ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione di un Chatbot Agentico per il Controllo di Sciami di Droni

Tesi di laurea in:
Programmazione ad Oggetti

Relatore
Prof. Mirko Viroli

Candidato
Federico Mariani

Correlatori
Dott. Gianluca Aguzzi
Dott. Nicolas Farabegoli

I Sessione di Laurea
Anno Accademico 2025-2026

"A te che mi hai insegnato i sogni e l'arte dell'avventura
A te che credi nel coraggio e anche nella paura"
A te Papà.

Sommario

La crescente complessità dei sistemi robotici rende necessario sviluppare interfacce accessibili anche a utenti privi di competenze specialistiche. I modelli linguistici di grandi dimensioni (Large Language Models, LLM) rappresentano in questo senso uno strumento promettente, in quanto consentono di interagire con sistemi complessi attraverso il linguaggio naturale.

Un caso emblematico è quello di Project Emerge, un sistema per il controllo di uno sciame di droni tramite un'interfaccia che presuppone la conoscenza di terminologie e concetti specifici del dominio. Il presente lavoro descrive la progettazione, lo sviluppo e la validazione di un chatbot basato su un'architettura agentica, concepito come livello di astrazione tra l'utente e il sistema di controllo: l'agente interpreta richieste in linguaggio naturale, individua la formazione più appropriata, guida l'utente nella configurazione dei parametri e invia i comandi allo sciame tramite protocollo MQTT.

Il sistema è stato sviluppato in Java mediante la libreria *LangChain4j*, adottando un'architettura modulare ed estendibile che consente di sostituire il modello linguistico sottostante senza modificare la struttura complessiva del sistema. A corredo del progetto, una fase di validazione sperimentale confronta le prestazioni di dieci modelli di intelligenza artificiale, appartenenti a tre provider differenti (Gemini, OpenAI e Ollama), su un dataset di undici richieste specifiche: tra i modelli cloud, *gpt-4.1-mini* ottiene il miglior compromesso tra accuratezza e velocità di risposta, mentre tra i modelli locali *qwen2.5:3b* si rivela il più performante.

I risultati confermano che l'approccio implementato è utilizzabile ed efficace, e che l'architettura adottata è modulare, estendibile e trasferibile ad altri scenari di interazione uomo-macchina mediata dal linguaggio naturale.

Indice

Elenco delle figure

Elenco dei listati

Capitolo 1

Introduzione

La crescente diffusione di sistemi software e robotici sempre più complessi ha reso evidente la necessità di sviluppare modalità di interazione accessibili anche a utenti privi di competenze specialistiche. Tradizionalmente, il controllo di tali sistemi richiedeva interfacce dedicate e una conoscenza approfondita del loro funzionamento. Lo sviluppo dell'Intelligenza Artificiale (IA) applicata al linguaggio naturale sta progressivamente modificando questo paradigma, introducendo modalità di interazione più intuitive e naturali.

Nel corso dell'ultimo decennio, l'IA ha vissuto una profonda e rapida evoluzione. Si è passati da sistemi rigidi, basati su regole logiche predeterminate, a modelli probabilistici avanzati capaci di comprendere il contesto delle richieste umane. È proprio all'interno di questa branca dell'IA che si collocano i modelli di linguaggio di ultima generazione, noti come Large Language Models (LLM). Essi costruiscono un ponte tra l'uomo e la macchina; tuttavia, mentre le soluzioni adottate fino ad oggi richiedevano la conoscenza dei linguaggi di programmazione, questa nuova tecnologia permette di comunicare direttamente attraverso il linguaggio umano.

I chatbot sono software progettati per simulare una conversazione con un essere umano. Questa interazione non implica necessariamente l'utilizzo di un'IA: i primi sistemi, infatti, erano programmati sulla base di un insieme molto limitato di comandi predefiniti, ai quali il software rispondeva secondo regole rigide e preimpostate. Nel tempo, tali sistemi sono diventati parte integrante della quotidianità, in particolar modo sotto forma di servizi come ChatGPT o Gemini, che

adottano un'interfaccia utente basata sulla chat, molto simile a quella delle comuni applicazioni di messaggistica, cui è possibile rivolgere richieste di varia natura. La diffusione degli LLM si inserisce esattamente in questo contesto, consentendo l'utilizzo di un chatbot senza la necessità di conoscere alcuna sintassi specifica.

Le potenzialità degli LLM, tuttavia, non si esauriscono nella capacità di comprendere e rispondere all'utente. Per interagire con il mondo esterno, questi modelli necessitano di un punto di contatto con l'ambiente fisico e digitale circostante: è in questo scenario che si collocano gli Agenti (*AI Agents*), che rappresentano un'evoluzione dei classici assistenti virtuali. A differenza di questi ultimi, un agente non si limita a conversare, ma agisce per completare compiti complessi attraverso *tools* messi a disposizione dal programmatore, come l'accesso a database o a Application Programming Interface (API), che fungono da interfacce operative per interagire con l'ambiente esterno. Il modello linguistico funge da componente decisionale, determinando autonomamente quando e quale tool utilizzare, mentre i tool stessi costituiscono le componenti operative attraverso cui l'agente produce effetti concreti.

Il caso d'uso specifico in cui è stata testata questa architettura agentica è il sistema dimostrativo *Project Emerge* [FAB+26], una piattaforma per il controllo di uno sciame di droni basata sulla tecnica dell'*aggregate computing*, un paradigma di programmazione collettiva in cui il comportamento dell'intero sciame viene espresso come un unico programma distribuito. Il controllo dello sciame non avviene tramite pilotaggio diretto, ma attraverso la selezione di configurazioni e formazioni predefinite, ognuna delle quali richiede l'impostazione di parametri specifici. Questo rappresenta un caso d'uso particolarmente sfidante per un sistema conversazionale: un errore di configurazione si traduce direttamente in un comportamento fisico indesiderato dello sciame, rendendo la fase di validazione e guida dell'utente un requisito critico. Il chatbot sviluppato si colloca come livello di astrazione tra l'utente e il sistema di controllo: l'utente esprime le proprie esigenze in linguaggio naturale, mentre il sistema interpreta la richiesta, individua la formazione più appropriata, guida la configurazione dei relativi parametri e traduce il risultato finale nei comandi comprensibili da *Project Emerge*. In questo modo non è necessario istruire direttamente l'utente all'utilizzo dello sciame e della sua interfaccia: sarà sufficiente fornire all'agente le informazioni necessarie affinché

possa rispondere autonomamente alle domande e alle richieste dell'utente.

Il presente lavoro descrive la progettazione, lo sviluppo e la validazione di tale assistente virtuale. Sul piano architetturale, il sistema affronta problematiche legate all'integrazione di modelli linguistici con strumenti esterni, realizzata in Java mediante la libreria *LangChain4j*, alla gestione del contesto conversazionale e alla comunicazione con il sistema di controllo dello sciame. Un'attenzione particolare è stata dedicata alla modularità: l'architettura consente di sostituire il modello linguistico sottostante e di estendere le funzionalità disponibili senza modificare la struttura complessiva del sistema. A integrazione del progetto, viene presentata una fase di validazione sperimentale che confronta le prestazioni di diversi modelli di IA, sia locali che in cloud, rispetto ai requisiti specifici del sistema. I capitoli successivi analizzano l'architettura software del prototipo, le logiche di integrazione dei tool e i risultati della fase di valutazione.

Questa tesi è divisa in capitoli e sottocapitoli ciascuno con uno scopo preciso. Nel Capitolo 2 vengono introdotti i concetti più importanti per la comprensione del progetto e dell'implementazione scelta, come i concetti fondamentali dell'IA, la sua evoluzione nelle componenti che compongono l'architettura del progetto ed un'introduzione al Project-Emerge.

Il capitolo 3 presenta il contributo originale del lavoro di tesi, articolato in quattro sezioni. La sezione di Analisi (Sezione 3.1) descrive il problema affrontato e i requisiti, funzionali e non funzionali, richiesti alla soluzione, accompagnati da una descrizione del dominio applicativo. La sezione di Design (Sezione 3.2) presenta l'architettura della soluzione proposta, illustrando le soluzioni adottate per i principali problemi architetturali riscontrati. La sezione di Sviluppo (Sezione 3.3) approfondisce le scelte implementative più rilevanti, offrendo al lettore alcuni estratti di codice a supporto della discussione. Chiude il capitolo la sezione di Validazione (Sezione 3.4), che riporta uno studio sperimentale dei risultati del benchmark sviluppato per il confronto tra diversi modelli di IA.

Capitolo 2

Stato dell'arte

2.1 Background

2.1.1 Intelligenza Artificiale e Large Language Models

L'IA è una disciplina dell'informatica che si occupa dello sviluppo di sistemi in grado di svolgere attività che, se eseguite da un essere umano, richiederebbero capacità cognitive quali apprendimento, ragionamento, pianificazione e comprensione del linguaggio. Il termine *Artificial Intelligence* fu coniato da John McCarthy nel 1955 nell'ambito della proposta di ricerca che portò alla Dartmouth Conference del 1956, evento generalmente considerato l'atto di nascita dell'Intelligenza Artificiale come disciplina scientifica autonoma [MMRS55, RN21].

Nei decenni successivi, l'evoluzione dell'AI è stata caratterizzata da periodi di forte crescita alternati a fasi di rallentamento, spesso indicate come *AI Winter*. Le limitazioni hardware e la ridotta disponibilità di dati rendevano infatti difficile realizzare sistemi capaci di soddisfare le aspettative iniziali. A partire dagli anni 2000, tuttavia, la crescente disponibilità di risorse computazionali, la diffusione di Internet e l'accumulo di enormi quantità di dati hanno favorito una significativa ripresa della ricerca nel settore.

Uno dei principali sottoinsiemi dell'IA è il *Machine Learning*, un insieme di tecniche che consentono ai sistemi informatici di apprendere automaticamente dai dati senza essere esplicitamente programmati per ogni possibile situazione. Invece di definire manualmente tutte le regole necessarie per risolvere un problema,

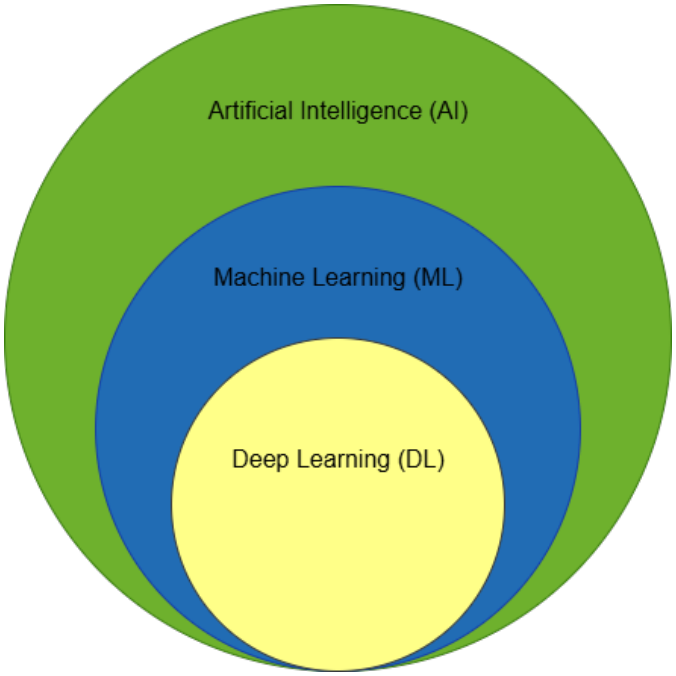


Figura 2.1: Relazione tra AI, Machine Learning e Deep Learning

gli algoritmi di Machine Learning identificano autonomamente schemi e relazioni presenti nei dati di addestramento.

Tra le varie tecniche di Machine Learning, come si può vedere in Figura 2.1, il Deep Learning rappresenta uno degli approcci più rilevanti degli ultimi anni. Esso si basa sull'utilizzo di reti neurali artificiali organizzati in livelli sovrapposti. Ogni livello elabora le informazioni ricevute dal livello precedente, estraendo caratteristiche via via più complesse: i primi livelli riconoscono elementi semplici, come i contorni di un'immagine, mentre i livelli più profondi sono in grado di identificare concetti di alto livello, come il volto di una persona o il significato di una frase. Grazie a questa capacità di apprendere rappresentazioni sempre più astratte, il Deep Learning ha ottenuto risultati particolarmente significativi in diversi ambiti applicativi, tra cui la visione artificiale, il riconoscimento vocale e l'elaborazione del linguaggio naturale.

All'interno di quest'ultimo ambito si collocano i LLM, modelli linguistici di grandi dimensioni addestrati su enormi quantità di testo proveniente da libri, articoli, documentazione tecnica e contenuti presenti sul Web. Lo scopo principale

di tali modelli consiste nel prevedere il token successivo all'interno di una sequenza di testo. Per *token* si intende l'unità minima di testo elaborata dal modello, corrispondente approssimativamente a una parola o a una sua parte. Nonostante questo obiettivo possa apparire relativamente semplice, l'enorme quantità di dati utilizzata durante la fase di addestramento consente ai modelli di acquisire una notevole conoscenza delle strutture linguistiche e delle relazioni semantiche presenti nel linguaggio umano.

La svolta che ha portato allo sviluppo dei moderni LLM è stata l'introduzione dell'architettura Transformer da parte di Vaswani et al. nel 2017. Tale architettura ha introdotto il meccanismo di *self-attention*, che permette al modello di individuare e pesare le relazioni tra elementi diversi di una stessa sequenza di testo. Grazie a questa innovazione è stato possibile addestrare modelli significativamente più grandi e performanti rispetto alle precedenti architetture basate su reti neurali ricorrenti.

L'evoluzione dei Transformer ha portato alla nascita di modelli sempre più avanzati, tra cui Chat-GPT, Gemini, Claude e LLaMA. Questi sistemi sono in grado non solo di comprendere richieste formulate in linguaggio naturale, ma anche di generare testi coerenti, produrre riassunti, tradurre contenuti, rispondere a domande e assistere gli utenti in attività complesse.

