Università Politecnica delle Marche

Facoltà di Ingegneria

Corso Di Laurea in Ingegneria Biomedica

Sviluppo e validazione in ambiente simulato di un robot mobile assistivo basato su Vision-Language Models

Development and Simulation-Based Validation of an Assistive Mobile Robot Using Vision-Language Models

Relatore:

Francesco Ferracuti

Correlatore:

Karameldeen Ibrahim Mohamed Omer

Tesi di Laurea di:

Edoardo Orselli

A.A. 2025/2026

Indice

1 Introduzione ... 1
1.1 Contesto e motivazione ... 1
1.2 Limiti degli approcci attuali ... 2
1.3 Obiettivi del lavoro ... 2
1.4 Contributo personale ... 3
1.5 Struttura della tesi ... 3
2 Vision-Language Models ... 4
2.1 Dai modelli linguistici ai modelli multimodali ... 4
2.1.1 Il Transformer e il meccanismo di attenzione ... 4
2.1.2 I modelli linguistici di grandi dimensioni ... 5
2.1.3 Dall'elaborazione del solo testo ai modelli multimodali ... 6
2.2 Architettura di un VLM ... 6
2.2.1 Schema a tre componenti e flusso dei dati ... 6
2.2.2 Encoder visivo ... 8
2.2.3 Proiettore e spazio di rappresentazione condiviso ... 9
2.2.4 LLM e generazione della risposta ... 9
2.3 Addestramento e instruction tuning ... 10
2.4 Capacità e limiti rilevanti per la robotica ... 11
2.4.1 Capacità ... 11
2.4.2 Limiti ... 11
2.5 Modelli aperti ed esecuzione in locale ... 12
2.5.1 Motivazioni e requisiti dell'esecuzione locale ... 12
2.5.2 Qwen2.5-VL ... 13
2.5.3 Gemma 4 ... 13
2.5.4 Esecuzione tramite server locale con API compatibile OpenAI ... 14
2.6 Sintesi del capitolo ... 14
3 Tecnologie per il controllo robotico ... 15
3.1 Dal linguaggio all'azione ... 15
3.1.1 Il divario tra piano e comandi ... 15
3.1.2 Approcci end-to-end: i modelli VLA ... 16
3.1.3 Approcci gerarchici ... 17
3.1.4 L'approccio di HIMA e le sue conseguenze ... 17
3.2 ROS 2 ... 18
3.2.1 Da ROS a ROS 2: il middleware ... 18
3.2.2 Nodi, topic e messaggi ... 21
3.2.3 Sistemi di riferimento e TF2 ... 22

3.2.4 Servizi e action 23
3.2.5 Esecuzione asincrona 25
3.3 Navigazione autonoma 25
3.3.1 Il problema della navigazione 25
3.3.2 SLAM e localizzazione 26
3.3.3 Mappe di costo 26
3.3.4 Pianificazione globale e controllo locale 27
3.3.5 Nav2 e il suo behavior tree 27
3.3.6 L'interfaccia NavigateToPose 28
3.4 Simulazione e piattaforme 28
3.4.1 Simulatori: Gazebo e MuJoCo 28
3.4.2 Piattaforme: TurtleBot3 e Stretch 29
4 Architettura del sistema 30
4.1 Panoramica 30
4.1.1 Organizzazione su due ambienti 30
4.1.2 I tre livelli e il bridge 30
4.2 Livello cognitivo 31
4.2.1 Il backend 31
4.2.2 Interfaccia con il modello 32
4.2.3 Classificazione dell'intento 32
4.2.4 Generazione del piano 32
4.2.5 Memoria spaziale 33
4.2.6 Visione e sintesi vocale 34
4.2.7 Modalità autonoma 34
4.3 Interfaccia utente 34
4.4 Comunicazione tra i sistemi 36
4.4.1 Canali e porte 36
4.4.2 Invio dei piani 36
4.4.3 Mappa e posa 36
4.4.4 Immagini della camera 37
4.5 Livello fisico 37
4.5.1 Robot e simulatore 37
4.5.2 Mappa e localizzazione 37
4.5.3 Nav2 37
4.5.4 Sequenza di avvio 38
4.6 Esecuzione delle azioni 38
4.6.1 Il nodo bridge tcp_listener 38
4.6.2 La procedura execute_plan 39

