi-Agent AI Systems Using MCP and A2A: Engineer Your Own Python-based Agentic AI Framework with Tool Use, Memory, and Multi-agent Workflows*, Packt Publishing.

SCHICK, T., DWIVEDI-YU, J., DESSÌ, R., RAILEANU, R., LOMELI, M., ZETTLEMOYER, L., CANCEDDA, N. e SCIALOM, T. (2023), «Toolformer: Language Models Can Teach Themselves to Use Tools», *arXiv:2302.04761*.

SHEN, Z. (2024), «LLM With Tools: A Survey», .

TUNSTALL, L., VON WERRA, L. e THOMAS (2022), *Natural Language Processing with Transformers*, O'Reilly Media.

VASWANI, A., SHAZEER, N., PARMAN, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., ŁUKASZ KAISER e POLOSUKHIN, I. (2017), «Attention is All You Need», .

Sitografia

- Anthropic, risorse di approfondimento sull'intelligenza artificiale – https://www.anthropic.com/
- arXiv, archivio di paper scientifici – https://arxiv.org/
- Baeldung, piattaforma di formazione – https://www.baeldung.com/cs/
- Datapizza Blog, blog con news in tema AI – https://datapizza.tech/it/blog
- Descope Blog, blog con risorse e approfondimenti – https://www.descope.com/blog/developers
- Exploring Language Models, newsletter con articoli e guide visive in ambito AI – https://www.maartengrootendorst.com/blog/
- FastMCP, framework per applicazioni basate sull'MCP – https://gofastmcp.com/getting-started/welcome
- Gemini, modelli LLM di Google – https://gemini.google.com/
- Gemini CLI, agente AI di Google – https://geminicli.com/
- Github, servizio web di hosting per progetti software – https://github.com/
- Hugging Face, piattaforma per modelli di machine learning – https://huggingface.co/
- IBM, risorse su intelligenza artificiale e NLP – https://www.ibm.com/think
- Medium, piattaforma di blogging e di pubblicazione di contenuti – https://medium.com/
- Microsoft Developer, portale ufficiale dedicato agli sviluppatori – https://developer.microsoft.com/it-it/
- Model Context Protocol, documentazione ufficiale – https://modelcontextprotocol.io/docs/
- OpenAI, documentazione API e modelli GPT – https://platform.openai.com/
- OpenAPI Initiative, specifiche e documentazione delle API – https://www.openapis.org/

- Rizzo AI Academy, community tech – `https://www.rizzoaiacademy.com/`
- Swagger Editor, editor che supporta specifiche API e formati di serializzazione – `https://swagger.io/tools/swagger-editor/`