Una caratteristica fondamentale dei moderni LLM è la capacità di utilizzare il contesto fornito durante l'interazione per interpretare correttamente le richieste ricevute. Tale contesto viene fornito al modello attraverso il *prompt*, ovvero il testo di ingresso che l'utente costruisce per guidarne il comportamento: può includere istruzioni, esempi, dati di riferimento o semplicemente una domanda. Attraverso opportune tecniche di *prompting*, ovvero la progettazione mirata di questi testi di ingresso, tali modelli possono adattare il proprio comportamento a differenti scenari applicativi senza richiedere un nuovo addestramento.

Nonostante le notevoli capacità dimostrate, gli LLM presentano alcune limitazioni. Tra queste vi è il fenomeno delle cosiddette *hallucinations*, ovvero la generazione di informazioni plausibili ma non necessariamente corrette. Inoltre, le conoscenze del modello sono generalmente limitate ai dati utilizzati durante l'addestramento, rendendo talvolta necessario integrare informazioni provenienti da sorgenti esterne. Per questo motivo, nei sistemi moderni i Large Language Models

vengono frequentemente affiancati da strumenti e componenti software specializzati che consentono di accedere a dati aggiornati e di eseguire operazioni nel mondo reale.

2.1.2 Chatbot e sistemi conversazionali

Un *chatbot* è un sistema software progettato per simulare una conversazione con un essere umano mediante linguaggio naturale. Il termine chatbot deriva dalla combinazione delle parole chat e robot e identifica una categoria di applicazioni il cui obiettivo è consentire agli utenti di interagire con un sistema informatico attraverso modalità comunicative simili a quelle utilizzate nelle conversazioni tra persone.

L'idea di realizzare sistemi capaci di sostenere un dialogo con un essere umano è strettamente legata alla storia dell'IA. Uno dei primi esempi significativi è ELIZA, sviluppato nel 1966 da Joseph Weizenbaum presso il Massachusetts Institute of Technology. ELIZA simulava il comportamento di uno psicoterapeuta attraverso semplici regole di sostituzione e riformulazione del testo inserito dall'utente. Sebbene le sue capacità fossero estremamente limitate, il sistema dimostrò come anche semplici meccanismi conversazionali potessero indurre gli utenti a percepire una forma di comprensione da parte della macchina.

Negli anni successivi furono sviluppati numerosi sistemi conversazionali basati su approcci simili. Tra questi si possono citare PARRY, realizzato negli anni Settanta, e successivamente ALICE, uno dei chatbot più noti degli anni Novanta. Tali sistemi erano generalmente basati su regole predefinite e pattern linguistici esplicitamente programmati dagli sviluppatori.

I chatbot tradizionali possono essere classificati come *rule-based chatbot*. In questa categoria rientrano i sistemi che determinano le proprie risposte attraverso meccanismi statici e predefiniti. Sebbene tali approcci consentano un elevato controllo sul comportamento del sistema, presentano importanti limitazioni in termini di flessibilità. Gli utenti devono infatti formulare richieste compatibili con gli schemi previsti dal progettista, mentre richieste formulate in modo inatteso possono facilmente condurre a errori o risposte non appropriate.

L'evoluzione delle tecniche di elaborazione del linguaggio naturale e l'introduzione del Machine Learning hanno progressivamente ampliato le capacità dei sistemi conversazionali. In particolare, la diffusione degli LLM ha determinato un cambiamento significativo nel modo in cui vengono progettati i chatbot moderni. A differenza dei sistemi basati esclusivamente su regole, i chatbot supportati da LLM sono in grado di interpretare richieste formulate liberamente dagli utenti, comprendere il contesto della conversazione e generare risposte dinamiche adattate alla situazione corrente.

Una componente fondamentale dei moderni sistemi conversazionali è la gestione del contesto. Durante una conversazione, infatti, il significato di un messaggio dipende spesso dalle informazioni scambiate in precedenza. Per questo motivo, i chatbot mantengono generalmente una memoria della conversazione che viene utilizzata per interpretare correttamente i messaggi successivi e garantire una maggiore coerenza delle risposte.

I chatbot trovano oggi applicazione in numerosi settori, tra cui assistenza clienti, supporto tecnico, formazione, commercio elettronico, sanità e automazione industriale. In molti casi essi costituiscono il principale punto di contatto tra utente e sistema software, consentendo di semplificare operazioni che altrimenti richiederebbero interfacce complesse o conoscenze tecniche specifiche.

Tra i principali vantaggi offerti dai sistemi conversazionali vi sono la riduzione della curva di apprendimento, l'accessibilità per utenti non esperti e la possibilità di utilizzare il linguaggio naturale come unico strumento di interazione. Tali caratteristiche risultano particolarmente rilevanti nei contesti in cui gli utenti devono accedere a funzionalità avanzate senza possedere competenze specialistiche relative al dominio applicativo.

Nonostante i notevoli progressi ottenuti negli ultimi anni, i chatbot moderni presentano ancora alcune criticità. La qualità delle risposte dipende fortemente dal modello linguistico utilizzato e dalle informazioni disponibili durante la conversazione. Inoltre, nei sistemi che consentono l'esecuzione di operazioni sul mondo reale è necessario introdurre adeguati meccanismi di controllo e validazione, al fine di evitare che eventuali errori di interpretazione possano causare comportamenti indesiderati.

2.1.3 AI Agents

L'evoluzione dei Large Language Models ha reso possibile la realizzazione di sistemi capaci non solo di generare testo, ma anche di interagire con l'ambiente circostante attraverso strumenti software dedicati. Questa evoluzione ha portato alla diffusione del concetto di *AI Agent*, ovvero un sistema intelligente in grado di percepire informazioni provenienti dall'ambiente, ragionare su di esse e intraprendere azioni finalizzate al raggiungimento di uno specifico obiettivo.

Il concetto di agente intelligente precede di molti anni l'avvento dei moderni modelli linguistici. Nella letteratura dell'Intelligenza Artificiale, un agente viene generalmente definito come un'entità capace di percepire il proprio ambiente attraverso sensori e di agire su di esso mediante opportuni attuatori. Tale definizione comprende una vasta gamma di sistemi, dai robot autonomi ai software in grado di prendere decisioni sulla base delle informazioni disponibili.

Con la diffusione dei Large Language Models, il termine AI Agent ha assunto un significato più specifico. In questo contesto, un agente è generalmente costituito da un modello linguistico che opera come componente decisionale centrale e che può utilizzare strumenti esterni per acquisire informazioni o eseguire operazioni. Il modello non si limita quindi a produrre una risposta testuale, ma è in grado di determinare autonomamente quali azioni siano necessarie per soddisfare una richiesta dell'utente.

Un tipico AI Agent è composto da diverse componenti. La prima è il modello linguistico, responsabile dell'interpretazione delle richieste e della pianificazione delle azioni da intraprendere. A questa si affianca una memoria, utilizzata per conservare informazioni rilevanti riguardanti la conversazione o lo stato corrente del sistema. Il comportamento dell'agente è inoltre condizionato dal suo *System Prompt* ovvero quella parte di prompt che non cambia mai tra una richiesta e l'altra, viene sempre aggiunta come preambolo ai messaggi inviati all'agente e definisce la personalità, le regole ed i compiti di esso. Infine, l'agente dispone di uno o più strumenti esterni, comunemente denominati *tools*, che consentono di accedere a dati aggiornati o di eseguire operazioni al di fuori del modello linguistico. L'architettura è descritta in Figura 2.2

L'utilizzo di strumenti esterni rappresenta uno degli aspetti più significativi

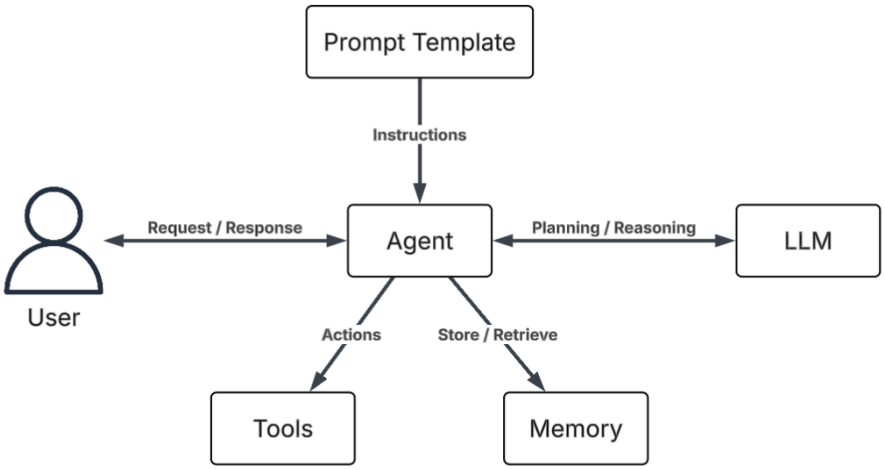


Figura 2.2: Diagramma dell'architettura di un sistema agentico

degli AI Agents moderni. I Large Language Models possiedono infatti una conoscenza limitata alle informazioni apprese durante la fase di addestramento e non hanno la capacità intrinseca di interagire con sistemi esterni. Attraverso l'integrazione di tools dedicati, il modello può invece recuperare dati da database, interrogare servizi web, eseguire calcoli, controllare applicazioni o interagire con dispositivi fisici.

Quando un utente formula una richiesta, l'agente analizza il problema e determina se sia possibile fornire una risposta utilizzando esclusivamente le informazioni già disponibili oppure se sia necessario utilizzare uno o più strumenti esterni. In quest'ultimo caso, il modello genera una richiesta strutturata che identifica il tool da utilizzare e i relativi parametri. Una volta ottenuto il risultato dell'operazione, tale informazione viene reintegrata nel contesto della conversazione e utilizzata per produrre la risposta finale.

Questo paradigma consente di superare una delle principali limitazioni dei Large Language Models tradizionali, trasformando il modello da semplice generatore di testo a componente in grado di partecipare attivamente all'esecuzione di processi complessi. Per tale motivo gli AI Agents vengono oggi impiegati in numerosi ambiti applicativi, tra cui assistenti virtuali avanzati, automazione di processi aziendali, sistemi di supporto alle decisioni e controllo di infrastrutture software.

Negli ultimi anni sono stati proposti numerosi *framework* dedicati allo sviluppo

di AI Agents.[1] Tali strumenti forniscono componenti software che semplificano la gestione della memoria conversazionale, l'integrazione di strumenti esterni e il coordinamento delle operazioni svolte dal modello linguistico.

Un aspetto particolarmente rilevante nella progettazione degli AI Agents riguarda il controllo delle azioni eseguite. Poiché il modello linguistico può commettere errori di interpretazione o generare informazioni non corrette, molte applicazioni introducono meccanismi di validazione e conferma prima dell'esecuzione di operazioni che producono effetti nel mondo reale. Tale approccio consente di mantenere l'utente all'interno del processo decisionale e di ridurre il rischio di comportamenti indesiderati.

Nel contesto del presente lavoro, il chatbot sviluppato può essere interpretato come un AI Agent specializzato nell'interazione con uno sciame di droni. Il modello linguistico non si limita infatti a fornire informazioni sulle formazioni disponibili, ma utilizza strumenti dedicati per ottenere la configurazione delle formazioni supportate, validare i parametri inseriti dall'utente e generare le istruzioni necessarie al controllo dello sciame. L'agente agisce quindi come intermediario tra il linguaggio naturale utilizzato dall'utente e le operazioni effettivamente supportate dal sistema di controllo dei droni, consentendo un'interazione più intuitiva e accessibile.

2.1.4 Project-Emerge e sciame di droni

La coordinazione di grandi gruppi di robot rappresenta una delle principali sfide della robotica distribuita e della *swarm robotics*. All'aumentare del numero di unità coinvolte, infatti, diventa sempre più complesso definire il comportamento di ciascun robot individualmente e garantire che l'insieme del sistema produca il comportamento collettivo desiderato. Per affrontare tale problema sono stati sviluppati approcci di macro-programmazione, nei quali il programmatore descrive il comportamento del gruppo nel suo complesso anziché specificare direttamente le azioni dei singoli robot.

[1] Alcuni dei framework più utilizzati ed open-source sono LangChain, Microsoft AutoGen, LlamaIndex e Semantic Kernel. Esistono anche approcci visivi che non richiedono scrittura di codice ma lo fanno tramite approcci di drag-and-drop, quali Botpress e Dify

Uno dei paradigmi più rilevanti in questo contesto è l'*Aggregate Computing* [BPV15, Vir21]. Tale approccio consente di considerare un insieme di dispositivi distribuiti come una singola entità computazionale, permettendo di descrivere il comportamento globale del sistema mediante astrazioni di alto livello. Invece di programmare separatamente ciascun robot, lo sviluppatore definisce il comportamento collettivo desiderato, lasciando al sistema il compito di determinare le azioni locali necessarie affinché emerga il comportamento globale previsto.

Il concetto fondamentale alla base dell'Aggregate Computing è quello del *Computational Field* (campo computazionale). Un campo computazionale può essere interpretato come una struttura dati distribuita che associa ad ogni dispositivo del sistema un determinato valore e che evolve nel tempo in funzione dello stato dell'ambiente e delle interazioni tra dispositivi. Formalmente, un campo rappresenta una mappatura tra i nodi della rete e i valori computati localmente da ciascun nodo. L'insieme di tali valori costituisce una rappresentazione globale dello stato del sistema distribuito.

L'utilizzo dei campi computazionali consente di ragionare a livello collettivo anziché individuale. In modo analogo, è possibile costruire campi che descrivono direzioni di movimento, vincoli geometrici o relazioni di vicinato tra i membri dello sciame. Combinando opportunamente tali campi è possibile ottenere comportamenti collettivi complessi mantenendo una descrizione del sistema relativamente compatta e indipendente dal numero di robot coinvolti.