4.6.3 Navigazione verso un luogo 40
4.6.4 Altre azioni 41
4.6.5 Piani concorrenti 41
4.6.6 Il pacchetto hima_bt_runner 42
4.7 Sviluppo dei nodi di navigazione 42
4.7.1 Analisi del flusso della versione precedente 42
4.7.2 Il prototipo simple_navigator 43
4.7.3 Il nodo hima_navigator 44
4.7.4 Confronto con la soluzione finale 47
4.8 Flusso di un comando 48
5 Test e validazione 51
5.1 Obiettivi e impostazione della validazione 51
5.2 Configurazione sperimentale 52
5.3 Scenari e procedura 54
5.3.1 Blocco di navigazione 54
5.3.2 Blocco sul modello 56
5.4 Metriche e criteri di successo 57
5.4.1 Navigazione 57
5.4.2 Livello cognitivo 58
5.5 Risultati della navigazione 58
5.6 Risultati del livello cognitivo 61
5.7 Discussione dei risultati 64
5.7.1 Esito dichiarato ed esito effettivo 64
5.7.2 Coerenza dell'interazione 64
5.7.3 Tempi di risposta 65
5.7.4 Validità dei risultati 65
5.8 Limiti della validazione 66
6 Conclusioni e sviluppi futuri 67
6.1 Risultati ottenuti 67
6.2 Contributo personale 68
6.3 Sviluppi futuri 69
6.3.1 Interventi sul codice 69
6.3.2 Hardware ed esecuzione del modello 69
6.3.3 Estensione delle capacità 69
6.3.4 Validazione 70
6.4 Considerazioni conclusive 70
Bibliografia 71

1 Introduzione

Questo capitolo presenta il contesto all'interno del quale si colloca il progetto HIMA, un sistema robotico assistivo che si basa sull'impiego di un modello visione-linguaggio (Vision-Language Model, VLM), incaricato di interpretare le richieste dell'utente e di pianificare le azioni di un robot mobile. Si definiscono inoltre gli obiettivi del lavoro e il contributo personale al progetto.

1.1 Contesto e motivazione

L'invecchiamento della popolazione rappresenta uno dei cambiamenti strutturali più marcati delle società europee e l'Italia ne è un esempio evidente. Al 1° gennaio 2026 le persone con almeno 65 anni residenti in Italia sono circa 14,8 milioni, pari al 25,1% della popolazione totale, mentre le persone con più di 85 anni superano i 2,5 milioni [1]. Nel panorama europeo l'Italia ha la quota di residenti anziani più alta dell'Unione, con il 24,7% al 1° gennaio 2025 a fronte di una media UE27 del 22,0% [1]. L'Istituto nazionale di statistica (Istat) ha presentato delle previsioni sull'invecchiamento della popolazione residente in Italia, secondo le quali nello scenario mediano la quota di ultrasessantacinquenni raggiungerà il 34,5% nel 2050. Le stesse previsioni indicano che la quota degli ultraottantacinquenni, la fascia d'età in cui il bisogno di assistenza è più frequente, supererà il 7% nel 2050 [2].

In parallelo all'aumento della longevità si ha una crescita della domanda di assistenza. L'indagine europea sulla salute condotta nel 2019 riporta che circa un anziano su dieci (10,6%) riferisce gravi difficoltà in almeno una delle attività base di cura della persona, come lavarsi, vestirsi o alzarsi dal letto. Per le persone con oltre 85 anni la quota sale al 37,2% [3]. Tra gli anziani con gravi difficoltà, il 44,2% dichiara di non disporre di ausili o di assistenza in misura adeguata [3]. Sebbene la famiglia rappresenti il principale sostegno, nel tempo le reti di aiuto familiare si sono ridotte [3] e il numero di anziani che vivono soli è previsto in crescita, da 4,6 a 6,6 milioni entro il 2050 [2].