L'esecuzione di un programma aggregato si basa sul paradigma *Sense-Compute-Actuate*, come si vede nella Figura 2.3. Durante la fase di *Sense*, ciascun dispositivo raccoglie informazioni dall'ambiente circostante e dai dispositivi vicini. Nella fase di *Compute*, il programma aggregato viene valutato utilizzando le informazioni disponibili, producendo nuovi valori di campo e nuove azioni da eseguire. Infine, nella fase di *Actuate*, tali azioni vengono applicate all'ambiente fisico. La ripetizione continua di questo ciclo consente al sistema di adattarsi dinamicamente ai cambiamenti dell'ambiente e alle perturbazioni esterne, favorendo fenomeni di auto-organizzazione e auto-riparazione.

Nel contesto della swarm robotics, uno degli esempi più significativi di comportamento collettivo è rappresentato dalle *formazioni*. Una formazione può essere definita come una particolare configurazione geometrica che lo sciame deve assu-

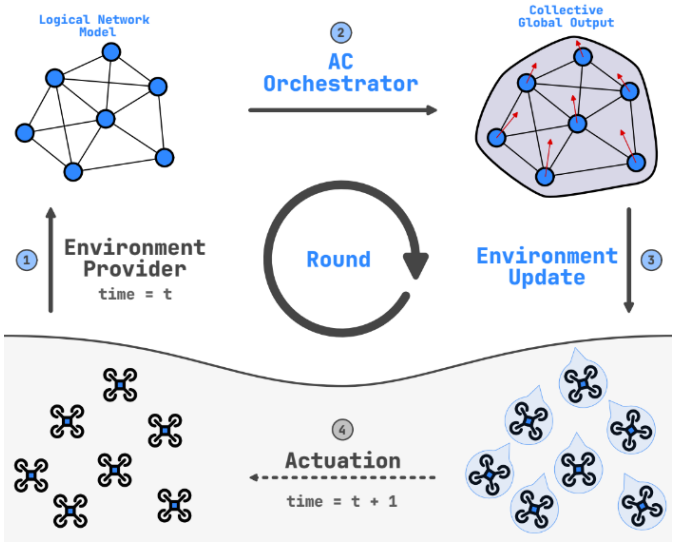
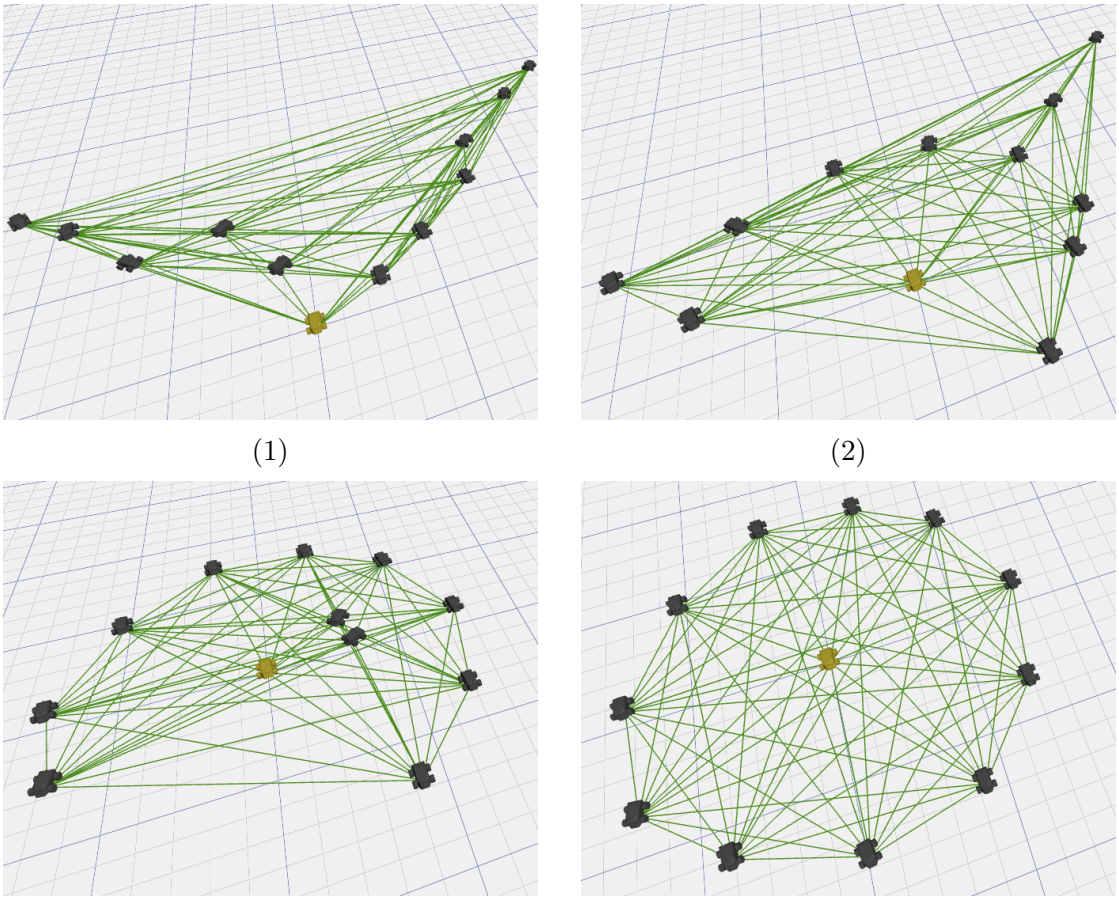


Figura 2.3: Rappresentazione del paradigma Sense-Compute-Actuate riportato nel contesto di Project-Emerge [FAB+26]

mere nello spazio. A differenza di una semplice disposizione statica dei robot, una formazione viene generalmente ottenuta attraverso regole locali che guidano ciascun robot verso una configurazione globale desiderata. Il vantaggio di questo approccio è che il sistema può reagire autonomamente a perturbazioni, guasti o modifiche dell'ambiente, riconfigurandosi fino a recuperare la struttura prevista.

Le formazioni possono assumere differenti forme a seconda dell'obiettivo applicativo. Nel caso di Project-Emerge sono state implementate diverse configurazioni spaziali, tra cui formazioni lineari, quadrate, circolari e a V, riportate in Figura 2.5, oltre a comportamenti di coordinazione basati sulla presenza di un robot leader. Ogni formazione è definita da uno specifico programma aggregato e può essere parametrizzata modificando opportuni valori che influenzano il comportamento dello sciame. Si può osservare, attraverso il programma di simulazione del Project-Emerge, lo spostamento dei droni dalla formazione a V a quella a cerchio in Figura 2.4.

Project-Emerge nasce con l'obiettivo di rendere concretamente applicabili i principi dell'Aggregate Computing su sciami robotici reali. Sebbene negli anni siano stati sviluppati numerosi framework teorici e strumenti di simulazione per la programmazione aggregata, il loro impiego in contesti fisici reali è stato spesso limi-



(1) (2)

(3) (4)

Figura 2.4: Spostamento dei droni dopo l'invio del comando

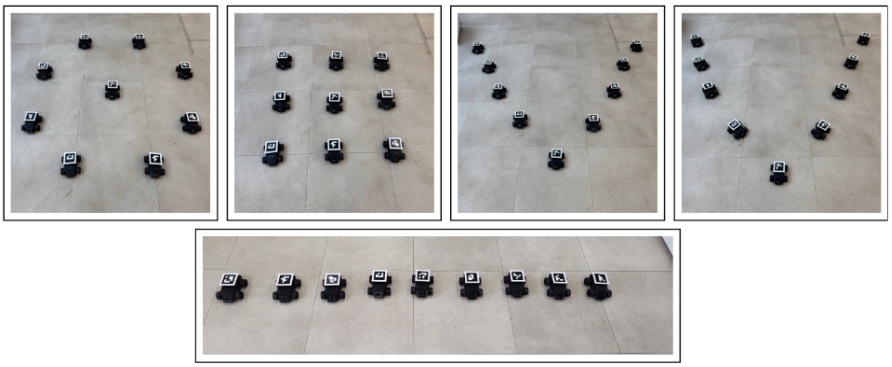


Figura 2.5: Formazioni disponibili nel Project-Emerge [FAB+26]

tato dalla complessità di integrazione con sistemi di localizzazione, infrastrutture di comunicazione e piattaforme robotiche eterogenee. Project-Emerge si propone di colmare tale divario fornendo una piattaforma completa per l'esecuzione di programmi aggregati sia in simulazione sia su robot fisici.

L'architettura del sistema è organizzata in componenti modulari responsabili della localizzazione dei robot, della gestione delle relazioni di vicinato, dell'esecuzione dei programmi aggregati, della comunicazione e dell'interazione con l'utente. Nella configurazione attuale il runtime aggregato viene eseguito su un server centrale che mantiene una rappresentazione logica dello sciame, calcola i campi necessari all'esecuzione del programma e genera i comandi da inviare ai robot. Le diverse componenti comunicano tra loro attraverso il protocollo Message Queuing Telemetry Transport (MQTT), mentre la posizione e l'orientamento dei robot vengono ottenuti mediante un sistema di visione basato su marker ArUco.

Il protocollo MQTT è un protocollo di comunicazione leggero progettato per lo scambio di messaggi tra dispositivi distribuiti. Sviluppato alla fine degli anni Novanta nasce con l'obiettivo di consentire comunicazioni affidabili anche in presenza di reti caratterizzate da elevata latenza, banda limitata o connessioni instabili. MQTT è oggi uno dei protocolli più diffusi nel settore dell'Internet of Thing (IoT), dove numerosi dispositivi devono comunicare tra loro in modo efficiente senza richiedere elevate risorse computazionali o di rete.

MQTT adotta il paradigma Publish/Subscribe, quindi i dispositivi non comunicano direttamente tra loro, ma utilizzano un componente centrale denominato *broker*. I dispositivi che producono informazioni pubblicano messaggi sul broker, mentre quelli interessati a riceverli si sottoscrivono ai relativi canali di comunicazione.

I principali elementi che compongono un sistema MQTT sono:

- **Broker**: componente centrale responsabile della ricezione, gestione e distribuzione dei messaggi.
- **Publisher**: entità che pubblica messaggi sul broker.
- **Subscriber**: entità che riceve messaggi sottoscrivendosi a specifici canali.
- **Topic**: identificatore logico utilizzato per organizzare e instradare i messaggi.

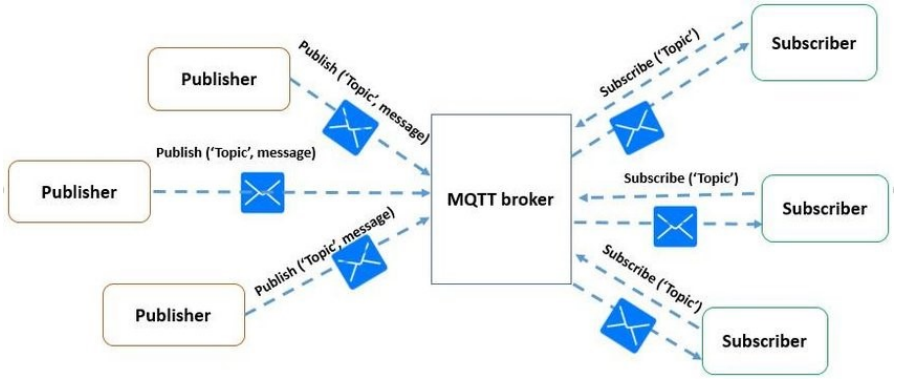


Figura 2.6: Architettura del protocollo MQTT [SEN20]

Quando un publisher invia un messaggio, esso viene associato a uno specifico topic. Il broker provvede successivamente a inoltrare il messaggio a tutti i subscriber che risultano iscritti a quel topic. Tale meccanismo consente di disaccoppiare produttori e consumatori di informazioni, poiché nessuna delle due parti necessita di conoscere direttamente l'altra, come mostrato in Figura 2.6.

Un ulteriore vantaggio di MQTT è rappresentato dalla sua scalabilità. L'introduzione del broker consente infatti di gestire un elevato numero di dispositivi mantenendo una struttura di comunicazione semplice e facilmente estendibile. Risulta perciò particolarmente vantaggioso disporre di un protocollo leggero, indipendente dalla piattaforma utilizzata e capace di gestire efficacemente comunicazioni asincrone tra componenti distribuite.

Un aspetto particolarmente rilevante di Project-Emerge è la sua capacità di preservare le proprietà di auto-organizzazione tipiche dell'Aggregate Computing anche in presenza di perturbazioni fisiche. Gli esperimenti descritti nel lavoro originale, [FAB+26], mostrano come lo sciame sia in grado di recuperare autonomamente una formazione desiderata dopo lo spostamento manuale di uno o più robot, la modifica dei parametri operativi o la sostituzione del robot leader. Questo comportamento emerge naturalmente dalla continua rivalutazione dei campi computazionali e costituisce una delle principali caratteristiche distintive dell'approccio aggregato.

Nel presente lavoro di tesi, Project-Emerge rappresenta il sistema di controllo sottostante con cui il chatbot interagisce. Le formazioni messe a disposizione

dell'utente corrispondono infatti a programmi aggregati già disponibili all'interno della piattaforma. Il chatbot non si occupa dell'esecuzione delle logiche di coordinazione, ma fornisce un'interfaccia conversazionale che consente agli utenti di selezionare e configurare tali programmi mediante linguaggio naturale, semplificando significativamente l'accesso alle funzionalità offerte dal sistema.

Capitolo 3

Contributo

3.1 Analisi

3.1.1 Descrizione e requisiti

Descrizione del sistema

Il progetto di Tesi mira alla realizzazione di un chatbot dotato di un'interfaccia semplice e intuitiva, progettata per consentire anche ad utenti privi di competenze tecniche specifiche di interagire con uno sciame di droni mediante linguaggio naturale. La soluzione proposta si pone l'obiettivo di ridurre al minimo le conoscenze richieste per l'utilizzo del sistema sviluppato nell'ambito del Project-Emerge, eliminando la necessità di apprendere procedure operative complesse o di conoscere nel dettaglio le modalità di configurazione dello sciame.