Si rilevano problemi anche dal lato dell'offerta di assistenza. L'Organizzazione per la cooperazione e lo sviluppo economico (OCSE) osserva che nei Paesi membri il numero degli operatori dell'assistenza a lungo termine cresce più lentamente rispetto alla popolazione anziana e che il settore fatica ad attrarre e trattenere personale [4]. Al fine di mantenere invariato il rapporto medio di circa cinque operatori ogni cento persone con almeno 65 anni, il numero di operatori dovrebbe aumentare di 13,5 milioni entro il 2040 [4]. La stessa organizzazione indica un utilizzo più esteso della tecnologia tra le leve per aumentare la produttività dell'assistenza [4].

In questo quadro la robotica assistiva è oggetto di studio per essere integrata come possibile supporto all'autonomia della persona e anche al lavoro di chi la assiste. Si distinguono sistemi che rientrano in categorie diverse tra loro. Una prima distinzione separa quei robot che possono fornire un aiuto fisico, quali i dispositivi per la riabilitazione e gli ausili alla mobilità, da quelli che offrono supporto unicamente attraverso l'interazione sociale e non il contatto fisico [5]. Si indicano poi i robot di servizio mobili per l'ambiente domestico, capaci di muoversi in autonomia all'interno dell'abitazione, svolgere piccoli lavori domestici e offrire una forma di monitoraggio per la sicurezza e la salute [6]. Diversi studi sul campo riportano che questi robot possono contribuire al benessere degli anziani e ridurre il carico di lavoro di chi li assiste [6]. Il sistema HIMA si colloca all'interno di quest'ultima categoria.

Un robot in campo assistivo deve offrire un'interazione accessibile e semplice anche agli utenti che hanno poca familiarità con le tecnologie digitali. Nel 2025 ha utilizzato Internet negli ultimi tre mesi il 72,5% delle persone tra 65 e 74 anni e soltanto il 35,7% di quelle con almeno 75 anni [7]. Quest'ultima è proprio la fascia d'età su cui si concentra la perdita di autonomia. Per questo tipo di utenti un'interfaccia basata su menu o su comandi da memorizzare può costituire una barriera. Il tempo necessario per imparare a usare l'interfaccia di un robot varia infatti in base al profilo cognitivo della persona e alla sua esperienza con il computer [6]. Da qui l'interesse per interfacce che riducano al minimo lo sforzo appena descritto, come quelle basate su sistemi in grado di comprendere il linguaggio naturale.

1.2 Limiti degli approcci attuali

Nei robot assistivi sviluppati finora l'interazione avviene attraverso interfacce progettate per un insieme chiuso di funzioni, basate su comandi impartiti con pulsanti di chiamata o con la voce. Nella rassegna di Asgharian et al. sui robot di servizio mobili per anziani, molti dei robot presi in esame si controllano attraverso menu presentati su uno schermo tattile, talvolta affiancati dalla voce, e diverse funzioni risultano ancora predefinite [6]. La stessa rassegna indica, tra i fattori che riducono la diffidenza degli anziani verso queste tecnologie, la facilità d'uso e la velocità con cui si impara a utilizzarle [6]. In questi sistemi è quindi l'utente ad adattarsi alla macchina, in quanto deve conoscerne le funzionalità e il modo di utilizzo. Inoltre le richieste che non rientrano nello schema previsto non vengono comprese dal sistema.

Un'alternativa può essere offerta tramite l'impiego dei modelli linguistici di grandi dimensioni (Large Language Model, LLM) e la loro estensione multimodale, i VLM. Queste tecnologie sono capaci di interpretare richieste formulate liberamente dall'utente in linguaggio naturale e scomporle in una sequenza ordinata di passi [8]. La loro integrazione nel contesto assistivo permette di uscire dagli schemi di comando prefissati in fase progettuale e consente a chi utilizza il sistema di svincolarsi dalla rigidità dei comandi e di interagire in maniera più semplice. Il compito di tradurre la richiesta in operazioni passa in questo modo dall'utente al sistema.