Il chatbot è stato concepito come un intermediario tra l'utente e il sistema di controllo dei droni. Attraverso una conversazione in linguaggio naturale, l'utente può ottenere informazioni sulle formazioni supportate, comprenderne le caratteristiche e ricevere assistenza nella scelta della configurazione più adatta alle proprie esigenze operative. Una volta individuata la formazione desiderata, il sistema guida l'utente nella definizione dei parametri necessari al suo utilizzo, verificando la correttezza e la completezza delle informazioni fornite.

Le formazioni disponibili non vengono generate dinamicamente, ma appartengono a un insieme predefinito di comportamenti eseguibili dallo sciame. Il compito del chatbot consiste pertanto nell'assistere l'utente durante il processo di selezione

e configurazione di tali comportamenti, traducendo infine le richieste formulate in linguaggio naturale in istruzioni interpretabili dal sistema di controllo. Al fine di garantire la sicurezza operativa, l'invio di una nuova configurazione richiede sempre una conferma esplicita da parte dell'utente prima dell'esecuzione.

Requisiti funzionali

F1 Il sistema deve consentire l'interazione con l'utente tramite un'interfaccia conversazionale basata su chat.

F2 Il sistema deve mantenere lo storico della conversazione al fine di preservare il contesto dell'interazione e consentire una comunicazione coerente con i messaggi precedenti.

F3 Per rendere le risposte coerenti con la conversazione attuale il sistema deve generare le proprie risposte considerando sia il messaggio corrente sia il contesto conversazionale accumulato.

F4 Il sistema deve ricevere in input, per ogni sessione conversazionale, l'insieme delle formazioni supportate, così da garantire al chatbot una conoscenza aggiornata delle capacità dello sciame di droni.

F5 Il sistema deve rappresentare ogni formazione mediante nome, descrizione e insieme dei parametri configurabili. Ogni parametro deve essere caratterizzato da nome, descrizione e tipologia.

F6 Il sistema deve supportare comandi per ottenere la lista delle formazioni disponibili e, per ciascuna formazione, i parametri configurabili, in modo tale da dare all'utente accesso completo alle risorse messe a disposizione dal sistema.

F7 Il sistema deve tradurre una richiesta di formazione dell'utente in un messaggio MQTT interpretabile da Project-Emerge, così da richiedere il cambio di formazione.

F8 Per non dover chiudere e riaprire il programma in caso di problemi con lo storico del chatbot il sistema deve consentire il reset della conversazione, eliminando lo storico dei messaggi e la memoria conversazionale associata.

F9 Il sistema deve richiedere una conferma esplicita dell'utente prima dell'invio di una nuova configurazione allo sciame di droni.

F10 Il sistema deve verificare la correttezza e la completezza dei parametri richiesti dalla formazione selezionata prima di procedere all'invio della configurazione.

Requisiti non funzionali

NF1 L'interfaccia utente deve risultare semplice e intuitiva, adottando paradigmi di interazione familiari anche a utenti privi di competenze tecniche specifiche.

NF2 L'interfaccia deve distinguere chiaramente i messaggi dell'utente da quelli generati dal chatbot.

NF3 Il sistema deve essere agnostico rispetto alla specifica formazione, funzionando a partire dalla configurazione ricevuta in input.

NF4 L'architettura software deve garantire modularità e separazione delle responsabilità tra il provider delle formazioni, il modulo di comunicazione e l'agente conversazionale.

NF5 Nello sviluppo di agenti è fondamentale se si usano provider in cloud, come OpenAi e Gemini, avere la possibilità di modificare velocemente modello di LLM che si adopera per le richieste API, perciò andrà predisposta una metodologia efficace per modificare il modello chiamato con pochi cambiamenti nel codice.

NF6 Il sistema deve prevenire l'esecuzione accidentale di comandi mediante meccanismi di conferma esplicita da parte dell'utente.

NF7 Il sistema deve essere facilmente aggiornabile e modificabile per sviluppi futuri e cambiamenti nell'operatività della soluzione.

Scenario di interazione utente-chatbot

All'avvio dell'applicazione, l'utente viene presentato a un'interfaccia conversazionale ispirata alle comuni applicazioni di messaggistica, come mostrato in Figura 3.1. Tale scelta progettuale mira a sfruttare paradigmi di interazione già familiari alla maggior parte degli utenti, riducendo la barriera di ingresso e favorendo

un utilizzo immediato del sistema. Il chatbot permette di scegliere una formazione e configurarne i parametri operativi. Tali formazioni sono predefinite siccome fanno parte di un set di programmi eseguibili dai droni, e perciò pre-programmati. L'interazione classica tra utente inesperto e chatbot è composta da una prima fase di ambientazione ed esplorazione del sistema da parte dell'utente. In questa fase può chiarire i propri dubbi riguardanti il funzionamento o lo scopo del chatbot, l'intento è che l'utente non necessiti nessun manuale per iniziare ad interagire con i droni. Superata questa prima fase l'utente dovrebbe aver chiarito il concetto di formazione e le possibilità messe a disposizione dal programma. Inizia perciò la fase vera e propria di decisione della formazione, in cui l'utente fa domande ed espone i propri bisogni funzionali ed il chatbot risponde in modo esaustivo e guida l'utente alla formazione più adeguata alle sue esigenze. L'interazione prosegue con la fase conclusiva, decisi formazione e valori dei parametri necessari, il chatbot effettua validazione e invio della formazione ai droni, sotto esplicita conferma dell'utente. Questo requisito è posto come fondamentale al chatbot, in modo da evitare l'invio di comandi non confermati dall'utente causando malfunzionamenti. Dopodiché l'utente osserva nel mondo reale i cambiamenti richiesti e se lo desidera può continuare l'utilizzo del chatbot ripartendo dalla seconda fase. Una rappresentazione schematica di tale interazione è presente in Figura 3.2.

3.1.2 Modello del Dominio

Il modello del dominio individua le principali entità coinvolte nell'interazione tra *utente* e *sciame di droni*. L'utente interagisce con il *chatbot* mediante l'invio di *messaggi* in linguaggio naturale, ai quali il sistema fornisce una risposta coerente. La sequenza dei messaggi scambiati costituisce una *conversazione*, che viene mantenuta dal sistema e utilizzata come contesto per la generazione delle risposte successive. Durante l'interazione l'utente sarà in grado di selezionare una tra le *formazioni* disponibili chiedendo di creare un'istanza di quest'ultima decidendo anche il valore dei *parametri* necessari. La configurazione risultante viene successivamente sottoposta a validazione da parte del chatbot. Il chatbot è in grado di eseguire le azioni richieste dall'utente attraverso i *Tool*. Se la validazione va a buon fine, l'utente dovrà confermare esplicitamente il desiderio di inviare tale

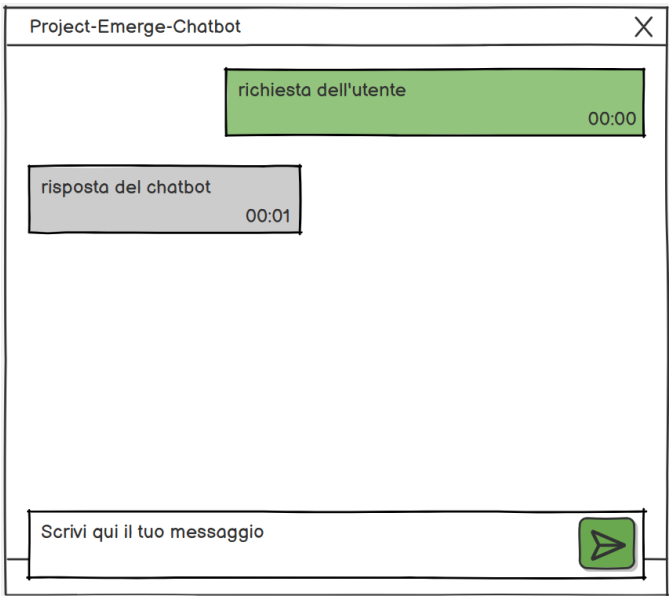


Figura 3.1: Mockup dell'interfaccia utente ispirata alle applicazioni di messaggistica

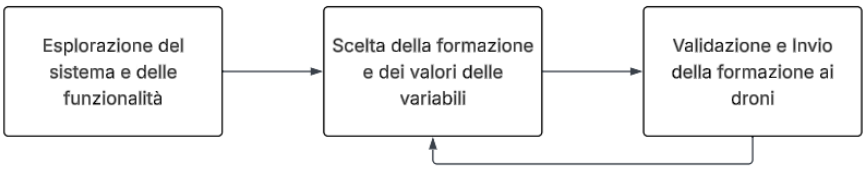


Figura 3.2: Diagramma rappresentante l'interazione classica tra user e chatbot

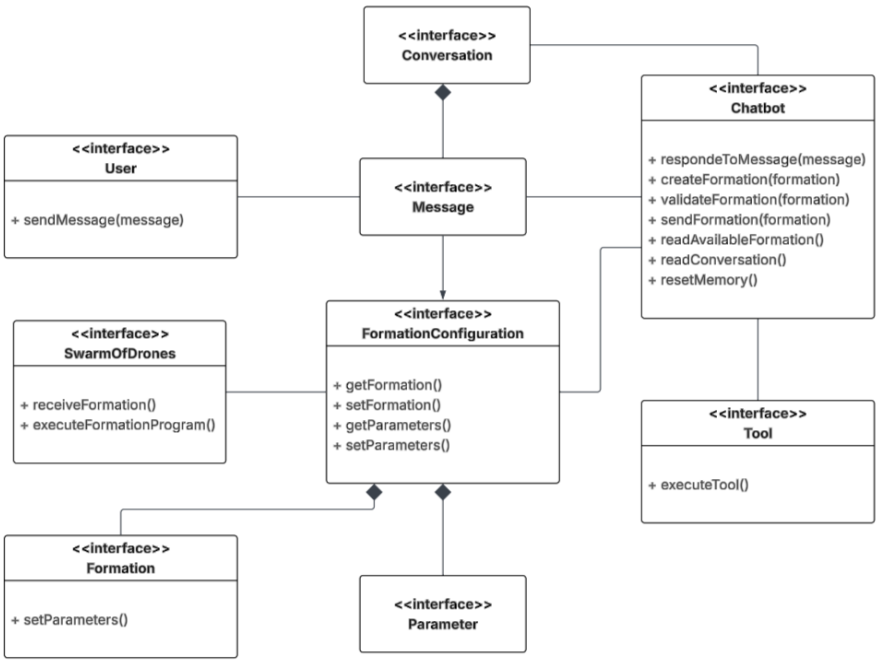


Figura 3.3: Diagramma del modello di dominio del sistema

formazione allo sciame di droni. Quest'ultimo, una volta ricevuta la configurazione, procede all'esecuzione della formazione selezionata utilizzando i parametri specificati dall'utente. Gli elementi costitutivi il problema sono sintetizzati in Figura 3.3.

3.2 Design

3.2.1 Architettura

L'architettura della soluzione segue il pattern MVC. La View compone tutto ciò che è visibile all'utente. Il Model si compone di diversi moduli funzionali che gestiscono la logica del sistema. L'entry point dal controller al model è proprio l'interfaccia *Model* che si occupa di wrappare la creazione di tutti i moduli e di esporre l'accesso all'agente. L'entry point della view è l'interfaccia *ChatView* che viene implementata dalla classe *ChatWindow* in modo da esporre al controller i metodi utili, mantenendone però sconosciuta l'implementazione. Ciò permette di

raggiungere un livello di incapsulamento adatto a una possibile sostituzione in blocco della view.

I moduli del model comprendono l'agente (*Agent*), il fornitore delle formazioni (*FormationProvider*) disponibili nel sistema, il componente che si occupa della comunicazione con i droni (*Sender*), e il modulo addetto alla gestione dei tools (*ToolsHandler*) che vengono offerti all'agente per un corretto funzionamento. Ognuna di queste componenti è modulare perciò può essere sostituita in modo semplice e senza dover apportare modifiche sul resto del codice. Entrando più nello specifico di questi elementi del model l'Agent rappresenta il cervello del chatbot, riceve messaggi ed elabora risposte che possono essere semplici o composte da diversi meta-dati. Il ToolsHandler viene fornito all'agente per dargli la possibilità di effettuare le azioni di cui è incaricato, tra le quali: leggere le formazioni disponibili, validare le configurazioni concordate con l'utente, inviare le istanze delle formazioni allo sciame di droni, e resettare la propria memoria conversazionale. Il set di tool è facilmente espandibile. Il Sender è incaricato di creare la vera e propria connessione tra il chatbot e lo sciame, quindi si connette allo sciame, invia messaggi e può disconnettersi se necessario. Viene utilizzato all'interno del ToolsHandler per rendere operativo il tool di invio. Il FormationProvider deve fornire all'agente l'elenco delle formazioni disponibili, con i relativi parametri operativi, all'agente. Tale elenco è fornito dal sistema che controlla i droni, poiché si tratta di un insieme di programmi predefiniti, perciò questo modulo riceve l'elenco e lo mette a disposizione dell'agente. Viene utilizzato anch'esso dal ToolsHandler per implementare il rispettivo tool di richiesta di informazioni. Tale architettura è schematizzata in Figura 3.4.

3.2.2 Design dettagiato

Problema: La capacità del sistema di svolgere il proprio compito è strettamente correlata alla potenza del modello di intelligenza artificiale scelto (*ChatModel*). Non sono rari i casi in cui è necessario cambiare modello, ad esempio durante una fase di validazione e benchmark delle capacità dei vari modelli (Sezione 3.4), come richiesto anche dal requisito **NF5**. Tuttavia, ciascun provider di IA richiede una procedura di creazione differente, con parametri di configurazione propri (chiave

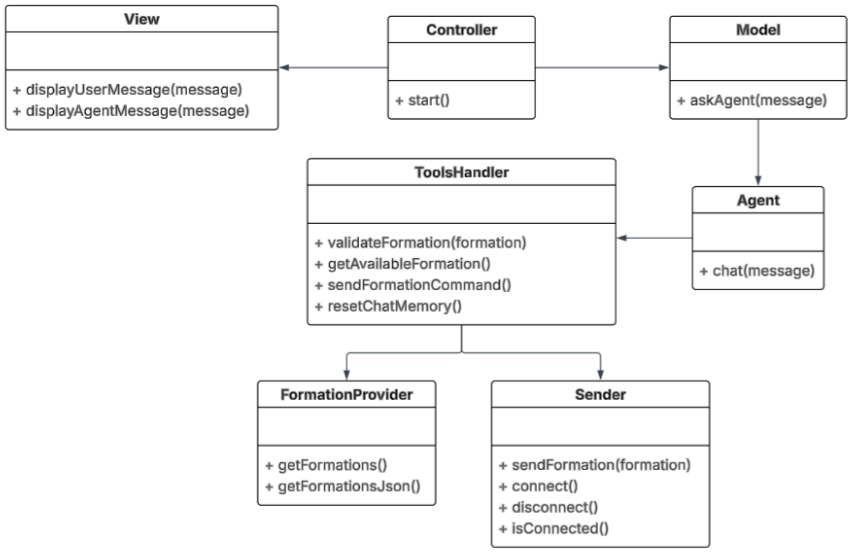


Figura 3.4: Architettura generale del sistema

API, URL del servizio, timeout, ecc.); se gestita direttamente dall'*Agent*, questa logica lo accoppierebbe inutilmente ai dettagli implementativi di ciascun provider.

Soluzione: Il modulo per la gestione del modello di IA utilizza il pattern *Simple Factory*[1], incarnato dalla classe *ChatModelFactory*. Questa nasconde all'*Agent* i dettagli di creazione di ciascun *ChatModel*, esponendo inizialmente un metodo dedicato per ciascun provider (Figura 3.5).

Problema: La soluzione precedente, pur disaccoppiando l'*Agent* dai dettagli di creazione, mantiene un metodo distinto per ciascun provider (`createGeminiChatModel`, `createOpenaiChatModel`, `createOllamaChatModel`). Aggiungere un nuovo provider richiederebbe quindi sia una nuova classe di metodi nella *Factory*, sia modifiche nel codice chiamante per selezionare il metodo corretto, tipicamente tramite confronti su stringhe poco robusti. Anche questo problema ci allontana dalla richiesta del requisito **NF5**.

[1]Non incluso tra i pattern GoF classici, ma ampiamente adottato in letteratura per centralizzare la logica di creazione di oggetti senza richiedere una gerarchia di factory polimorfiche.

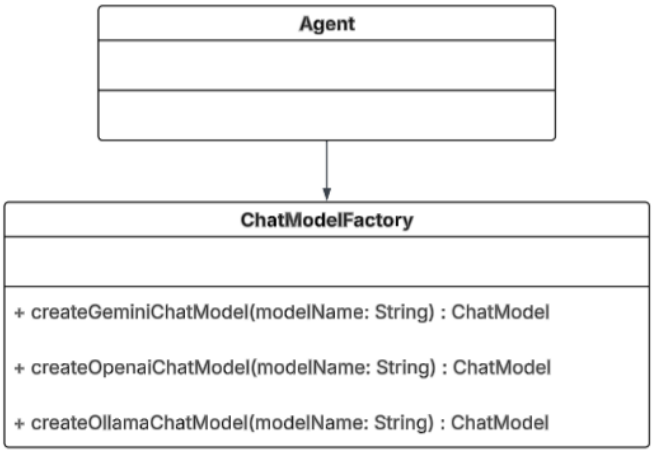


Figura 3.5: Diagramma della Factory di ChatModel

Soluzione: Per risolvere questo problema viene introdotta l'enumerazione *ModelProvider*, che associa a ciascun modello disponibile il proprio nome specifico e il relativo *ProviderType* (es. `OPENAI`, `GEMINI`, `OLLAMA`). Questa associazione permette alla *Factory* di esporre un singolo metodo `createChatModel` (Figura 3.6), che seleziona internamente la procedura di creazione corretta tramite uno switch sul tipo di provider, type-safe in fase di compilazione e privo dei rischi tipici del parsing di stringhe. L'aggiunta di un nuovo modello richiede così la sola estensione dell'enumerazione, senza alcuna modifica al codice chiamante.

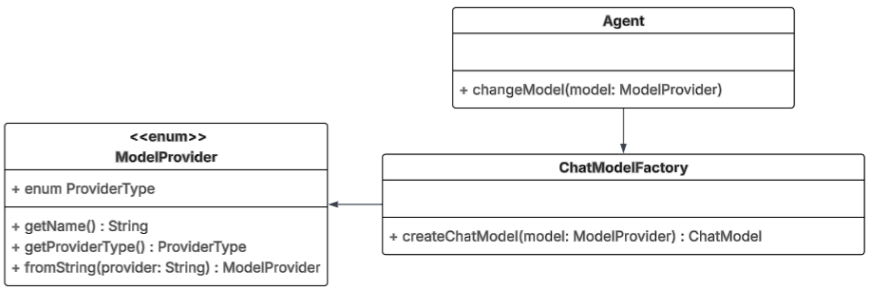


Figura 3.6: Diagramma dell'enum per gestire i diversi provider

Problema: Nel set di *Tools* forniti all'agente è presente anche *resetMemory*, che permette di azzerare la memoria conversazionale dell'agente per ripartire da una conversazione pulita. L'azione di reset, tuttavia, deve essere eseguita dall'*Agent* stesso, poiché è lui a possedere e gestire la propria *ChatMemory*. Se il *ToolsHandler* mantenesse un riferimento diretto all'*Agent* per invocarne il reset, si creerebbe una dipendenza circolare: l'*Agent* possiede il *ToolsHandler*, che a sua volta dovrebbe possedere un riferimento all'*Agent*. Questa architettura viola il principio di modularità richiesto dal requisito **NF4**

Soluzione: Per evitare tale dipendenza circolare, il sistema adotta un meccanismo di *callback*: il *ToolsHandler* non conosce l'*Agent*, ma espone un metodo `setOnResetMemoryCallback` che accetta un *Runnable*, invocato internamente dal tool `resetChatMemory` quando l'azione viene richiesta. È l'*Agent*, in fase di costruzione, a registrare come callback il proprio metodo di reset, mantenendo così il controllo sulla propria memoria senza essere conosciuto direttamente dal *ToolsHandler*. Questa tecnica, riconducibile al pattern *Observer* nella sua forma più elementare (un solo osservatore), introduce nel *ToolsHandler* uno stato mutabile che richiede un controllo di nullità prima di ogni invocazione. La callback è assente finché non viene esplicitamente impostata. La soluzione è rappresentata in Figura 3.7.

Problema: Durante lo sviluppo del sistema sono state adottate delle soluzioni operative specifiche, come l'utilizzo del protocollo MQTT per l'invio dei messaggi, o la lettura da file delle formazioni. Ma prevedendo futuri sviluppi del sistema, con possibili cambiamenti nell'operatività stessa dei moduli, come richiesto dal requisito **NF7**, un approccio standard e statico rende tali cambiamenti molto impegnativi.

Soluzione: Il sistema utilizza il pattern *Strategy* sui moduli *Sender* e *FormationProvider* esponendo un'interfaccia completamente distaccata dal funzionamento di basso livello delle soluzioni adoperate, come mostrato in Figura 3.8. Tale soluzione rende intercambiabili i moduli nascondendone il funzionamento. Un esempio è l'utilizzo, nel modulo di validazione dei modelli, di due implementazioni differenti

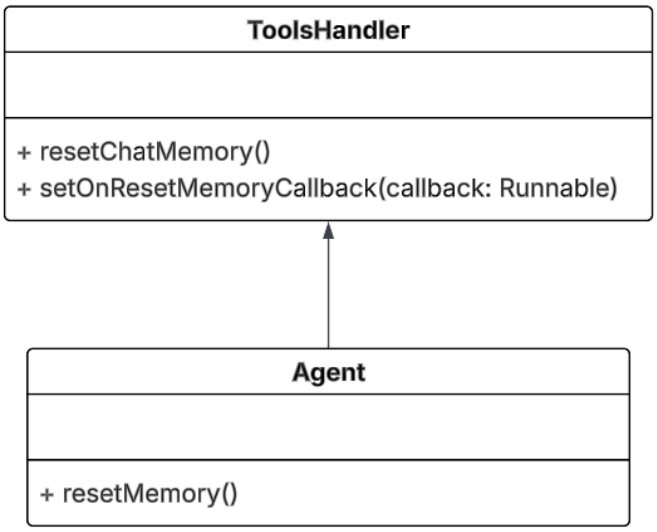


Figura 3.7: Diagramma della callback tra Agent e ToolsHandler

rispetto al modulo operativo normale, al posto del MqttSender e FormationProviderImpl vengono utilizzati *StubSender* e *StubFormationProvider*. Questo è possibile anche grazie alla *Dependency Injection* che viene fatta nel costruttore del ToolsHandler, ovvero i moduli non vengono istanziati da quest'ultimo, ma gli vengono passati attraverso il costruttore. Questo permette la vera intercambiabilità senza dover effettivamente modificare l'Agent.

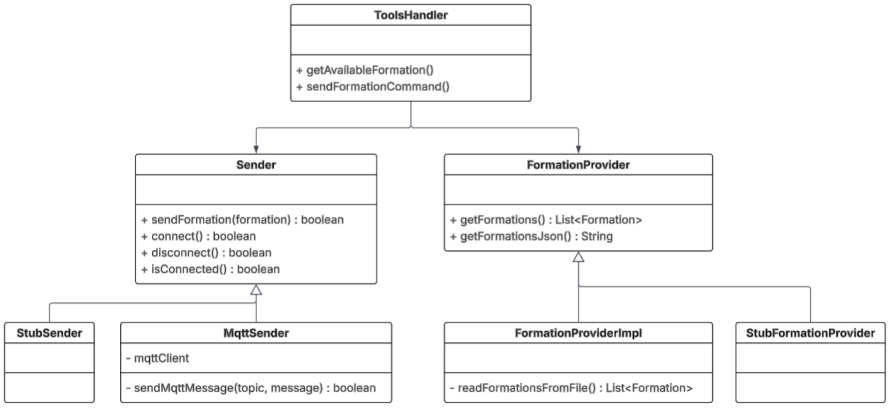


Figura 3.8: Diagramma del pattern Strategy per Sender e FormationProvider

3.3 Sviluppo

Il codice sorgente del progetto è disponibile pubblicamente al seguente indirizzo: `https://github.com/FedeMariani18/Project-Emerge-chatbot.git` ([Mar25]).

3.3.1 Tecnologie utilizzate

Java e LangChain4j: Il sistema è stato implementato interamente in Java. La scelta è ricaduta su questo linguaggio per la maturità del suo ecosistema, la solidità delle librerie disponibili e la piena compatibilità con *LangChain4j*, il framework utilizzato per l'integrazione dei modelli linguistici. Rispetto ad altri linguaggi comuni in ambito intelligenza artificiale, come Python, Java offre un sistema di tipizzazione statica più rigoroso, che si è rivelato particolarmente utile nella definizione delle interfacce tra i moduli del sistema e nella gestione type-safe dei provider (Sezione 3.2.2).

LangChain4j è la versione Java di LangChain, framework open-source originariamente sviluppato per Python. Pur non essendo architetturalmente identica alla versione originale, ne persegue gli stessi obiettivi e ne offre funzionalità analoghe. Il framework è altamente specializzato nell'integrazione di LLM nelle applicazioni Java e fornisce un'API unificata per interagire con diversi provider di modelli (tra cui OpenAI, Gemini e Ollama), evitando di dover implementare chiamate specifiche per ciascun servizio. Tra le funzionalità offerte rilevanti per questo progetto si annoverano: la gestione della memoria conversazionale, il meccanismo di *tool calling* tramite annotazioni, la costruzione di agenti AI tramite `AiServices`, e il supporto ai *prompt template*. La disponibilità di queste astrazioni ha permesso di concentrare lo sviluppo sulla logica applicativa del sistema, delegando al framework la complessità dell'integrazione con i modelli linguistici.

Protocollo MQTT: La scelta del protocollo MQTT come canale di comunicazione con lo sciame di droni è stata determinata dalla sua adozione all'interno di Project Emerge, con il quale il sistema sviluppato deve integrarsi. Project Emerge ha adottato MQTT per la sua particolare adeguatezza nei sistemi IoT e robotici: il modello publish/subscribe su cui si basa permette di realizzare architetture disaccoppiate ed efficienti, in cui tutti i componenti comunicano attraverso un broker

centrale anziché instaurare connessioni dirette tra loro. In un contesto come quello degli sciami di droni, questo approccio evita la proliferazione di canali di comunicazione punto-punto, consentendo a ciascun componente di pubblicare e sottoscrivere messaggi senza dover conoscere gli altri dispositivi presenti nel sistema.

Sul piano implementativo, l'impegno richiesto è risultato contenuto: il broker MQTT era già operativo all'interno di Project Emerge, e il sistema sviluppato ha richiesto unicamente la realizzazione di un client in grado di connettersi al broker e pubblicare i comandi di formazione sul topic appropriato. Tale client è incapsulato nella classe *MqttSender*, che implementa l'interfaccia *Sender* descritta nel capitolo di Design (Sezione 3.2.2).

Interfaccia grafica (Swing): L'interfaccia grafica è stata realizzata mediante la libreria *Swing*, inclusa nel *Java Development Kit* e quindi priva di dipendenze esterne aggiuntive. La semplicità di integrazione con il resto del codice Java e la disponibilità di componenti predefiniti (pannelli, campi di testo, pulsanti, gestori di layout) hanno permesso di realizzare un'interfaccia funzionale senza distogliere l'attenzione dalla logica del sistema, che rappresentava la priorità dello sviluppo. Dato che l'interfaccia grafica è completamente disaccoppiata dalla logica tramite il pattern MVC, una sua eventuale sostituzione, ad esempio con un'interfaccia web o vocale, non richiederebbe alcuna modifica al resto del sistema.