Il problema aperto è il passaggio dal linguaggio all'azione. Un modello linguistico può interpretare le richieste che gli si presentano e in uscita produce testo, ma non conosce a priori le capacità del robot e l'ambiente in cui opera. Lavori come SayCan mostrano che piani d'azione eseguibili possono essere ottenuti se le proposte del modello sono vincolate alle abilità di cui il robot è in possesso [9]. Il piano deve inoltre essere eseguito in modo affidabile e il sistema deve riuscire a rispondere alle richieste in tempi compatibili con un uso reale. Infine, un robot che opera all'interno di un'abitazione raccoglie dati personali dell'utente. Un requisito importante è quindi la tutela della riservatezza, condizione per la fiducia dell'utente nel sistema [6]. La possibilità di eseguire il modello in locale evita che i dati raccolti siano inviati a server esterni, ma si scontra con l'elevato costo computazionale dei modelli con miliardi di parametri [10].

1.3 Obiettivi del lavoro

L'obiettivo principale di questa tesi è contribuire allo sviluppo del sistema HIMA e raccogliere dati sperimentali allo scopo di validarne il funzionamento. Nel progetto HIMA si impiega un VLM, eseguito in locale, che interpreta le richieste dell'utente e pianifica una serie di azioni che un robot mobile esegue tramite ROS 2 (Robot Operating System 2) e il sistema di navigazione autonoma Nav2.

Il lavoro si concentra sul collegamento tra il piano prodotto dal modello e la sua esecuzione e sulla misura del comportamento del sistema in condizioni controllate.

L'obiettivo da raggiungere può essere scomposto in tre fasi. Inizialmente ci si concentra su come collegare il piano generato dal modello all'esecuzione della navigazione, attraverso lo sviluppo di nodi ROS 2 dedicati. In seguito si verifica sperimentalmente il comportamento del livello fisico e del livello cognitivo, i quali vengono valutati separatamente, con una serie di prove ripetute e metriche ben definite. Come ultima fase si valuta la praticabilità dell'esecuzione locale del modello su una macchina di uso comune, con risorse limitate e priva di una scheda grafica dedicata. I criteri alla base di questa valutazione sono il tempo di risposta e la velocità di elaborazione.

Il lavoro si svolge interamente in ambiente simulato, con il robot TurtleBot3 burger nel simulatore Gazebo. Le prove su un robot reale non rientrano negli obiettivi di questa tesi.

1.4 Contributo personale

Prima di arrivare alla versione finale descritta in questa tesi, il progetto HIMA ha attraversato più versioni, non tutte realizzate nell'ambito di questo lavoro.

In fase preliminare, nella versione precedente del sistema è stata svolta un'analisi del flusso di comandi, grazie alla quale, insieme alle prove dei nodi sviluppati, si è osservata l'assenza di un collegamento tra il piano prodotto dal modello e Nav2. In seguito ai requisiti individuati sono stati sviluppati due nodi ROS 2 per integrare l'azione di navigazione, `simple_navigator` e `hima_navigator`, descritti con il codice commentato nella Sezione 4.7. Tra le attività svolte rientrano anche la configurazione dell'ambiente di sviluppo e la correzione di alcuni malfunzionamenti della versione precedente. Infine sono state impostate ed eseguite prove sperimentali ripetute in diversi scenari, con la scelta delle metriche e dei criteri di successo e l'analisi dei risultati raccolti.

Nella versione finale, descritta nel Capitolo 4, l'esecuzione dei piani d'azione e la navigazione sono affidate al bridge `tcp_listener`, che non fa parte del contributo personale.