3.3.2 Implementazione del Model

Integrazione dei tool con LangChain4j: LangChain4j fornisce un meccanismo nativo per la registrazione dei tools, fornendo la possibilità di definire una classe dedicata da registrare interamente presso l'agente in fase di costruzione. Questo metodo centralizza tutti gli strumenti che vanno ad interagire con il mondo esterno quindi ad implementare i tools in una sola classe. La classe in questione è il *ToolsHandler*, la definizione di un tool avviene attraverso l'annotazione **@Tool(descrizione: String)** da apportare su un metodo, come si vede nel Listato 3.1, in questo modo esso diventa uno strumento utilizzabile dall'agente. La descrizione associata all'annotazione costituisce il principale riferimento semantico utilizzato dal modello linguistico per decidere quando invocare il tool. Quindi

Listato 3.1: Classe ToolsHandler, contenente la definizione dei tools

```
public class ToolsHandler {
    private FormationProvider formationProvider;
    private Sender sender;

    public ToolsHandler(FormationProvider formationProvider, Sender sender) {
        this.formationProvider = formationProvider;
        this.sender = sender;
    }

    @Tool("Ottieni una stringa in formato json con tutte le formazioni disponibili
         per i droni")
    public String getAvailableFormations(){
        return formationProvider.getFormationsJson();
    }
}
```

viene spiegato brevemente cosa fa al suo interno, quali sono i parametri in input e quali i risultati in output, in modo da dare una spiegazione semanticamente esaustiva.

Nel Listato 3.1 possiamo inoltre notare l'implementazione fondamentale per l'utilizzo del pattern *Strategy*, infatti si vede come il *FormationProvider* ed il *Sender* vengono iniettati all'interno del costruttore, inoltre notiamo che getAvailableFormations() utilizza solo il metodo dell'interfaccia con quel compito specifico. Questo meccanismo consiglia il possibile utilizzo di un qualsiasi FormationProvider o Sender che venga passato nel costruttore, senza nessuna necessità di modificare il ToolsHandler.

Gestione dell'Agent: La classe *Agent*, che è il cuore del sistema, è fornita di alcuni meccanismi per renderla più modulare e personalizzabile. Infatti la classe espone tre costruttori che permettono di specificare progressivamente il provider, la dimensione della memoria e i tool, ricadendo su valori di default per i parametri non forniti. Come si può vedere nel Listato 3.2

Un altro meccanismo di personalizzazione è il metodo *changeModel(modelProvider: ModelProvider)* in modo da poter cambiare il modello a runtime. Un altro parametro modificabile attraverso il metodo *changeMemory(newSizeMemoryWindow: int)* è la grandezza della memoria, ovvero il numero di messaggi che il chatbot può tenere in memoria, ciò avviene attraverso una *sliding window* che scorre all'arrivo di ogni messaggio. Siccome la classe Agent in

Listato 3.2: Firma dei vari costruttori dell'Agent

```
public Agent(ToolsHandler tools) { ... }

public Agent(ToolsHandler tools, ModelProvider modelProvider) { ... }

public Agent(ToolsHandler tools, ModelProvider modelProvider, int sizeMemoryWindow
    ) { ... }
```

Listato 3.3: Definizione dell'interfaccia dell'Assistant e builder all'interno dell'Agent

```
interface Assistant {
    @SystemMessage(SYSTEM_PROMPT)
    String chat(String message);

    @SystemMessage(SYSTEM_PROMPT)
    Result<String> chat(UserMessage message);
}

private Assistant buildAssistant() {
    return AiServices.builder(Assistant.class)
        .chatModel(model)
        .chatMemory(chatMemory)
        .tools(tools)
        .build();
}
```

realtà incapsula la classe *Assistant* di LangChain4j, vedasi Listato 3.3, che è la vera implementazione dell'agente, è stato introdotto il metodo *buildAssistant() : Assistant* in modo da non dover distruggere e ricreare l'agente da zero in caso di cambiamenti nei parametri configurabili.

Sistema prompt e calibrazione: Un componente fondamentale nello sviluppo dell'Agente è rappresentata dal *System Prompt*, quest'ultimo dà le direttive comportamentali all'agente, che verranno seguite rigidamente. Il System Prompt creato si divide in sezioni per essere compreso più facilmente dall'LLM. La prima parte specifica il macro compito dell'agente in modo che l'Agente risponda sempre in modo contestualizzato al problema. Dopodiché vengono fatti vari elenchi di direttive e indicazioni da seguire. Ad esempio viene chiarito quale sia l'utente target e specificate le regole da seguire al riguardo, soprattutto per non mostrargli dettagli che a lui non sono interessanti. Di seguito si trovano le regole di comportamento ed il processo decisionale, in modo che l'Agente stesso possa guidare la

conversazione attraverso gli stadi visti nella Sezione 3.1.1. Nella sezione conclusiva del System Prompt si trovano le specifiche sui tools, qui vengono definite le condizioni in cui ciascun tool deve essere utilizzato, ed in particolar modo è mostrato il formato che deve avere una formazione per essere validata e poi inviata, viene fatto attraverso schema generale ed esempio.

3.3.3 Implementazione della View e del Controller

Gestione del threading: Un aspetto implementativo critico nella realizzazione di interfacce grafiche con Swing riguarda la gestione della concorrenza. Swing adotta un modello a thread singolo: tutte le operazioni sui componenti grafici devono essere eseguite sull'*Event Dispatch Thread* (EDT), il thread dedicato alla gestione degli eventi dell'interfaccia. Eseguire operazioni bloccanti sull'EDT, come una chiamata a un modello linguistico remoto, che può richiedere diversi secondi, causerebbe il congelamento dell'intera interfaccia per tutta la durata dell'operazione.

Per evitare questo problema, il *Controller* esegue la chiamata al modello in un thread separato, riservando all'EDT esclusivamente gli aggiornamenti grafici, come mostrato nel Listato 3.4. Prima di avviare il thread, il Controller disabilita l'input dell'utente e mostra un indicatore di caricamento; al termine, indipendentemente dall'esito, ripristina lo stato dell'interfaccia.

Gli aggiornamenti grafici invocati dall'interno del thread (`showAgentMessage`, `showLoading`, `setEnabled`) non modificano direttamente i componenti Swing: ogni metodo della *ChatView* che tocca l'interfaccia è wrappato in una chiamata a `SwingUtilities.invokeLater()`, che accoda l'operazione sull'EDT garantendo la thread-safety dell'aggiornamento grafico. Vedasi Listato 3.5.

Interfaccia grafica: L'interfaccia grafica è composta da tre classi principali con responsabilità distinte. *ChatWindow* estende `JFrame` e implementa l'interfaccia *ChatView*: è il punto di contatto tra il Controller e la View, e si occupa di wrappare tutte le operazioni sui componenti grafici in chiamate a `SwingUtilities.invokeLater()`. *ChatPanel* gestisce il layout generale della finestra di chat, composto da tre aree: la zona messaggi, una barra di caricamento

Listato 3.4: Metodo centrale del Controller per processare in parallelo l'input dell'utente

```
public void processUserInput(String userMessage) {
    view.setEnabled(false);
    view.showLoading(true);

    new Thread(() -> {
        try {
            String response = model.askAgent(userMessage);

            // display the response in the GUI
            view.showAgentMessage(response);
        } catch (Exception e) {
            view.showAgentMessage("Error: " + e.getMessage());
        } finally {
            view.showLoading(false);
            view.setEnabled(true);
        }
    }).start();
}
```

Listato 3.5: Esempio di metodo di entry point della ChatWindow

```
@Override
public void showAgentMessage(String response) {
    SwingUtilities.invokeLater(() -> {
        getChatPanel().displayAgentMessage(response);
    });
}
```

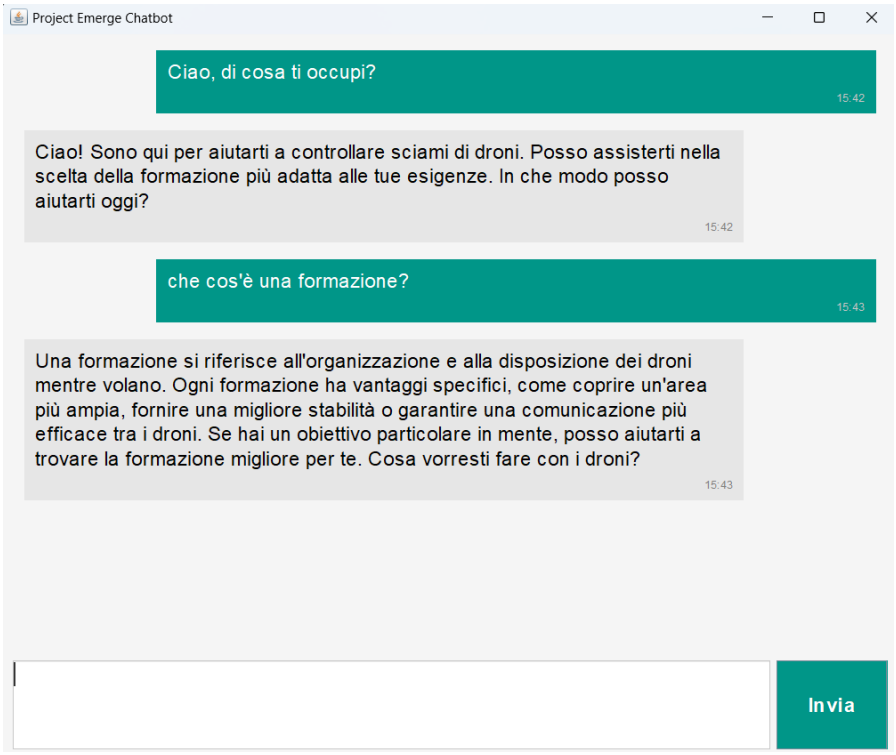


Figura 3.9: Interfaccia utente

indeterminata mostrata durante l'attesa della risposta, e un pannello inferiore con campo di testo e pulsante di invio, come mostrato in Figura 3.9. *MessagePanel* è responsabile della visualizzazione dei singoli messaggi sotto forma di bolle, con allineamento a destra per i messaggi dell'utente e a sinistra per quelli dell'agente.

Un aspetto implementativo non banale riguarda la gestione del punto di connessione tra View e Controller. La *ChatPanel* non conosce il Controller: espone invece un metodo `setOnMessageSent(Consumer<String>)`, mostrato nel Listato 3.6, che accetta una callback invocata ogni volta che l'utente invia un messaggio. È il Controller a registrare questa callback in fase di costruzione, passando il proprio metodo `processUserInput`.

Questo approccio, analogo al pattern *Observer* nella sua forma più elementare, mantiene la View completamente ignara della logica applicativa, delegando al Controller la definizione del comportamento da eseguire all'invio di ogni messaggio.

Listato 3.6: Implementazione del pattern Observer per mantenere separati View e Controller

```
// Nel Controller
this.view.setOnMessageSent(userInput -> processUserInput(userInput));

// In ChatPanel, invocata alla pressione del tasto Invio o del pulsante
private void sendButtonPressed() {
    String userMessage = inputArea.getText().trim();
    if (!userMessage.isEmpty() && onSendButtonPressedHandler != null) {
        displayUserMessage(userMessage);
        onSendButtonPressedHandler.accept(userMessage);
        inputArea.setText("");
    }
}
```

3.4 Validazione

3.4.1 Benchmark, dataset e modelli valutati

In sistemi non deterministici come quelli costruiti sull'IA, effettuare test tradizionali risulta poco efficace proprio a causa di questa natura probabilistica. Si predilige quindi un approccio di *benchmarking*, ovvero la valutazione sistematica e il confronto delle prestazioni di uno o più modelli su un insieme di test standardizzati, utilizzando metriche definite e condizioni di esecuzione controllate. L'obiettivo di questa operazione, condotta sui diversi modelli, è valutarne la capacità di utilizzare il tool o i tool necessari a soddisfare una richiesta specifica.

Per effettuare tale benchmark è stato definito un *dataset* di 11 domande; ciascuna di esse è caratterizzata da un id identificativo, dal testo della domanda, da una breve descrizione utile a renderla semanticamente esaustiva, e da una lista di tool attesi. Questi ultimi costituiscono la vera metrica di valutazione del benchmark: il modello riceve un punto solo se utilizza, per quella domanda, esattamente i tool attesi. Le domande sono suddivise in quattro categorie, ciascuna caratterizzata da una diversa difficoltà apparente. Il numero di domande assegnato a ogni categoria riflette una valutazione euristica della frequenza con cui ci si aspetta che richieste di quel tipo vengano formulate o in base a quanto viene ritenuto interessante come test da effettuare.

- **0-tool** [n. domande: 1]: richiesta generica che non implica l'utilizzo di alcun tool.

- **1-tool** [n. domande: 7]: richiesta che rappresenta l'interazione più classica, in cui si richiede un servizio che necessita di un solo tool, come ad esempio "dimmi tutte le formazioni disponibili", "valida questa formazione..." o "invia questa formazione...".

- **2-tool** [n. domande: 2]: richiesta che aggiunge complessità rispetto alla categoria precedente, in quanto richiede di effettuare due azioni in sequenza.

- **3-tool** [n. domande: 1]: richiesta specifica che richiede di utilizzare in sequenza tutti i tool principali messi a disposizione.

Come anticipato, i modelli di IA non forniscono risultati deterministici; per questo motivo la validazione di un modello non può basarsi su una sola risposta per domanda. Ciascuna delle 11 domande è stata quindi ripetuta 10 volte per ogni modello, e contando il numero di successi è stato possibile calcolare l'accuratezza (%) e il tempo medio di risposta. Per rendere più pulito e lineare il processo di validazione, l'Agent è stato fornito di tool stub, ovvero privi di implementazione operativa, per farlo sono stati passati al ToolsHandler lo StubSender e lo StubFormationProvider.

I modelli presi in considerazione sono 10, appartenenti a 3 provider diversi: `GEMINI` (3), `OPENAI` (3) e `OLLAMA` (4), Tabella 3.1. Tali modelli si dividono in due categorie: modelli *cloud*, in cui il calcolo e il ragionamento vengono effettuati su un'infrastruttura remota e il sistema si limita a inviare la richiesta e ricevere la risposta; e modelli *locali*, scaricati ed eseguiti direttamente sulla macchina su cui gira il programma, con un costo computazionale a carico dell'hardware disponibile. Questi due approcci presentano vantaggi e svantaggi complementari. Il principale vantaggio dei modelli locali è la sicurezza: non esponendo mai i dati della richiesta e della conversazione al di fuori della propria macchina, risultano preferibili quando il sistema deve trattare dati sensibili. Lo svantaggio è che le loro capacità dipendono strettamente dalle caratteristiche hardware del sistema su cui vengono eseguiti: solo un calcolatore sufficientemente potente può garantire un funzionamento adeguato, una garanzia che invece è sempre presente quando si utilizzano modelli cloud.

Tabella 3.1: Modelli valutati nel benchmark

Modello	**Provider**	**Tipo**	**Param.**	**Dim. disco**	**Contesto**
`mistral`	OLLAMA	Locale	7B	4,4 GB	32K
`mistral:7b-instruct`	OLLAMA	Locale	7B	4,4 GB	32K
`qwen2.5:3b`	OLLAMA	Locale	3B	1,9 GB	32K
`llama3-groq-tool-use`	OLLAMA	Locale	8B	4,7 GB	8K
`gemini-2.5-flash-lite`	GEMINI	Cloud	n.d.	—	1M
`gemini-2.5-flash`	GEMINI	Cloud	n.d.	—	1M
`gemini-2.5-pro`	GEMINI	Cloud	n.d.	—	1M
`gpt-4o-mini`	OPENAI	Cloud	n.d.	—	128K
`gpt-4.1-mini`	OPENAI	Cloud	n.d.	—	1M
`gpt-5.1`	OPENAI	Cloud	n.d.	—	400K

Nota: i parametri dei modelli cloud non sono pubblicamente dichiarati dai rispettivi provider. La dimensione su disco è indicata solo per i modelli locali.

3.4.2 Risultati

Per rendere più facilmente fruibili i risultati di questa operazione è stato programmato uno script python che rappresentasse graficamente i risultati trovati. Di seguito si trovano i diagrammi prodotti con spiegazione.

La prima rappresentazione grafica importante da sviluppare è quella sull'effettiva efficacia generale dei modelli, ovvero l'*accuracy overall*, in Figura 3.10. Tale dato indica la capacità effettiva dei modelli a svolgere il compito che gli è stato affidato, sommando i risultati di tutte le domande. Questo quindi mette in risalto quale modello sia capace di comportarsi come richiesto in un utilizzo generale del programma, quindi facendo richieste da 0/1/2/3 tool alla volta. Si osserva la scarsa efficacia dei modelli Mistral di ollama. Tralasciando però questi due gli altri modelli hanno capacità rilevanti, raggiungendo fino al 70,9% di accuratezza. Proseguendo l'analisi dei dati trovati si osserva che i dati possono essere contestualizzati alla luce di due fattori: da un lato, una parte delle domande del dataset non è banale, richiedendo il successo congiunto di 2 o 3 tool; dall'altro, anche i fallimenti di sistema, come le richieste annullate per timeout o per eccesso di chiamate, gestite tramite un meccanismo di retry, vengono comunque conteggiati come errori se continuano a fallire. Tenendo conto di questi elementi, i dati assumono una prospettiva diversa.

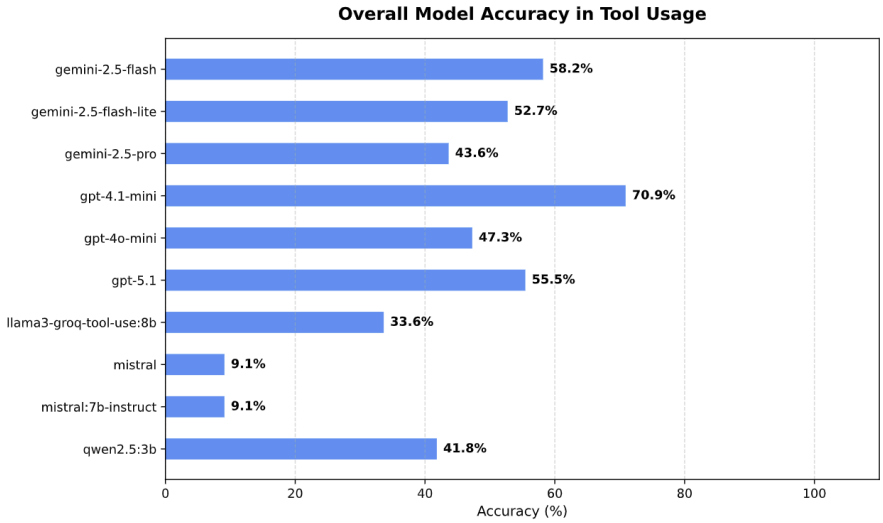


Figura 3.10: Diagramma dell'accuracy overall

Per visualizzare nel dettaglio la capacità di ogni modello di rispondere correttamente ad una richiesta specifica è stata rappresentata graficamente utilizzando una heatmap, in Figura 3.11. Il valore di ogni domanda per modello rappresenta l'accuracy e varia nell'intervallo [0,1] a seconda del numero di volte in cui è stata eseguita correttamente la richiesta sui 10 tentativi dati. Ad esempio il valore 0,5 indicherà che tale modello potrebbe rispondere correttamente la metà delle volte. Questo grafico permette inoltre di individuare le categorie di domande più difficili per i modelli. Definiamo perciò la categoria di appartenenza di ogni domanda: 0-tool=(Q1), 1-tool=(Q2, Q3, Q4, Q5, Q6, Q7, Q8), 2-tool=(Q9, Q10), 3-tool=(Q11). Si osserva innanzitutto l'incapacità sistematica di tutti i modelli di completare correttamente la richiesta di categoria 3-tool (Q11). Questo dato va però contestualizzato: l'agente riceve nel proprio *system prompt* la direttiva esplicita di richiedere conferma all'utente prima di inviare una formazione. Poiché Q11 richiede di eseguire ricerca, validazione e invio in un'unica richiesta senza alcuna conferma intermedia, essa entra in conflitto diretto con le linee guida comportamentali dell'agente. Il fallimento su questa domanda non è quindi indicativo di un'incapacità del modello, bensì di una coerenza con il comportamento desiderato del sistema. Analogamente, le domande Q4 e Q7, che richiedono l'utilizzo del tool di invio della formazione, mostrano difficoltà generalizzate, riconducibili alla stessa

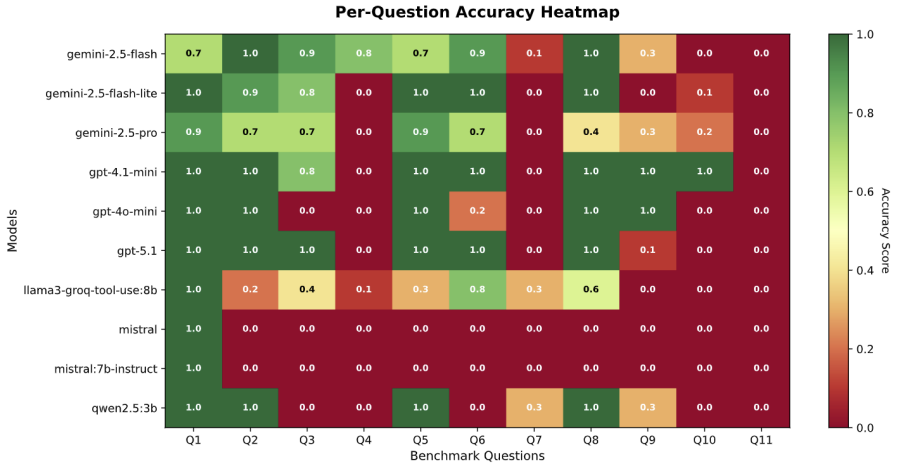


Figura 3.11: Diagramma heatmap, rappresentante l'accuracy dei modelli per ogni domanda

causa: l'agente tende a richiedere esplicitamente la conferma dell'utente prima di procedere all'invio, comportamento deliberatamente progettato per limitare azioni indesiderate.

L'usabilità di un modello non è definita solamente dalla sua accuracy, infatti un altro dato fondamentale preso in considerazione è stato il *tempo di risposta*. Misurato dal momento in cui parte la richiesta al momento in cui arriva la risposta. Questo dato influisce molto sulla vera efficacia di un modello e su quanto sia effettivamente utilizzabile. La struttura a "chat conversazionale" del programma necessita uno scambio continuo di messaggi ed un attesa prolungata per una risposta rende inutilizzabile la soluzione. I modelli in cloud in questa prova sono risultati perfettamente soddisfacenti, attese di pochissimi secondi, da un minimo di 1 secondo ad un massimo di 12 per il modello peggiore tra essi, vedasi Figura 3.12. Al contrario i modelli in locale essendo strettamente legati alle performance del calcolatore su cui si trovano tendono ad avere tempi più dilatati aggirandosi intorno ai 30 secondi, come riportato in Figura 3.13. Anche se con prestazioni non banali per due modelli in particolare, aggirandosi intorno ai 30 secondi, con l'eccezione di due modelli che mostrano prestazioni significativamente migliori. Per questo si è deciso di riportare tali dati in due grafici separati. Dal grafico possiamo anche notare modelli che hanno impiegato tempi anomali (*Outliers*) nel fornire la risposta, che li rende potenzialmente meno affidabili. Il valore comunque più importante

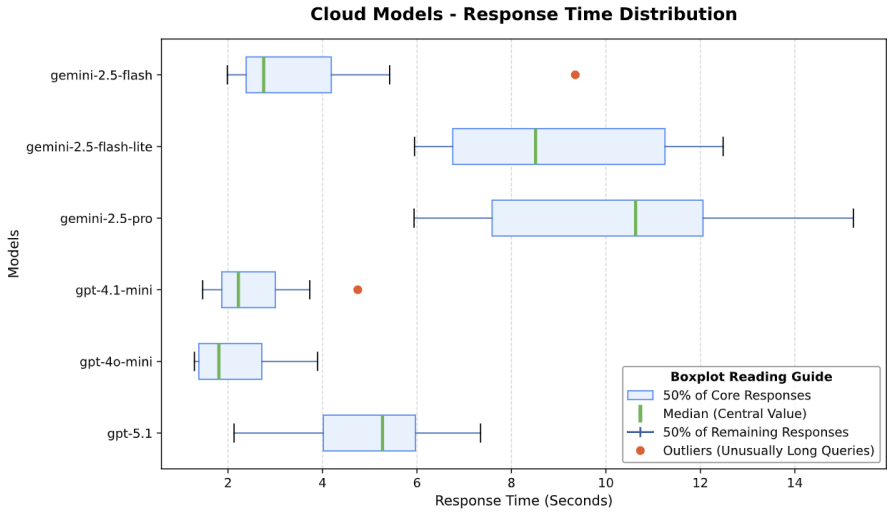


Figura 3.12: Diagramma di Response Time dei modelli in cloud

da tenere in considerazione è la mediana (linea verticale verde) che rappresenta il valore centrale dello span di risultati.

L'unione dei dati mostrati in precedenza fornisce un riferimento chiaro su quale sia il modello più efficace per la soluzione sviluppata. Rappresentando perciò la relazione *accuracy x response time* si individua con facilità il modello che concilia velocità e precisione. Esso si posizionerà nella zona superiore e a sinistra del grafico. In caso non ci sia un modello superiore in assoluto rispetto a queste due variabili è possibile scegliere quello che è più in linea con le necessità specifiche. Se si necessita di avere più probabilità che il chatbot esegua correttamente il compito richiesto si potrà scegliere il modello che si posiziona più in alto nel grafico, mentre se stiamo cercando un modello di cui ci assumiamo i rischi degli errori ma che deve rispondere nel minor tempo possibile si potrà selezionare quello posizionato più a sinistra. Da grafico in Figura 3.14 si osserva che il modello in definitiva considerabile migliore è *gpt-4.1-mini*[4], nonostante non sia il più veloce supera di un buon 20% di accuracy *gpt-4o-mini*[5]. Perciò si può considerare il migliore. Mentre per i modelli in locale, graficati in Figura 3.15, il migliore è *qwen2.5:3b*[10] sia in accuracy che in velocità di risposta.