1.5 Struttura della tesi

Nei Capitoli 2 e 3 si presentano le basi su cui poggia il sistema e i concetti necessari per comprenderne l'architettura. Il Capitolo 2 è dedicato ai modelli visione-linguaggio, su cui si basa il livello cognitivo, con particolare attenzione ai limiti rilevanti per la robotica e all'esecuzione locale. Il Capitolo 3 affronta il passaggio dal linguaggio naturale all'azione del robot e presenta gli strumenti del livello fisico, cioè ROS 2, Nav2 e l'ambiente di simulazione.

Il Capitolo 4 descrive l'architettura e il funzionamento di HIMA nella versione finale e riporta il contributo personale al progetto. In particolare la Sezione 4.7, dedicata ai nodi di navigazione, corrisponde alla prima fase dell'obiettivo descritto nella Sezione 1.3.

Nel Capitolo 5 si riporta la validazione sperimentale del sistema, che corrisponde alla seconda e alla terza fase. Si descrivono la configurazione adottata nelle prove, gli scenari e le metriche, i risultati sono poi presentati separatamente per il livello fisico e per il livello cognitivo.

Infine nel Capitolo 6 si riassumono le osservazioni sul sistema tratte dai risultati del capitolo precedente, e si propone una serie di sviluppi necessari per superare i limiti riscontrati.

2 Vision-Language Models

Il sistema descritto in questa tesi affida a un modello visione-linguaggio (VLM) il compito di interpretare i comandi dell'utente e di andarli a trasformare in una sequenza di azioni eseguibili dal robot.

Comprendere che cosa sia un modello di questo tipo, come sia costruito e quali siano i suoi limiti è quindi il presupposto per discutere, nei capitoli successivi, l'architettura del sistema e i risultati dei test.

Con il termine *Vision-Language Model* si indica un modello che sia in grado di ricevere in ingresso comandi in linguaggio naturale (testo semplice) e immagini, per produrre in uscita una risposta testuale [11].

Rispetto ad un modello linguistico (LLM – Large Language Model) che è limitato principalmente all'elaborazione di sequenze di parole, i VLM si estendono anche agli input visivi. Il modello non viene addestrato a risolvere un singolo problema, come può essere per i sistemi di visione artificiale specializzati in un singolo compito, ma sono modelli dotati di elaborazione visiva e vanno a combinare le informazioni ricevute in forma testuale con il contesto fornito tramite immagine. La forza di questi modelli sta appunto nell'ampliare le specifiche di ragionamento, offrendo la possibilità di sfruttare la percezione spaziale per migliorare la comprensione del contesto fornita dall'utente e di pianificare azioni in funzione dell'ambiente circostante [8].

In questo capitolo si ricostruisce anzitutto il percorso che dai modelli linguistici ha condotto ai modelli multimodali, si dedicherà poi una parte nel descrivere l'architettura di questi modelli, introducendo anche qualche nozione per quanto riguarda l'addestramento e l'implementazione dei VLM. Infine si conclude con dei cenni per quanto riguarda le capacità e i limiti che si rilevano quando un VLM viene impiegato nella robotica e verranno trattati anche i modelli utilizzati nel progetto.

2.1 Dai modelli linguistici ai modelli multimodali

2.1.1 Il Transformer e il meccanismo di attenzione

I modelli che vengono di seguito trattati discendono tutti da una comune architettura di rete neurale, il Transformer [12], introdotta nel 2017 per compiti di traduzione automatica.

Il transformer è un'architettura di rete neurale che si basa su un meccanismo di autoattenzione, il quale consente di calcolare la rappresentazione di ciascun elemento della sequenza di dati che si riceve in input come combinazione pesata di tutti gli altri elementi. Si va quindi a considerare la rappresentazione che ciascun elemento ha all'interno del contesto, dato dalla somma di tutti gli elementi. Questo metodo di analisi si distingue da una procedura strettamente sequenziale adottata dalle reti ricorrenti che l'avevano preceduta.

Per essere elaborato, il testo deve per prima cosa essere suddiviso in unità elementari, che vengono definite con il termine token, i quali possono corrispondere a parole o frammenti di esse.

Ogni token viene poi associato ad un vettore numerico, definito embedding, il quale va a dare una rappresentazione del valore del singolo token in uno spazio N-dimensionale. In questo spazio vettoriale che viene creato, il token ha un determinato valore e in questo senso token usati in contesti simili vengono rappresentati da vettori che occupano posizioni vicine. Il transformer è caratterizzato

da diversi strati, all'interno dei quali ciascun vettore viene trasformato ripetutamente, e ad ogni strato il meccanismo di attenzione permette a ciascun token di raccogliere informazioni dagli altri della sequenza.

Dal punto di vista matematico, ogni token viene trasformato in tre vettori tramite prodotti matriciali (trasformazioni):

- Query (Q): che cosa il token sta cercando negli altri token
- Key (K): che cosa il token offre agli altri
- Value (V): l'informazione che il token trasmette se viene selezionato

Si va a calcolare il prodotto scalare tra la Query di un token e la Key di tutti i token della sequenza, compreso sé stesso, in questo modo si ottiene quello che viene definito un punteggio di affinità.

I punteggi vengono poi elaborati attraverso una particolare funzione (funzione Softmax) che ne valuta il peso (la cui somma per tutti i token della sequenza restituisce 1).

Infine i pesi calcolati vengono moltiplicati per i vettori Value per creare una rappresentazione del singolo token che dipende da tutto il contesto [12].

Due conseguenze di questa architettura sono rilevanti per il seguito. La prima è la sua generalità: ogni dato che si può esprimere come una sequenza di vettori, a sua volta può essere elaborato da questo tipo di rete. Questa proprietà sta alla base dell'estensione alle immagini.

Nei VLM il meccanismo dei transformer non si applica solo alle parole, ma anche alle immagini. Nella Sezione 2.2 verrà trattato il principio di scomposizione dell'immagine in token.

La seconda è il costo computazionale, in quanto ogni token viene soppesato tramite confronto con gli altri, il carico di calcolo del meccanismo di attenzione cresce in modo quadratico con la lunghezza della sequenza [10], questo aspetto condiziona l'impiego e l'esecuzione di tali modelli su hardware limitato.

2.1.2 I modelli linguistici di grandi dimensioni

I Large Language Model (LLM) sono modelli basati su un'architettura Transformer, addestrati su grandi quantità di testo con l'obiettivo di prevedere il token successivo di una sequenza [13].

Da questo compito elementare, che viene ripetuto su tantissimi esempi, emergono le capacità che vengono poi utilizzate, quali la sintassi, le conoscenze e come seguire le istruzioni. Il modello in questo senso non viene sottoposto ad una fase di addestramento in cui deve apprendere come si risponde alle domande o capire come eseguire le richieste dell'utente. Queste sue abilità nascono come effetto collaterale della previsione del token successivo.

La risposta viene generata con una tecnica di autoregressione, il modello non produce la risposta in blocco, genera un token che viene applicato in fondo alla sequenza in ingresso, ripetendo questo processo fino a concludere il testo. Ogni token in questo senso costa un passaggio completo attraverso la rete.

Da ciò ne conseguono due osservazioni pratiche da considerare per un sistema che deve reagire in tempi utili all'utente. La risposta non può essere presentata in blocco e il tempo di generazione cresce direttamente con la lunghezza del testo prodotto.

La quantità di testo che il modello può considerare in un singolo scambio è regolata dalla *finestra di contesto*, misurata in token. In tale finestra rientrano sia l'istruzione dell'utente sia le informazioni che il sistema decide di fornire al modello, per esempio la descrizione dei compiti disponibili o la cronologia della conversazione.