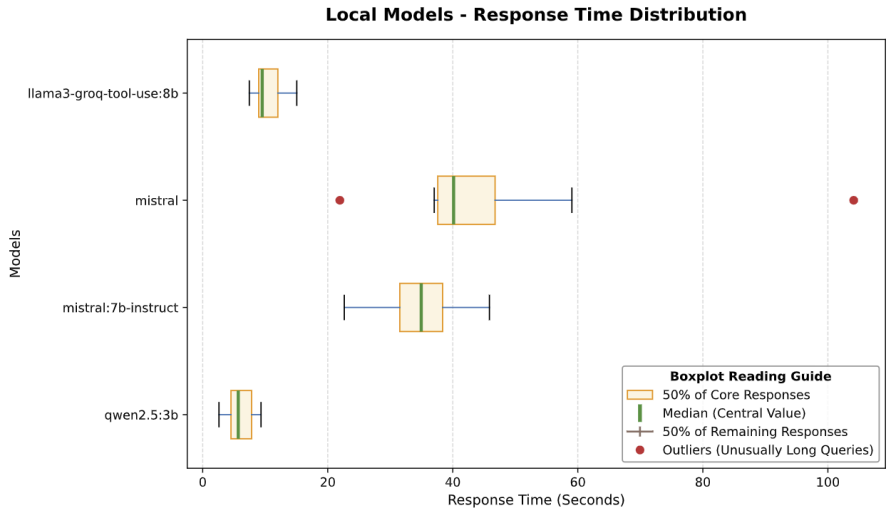


Figura 3.13: Diagramma di Response Time dei modelli in locale

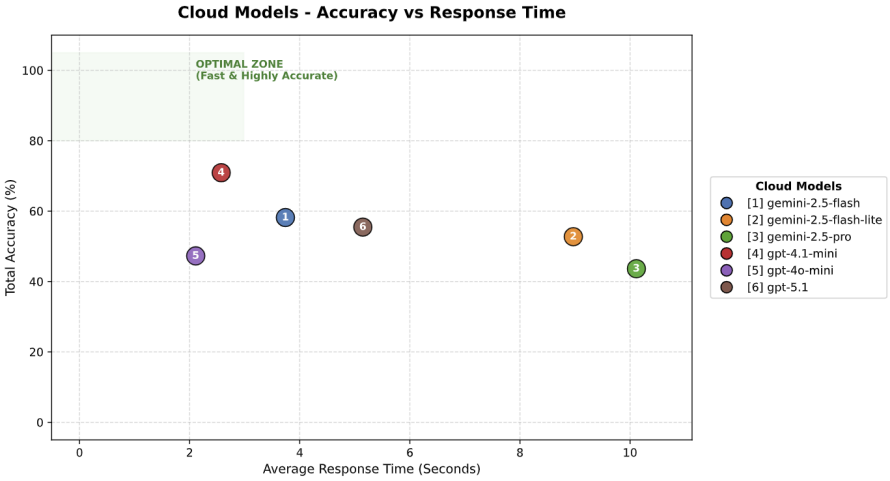


Figura 3.14: Diagramma di accuracy x response time dei modelli in cloud

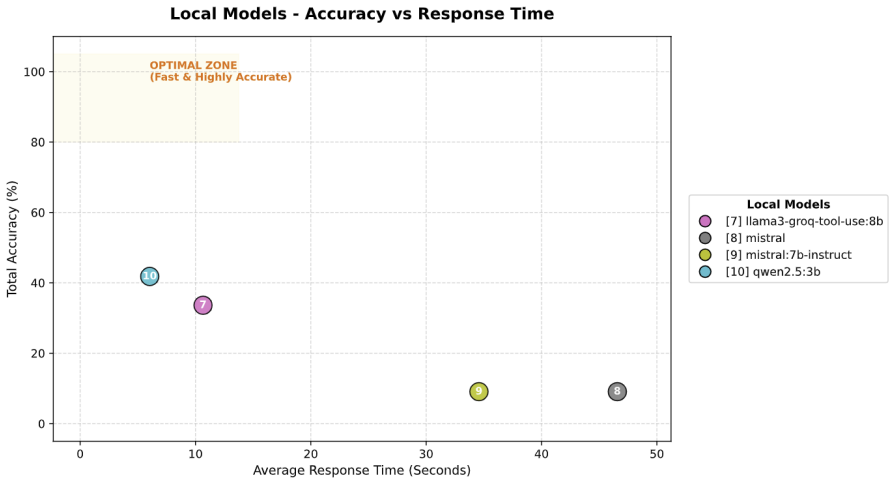


Figura 3.15: Diagramma di accuracy x response time dei modelli in locale

3.4.3 Considerazioni finali

I dati riportati permettono di valutare le effettive potenzialità della soluzione e forniscono indicazioni concrete sul comportamento del chatbot in condizioni di utilizzo reale. Va tenuto in considerazione che i modelli analizzati rappresentano una minima parte dell'offerta attuale: i provider selezionati sono stati scelti in base alla facilità di integrazione e alla disponibilità di documentazione accurata, ma altri modelli, sia cloud che locali, potrebbero produrre risultati sensibilmente diversi, considerato anche il ritmo di miglioramento con cui nuovi modelli vengono rilasciati. Analogamente, un dataset più ampio o strutturato diversamente potrebbe rivelare aspetti del comportamento dell'agente non catturati da quello attualmente utilizzato.

Questa apertura a sviluppi futuri non è casuale, ma è una conseguenza diretta delle scelte architetturali discusse nel capitolo di Design. L'enumerazione *ModelProvider* e la *ChatModelFactory* (Sezione 3.2.2) sono state progettate con l'obiettivo esplicito di rendere l'aggiunta di nuovi modelli un'operazione minimamente invasiva: estendere il benchmark a un nuovo provider richiede unicamente l'aggiunta di un valore nell'enumerazione, senza alcuna modifica al resto del sistema. Allo stesso modo, il modulo di validazione è stato implementato in modo da poter essere alimentato con dataset aggiornati senza alcun intervento sul codice. La sezione di validazione è quindi, a tutti gli effetti, un componente del sistema

progettato per crescere insieme ad esso.

Capitolo 4

Conclusione e sviluppi futuri

Il lavoro svolto ha affrontato la progettazione architetturale, lo sviluppo e la validazione di un chatbot capace di interfacciarsi con un sistema autonomo di controllo di sciami di droni. L'obiettivo era semplificare l'accesso al sistema per utenti privi di competenze tecniche specifiche, rendendolo fruibile senza richiedere alcuna formazione preliminare. Tale obiettivo può considerarsi raggiunto: la soluzione sviluppata rende accessibile un sistema che in precedenza richiedeva conoscenze specialistiche, soddisfacendo al contempo tutti i requisiti definiti in fase di analisi, tra cui la modularità e l'estendibilità, fondamentali per gli sviluppi futuri del progetto.

La fase di validazione fornisce dati concreti a supporto di questa conclusione: il chatbot, mediante l'utilizzo del modello più adatto, raggiunge fino al 70,9% di accuratezza, confermando l'efficacia della soluzione nel gestire correttamente le interazioni previste dal dominio. Il modulo di validazione stesso va inoltre considerato come uno strumento autonomo: anch'esso modulare, estendibile e riutilizzabile in contesti analoghi.

I limiti della soluzione riguardano principalmente i modelli linguistici locali, i quali non risultano completamente adeguati per un'implementazione di tipo chatbot a causa dei tempi di risposta prolungati, strettamente dipendenti dalle caratteristiche hardware del sistema su cui vengono eseguiti. Un secondo limite riguarda la trasparenza della validazione: il system prompt contiene direttive che impediscono l'esecuzione automatica di tool consecutivi, richiedendo una conferma

esplicita da parte dell'utente prima di procedere. Si tratta di una scelta progettuale consapevole e necessaria per la sicurezza dell'interazione, ma che introduce un bias nei risultati della fase di test, in particolare nelle categorie multi-tool. A questo proposito, uno studio più approfondito del prompting potrebbe apportare benefici significativi alla soluzione: il comportamento dell'agente dipende in modo diretto dalle istruzioni fornite nel system prompt, rendendo la qualità di quest'ultimo di primaria importanza per le prestazioni complessive del sistema.

In relazione al progetto descritto, emergono diverse direzioni di sviluppo futuro. Un contributo particolarmente interessante sarebbe la generazione di nuove formazioni direttamente da parte del chatbot: attualmente il sistema si limita a selezionare e configurare formazioni già esistenti, definite in un file Scala. Consentire all'agente di generare nuove formazioni a partire dalle richieste degli utenti amplierebbe significativamente le potenzialità del sistema, sfruttando la creatività degli utenti per estendere il catalogo disponibile. Un ulteriore sviluppo riguarda l'interfaccia utente: l'architettura MVC adottata consentirebbe di sostituire l'attuale interfaccia grafica con un'interfaccia vocale senza alcuna modifica alla logica del sistema. Più in generale, sarebbe auspicabile integrare il chatbot direttamente nella dashboard di Project Emerge, ottenendo un sistema unificato ed eliminando la necessità di avviare applicazioni separate. Per quanto riguarda la validazione, sarebbe infine opportuno ampliare il dataset con categorie più granulari e introdurre un meccanismo di gestione della conferma esplicita, così da poter valutare correttamente anche i tool di invio dei comandi nelle sequenze multi-step.

Il lavoro presentato si colloca in un momento di rapida evoluzione dei modelli linguistici e delle loro applicazioni pratiche. I risultati ottenuti suggeriscono che l'integrazione tra agenti basati su LLM e sistemi robotici complessi rappresenta una direzione promettente per rendere accessibile la tecnologia a un pubblico più ampio, indipendentemente dal dominio applicativo specifico. La natura open-source e modulare dell'architettura sviluppata intende favorire questa direzione, offrendo una base replicabile per scenari analoghi di interazione uomo-macchina mediata dal linguaggio naturale.

Bibliografia

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[FAB+26] Nicolas Farabegoli, Gianluca Aguzzi, Martina Baiardi, Angela Cortecchia, Davide Domini, Danilo Pianini, and Mirko Viroli. Project emerge: A demonstrator for self-organizing robot teams. *Science of Computer Programming*, 2026. Preprint submitted May 21, 2026.

[Mar25] Federico Mariani. Project emerge chatbot. `https://github.com/FedeMariani18/Project-Emerge-chatbot`, 2025. Ultimo accesso: giugno 2026.

[MMRS55] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon. A proposal for the dartmouth summer research project on artificial intelligence, 1955. Submitted August 31, 1955.

[RN21] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2021.

[SEN20] SENECA. Il protocollo MQTT. `https://blog.seneca.it/it/il-protocollo-mqtt/`, 9 2020. Ultimo accesso: giugno 2026.

[Vir21] Mirko Viroli. *Distributed Cooperative Intelligence: An Introduction to Aggregate Computing*. MIT Press, 2021.

Ringraziamenti

Papà, questa laurea la dedico a te. Sei la mia guida, il mio modello, il mio eroe. Mi hai insegnato i valori fondamentali della vita, tra cui la libertà che dà lo studio, la conoscenza. La dedico a te anche perché è sempre stato un po' il tuo sogno, quel piccolo rimpianto che avresti voluto inseguire. Se oggi sono qui è anche grazie a te, che mi hai fatto conoscere questo mondo e lo hai condiviso con me fin da quando ero piccolo: da quando mi lasciavi giocare a quel videogioco di Spiderman degli anni '90, da quando mi insegnasti a collegare la Wii alla TV, da quando mi sconfiggevi i boss più difficili di Super Mario. Non sono mai riuscito a spiegarti con esattezza come funziona Internet — forse devo studiare ancora un po' prima di capirlo bene anch'io. Ma è proprio questa, in fondo, la parte più bella: mi hai regalato domande che da solo non avrei saputo pormi, e sappiamo entrambi che le domande sono ciò che conta di più nel far nascere un'idea. Per questo sono preziose, perché *"il mondo è di chi ha le idee"*.

Mamma, ti ringrazio perché se sono riuscito a laurearmi in tempo è anche grazie a te, che mi hai dato la forza di continuare a impegnarmi. Non ti fermi davanti a nulla: sei una forza intramontabile, e quando la vita ti mette ostacoli davanti trovi sempre il modo di andare avanti. Mi hai insegnato la tenacia, ma soprattutto l'amore, perché è quella la tua vera forza. Le tue energie si esauriscono, ma il tuo affetto continua a scaldarmi e a tenermi vivo. Ti voglio bene, Mamma.

Poi devo ringraziare le mie sorelle, quelle birbanti. *Meggie*, tu mi hai spianato la strada lungo questo percorso universitario: la mia paura dell'ignoto era un po' meno tenebrosa grazie al tuo lanternino a precedermi. *Emma*, tu sei il mio sorriso e la mia spensieratezza: la tua gioia e la tua giocosità colorano le giornate e fanno risplendere una luce anche nei momenti più bui. Quando sei triste tu, il mondo si ingrigisce; quando sono triste io, ti bastano cinque minuti per farmi tornare il

sorriso. Vi voglio bene, ragazze.

Un enorme grazie va al mio fratello non di sangue, *Andre*: ci sei sempre nei momenti difficili e, nonostante la mia difficoltà nell'aprirmi, riesci sempre a farmi tirare fuori tutto e a liberarmi di ogni peso. Grazie.

Per concludere, ringrazio tutte le persone che mi sono state vicine in questi tre anni. Grazie *Vale* per avermi sopportato in tutti i miei scleri e le mie idiozie, sei sempre un porto sicuro. Grazie a tutti i *Rimbo*, i miei fidati compagni di viaggio, con cui continuo a crescere giorno dopo giorno. Grazie anche al resto dei *Fagiolini* che ci sono sempre e che sanno sempre strapparmi un sorriso.

Grazie *Gio* per essermi sempre stata vicina anche quando te la rendevo più difficile, e per avermi regalato un'infinità di momenti bellissimi. Grazie ai miei *compagni dell'Uni* che hanno riempito le lunghe giornate di lezioni con risate, incazzature e partite a culo presidente. Grazie *Gab* e *Marci*, siete i miei amori, sapete farmi stare bene. Grazie a tutti i ragazzi della *squadra del CUSB* di quest'anno e dell'anno scorso, perché attraverso lo sport e la vostra carica siete sempre riusciti a farmi staccare la testa da qualsiasi altro problema.

Grazie alle mie donne del *Pigiama Party*: nonostante ci vediamo poco, ogni volta siete fuochi d'artificio. Grazie alla *Marti*, alla *Lu* e all'*Ali*, mi avete regalato un posto in cui mi è facile essere al 100% me stesso. Grazie ai ragazzi del *King*, perché nonostante ci vediamo di rado ogni volta è come riabbracciare dei fratelli. Grazie ai miei amici dalle elementari, *Aldo* e *Lollo*: abbiamo camminato a lungo insieme, e questa strada mi ha portato fino a qua.

Grazie alla famiglia allargata, *Zio Filo*, *Marti*, *Dile*, *Edo*, *Mara*, *Enri*, *Vale*, *Pietro*, *Irene*, *Davide*, che mi avete dato stabilità e sicurezza quando non ne trovavo.