Il testo che viene sottoposto al modello prende il nome di *prompt*. Nelle interfacce conversazionali il prompt è organizzato in due ruoli distinti, il *system* contiene le istruzioni generali che definiscono il comportamento del modello, mentre il ruolo *user* contiene la richiesta dell'utente. La distinzione è rilevante per questa tesi, perché è attraverso il prompt di sistema che si specifica il formato della risposta attesa, senza modificare i parametri del modello. Non tutti i modelli gestiscono i ruoli allo stesso modo, ad esempio Gemma 4 prevede il supporto nativo del ruolo di sistema [14].

2.1.3 Dall'elaborazione del solo testo ai modelli multimodali

In origine l'elaborazione del linguaggio in testo naturale e la visione artificiale sono state sviluppate in percorsi separati, con architetture distinte e addestrate per svolgere un singolo compito, quello di tradurre e comprendere testi, classificare immagini e rilevare oggetti.

La pubblicazione di Shao et al. va a descrivere come si è passati dalle iniziali architetture specializzate ad un'evoluzione verso modelli unificati, i quali sono in grado, con un'unica rete, di affrontare compiti multimodali diversi [8].

Questa transizione è stata resa possibile in primo luogo dall'impiego di encoder visivi pre-addestrati su vaste raccolte di immagini, le quali associate a didascalie, vanno a produrre rappresentazioni generali non più vincolate a un singolo compito [15].

Combinando ciò con la capacità dei modelli linguistici di grandi dimensioni di svolgere il ragionamento sul contesto che viene fornito, è stato possibile ottenere un modello che ricevendo in input un'immagine ed una richiesta restituisce una risposta in linguaggio naturale in uscita [11].

Lo stesso modello è in questo senso capace di ragionare e rispondere ad una domanda posta dall'utente e descrivere un'immagine che gli viene fornita oppure, se specificato nel prompt, produrre una serie di dati in forma testuale. Quest'ultima caratteristica è in particolare la base su cui si appoggia l'architettura descritta in questa tesi.

2.2 Architettura di un VLM

2.2.1 Schema a tre componenti e flusso dei dati

La struttura ricorrente dei VLM si articola in tre componenti [8]:

- un encoder visivo che converte un'immagine in una sequenza di vettori numerici
- un proiettore che trasferisce questi vettori in quello che viene definito lo spazio di rappresentazione per il testo
- un modello linguistico che gestisce il ragionamento multimodale, integrando l'analisi visiva per elaborare la risposta

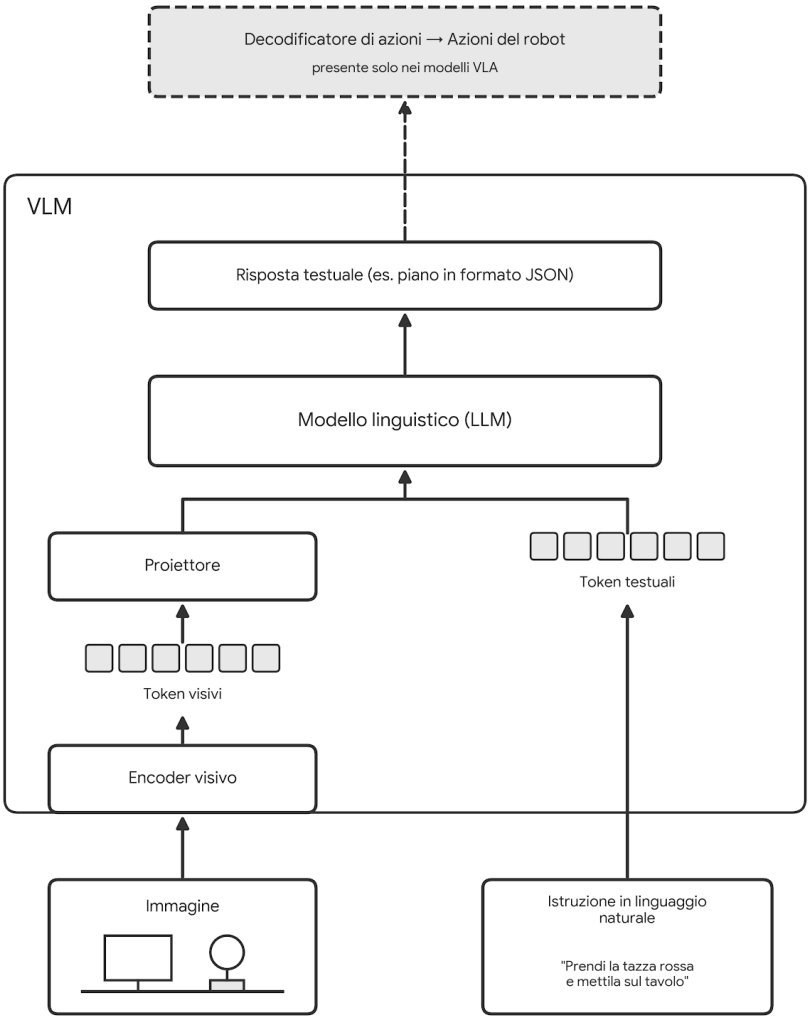


Figura 2.1 – Architettura generale di un modello visione-linguaggio (VLM) ed estensione a Vision-Language-Action (VLA). (iImmagine generata con IA).

La pipeline del VLM si ferma all'output del modello LLM, la successiva integrazione di action decoder per la generazione di token di azioni va a caratterizzare l'architettura di un modello VLA (Vision-Language-Action).

Gli autori formalizzano la prima parte come segue [10]:

l'encoder visivo, indicato con E_{img} riceve in input un'immagine I che traduce in una successione di token visivi v:

$$v = E_{\text{img}}(I;\ \theta_{\text{img}}), \quad (2.1)$$

dove $I \in \mathbb{R}^{H\times W\times 3}$ è l'immagine in ingresso, H e W parametri che rappresentano altezza e larghezza in pixel, 3 indica tre canali di colore.

$v \in \mathbb{R}^{N_v\times D_v}$ è la sequenza di N_v token visivi che hanno dimensione D_v e θ_{img} specifica i parametri dell'encoder.

L'elaborazione congiunta viene espressa dalla formula.

$$h = \text{LLM}(P(v, l);\ \theta_{\text{LLM}}), \quad (2.2)$$

dove P è il proiettore, che va ad allineare le due modalità, $l \in \mathbb{R}^{N_l \times D_l}$ sono token di linguaggio (del testo in ingresso), $h \in \mathbb{R}^{N_h \times D}$ sono gli stati interni che vengono prodotti dal modello, θ_{LLM} rappresenta i parametri dell'LLM.

Gli stati h sono infine convertiti in quelli che sono i token della risposta.

Nelle architetture usate in robotica, a queste due espressioni se ne aggiunge una terza, per quello che nella Figura 2.1 appare come action decoder, che va a tradurre h in comandi per gli attuatori [10]. Questo è il tratto distintivo dei VLA, che verranno discussi successivamente nella Sezione 3.1.

Nel sistema oggetto di questa tesi quel componente non è presente e l'uscita del modello resta in forma testuale.

2.2.2 Encoder visivo

L'encoder visivo si occupa di rendere l'immagine in un oggetto trattabile per un'architettura nata per reagire ad input testuali. A tale scopo l'immagine viene suddivisa in regioni quadrate con dimensione fissa, definite patch, che a loro volta sono disposte in sequenza e convertite in vettori. Le immagini, alla stregua delle informazioni testuali, vengono trasformate in un insieme di token visivi della stessa forma dei token linguistici [10].

La quantità di token visivi che possono essere generati dipende dalla risoluzione dell'immagine e da come viene definita la dimensione della patch, entrambe queste caratteristiche hanno importanti conseguenze sul costo computazionale del processo di analisi.

Gli encoder impiegati per i VLM sono solitamente ripresi da modelli pre-addestrati, in quanto l'addestramento da zero risulta essere troppo dispendioso in termini di dataset e tempo richiesto per l'elaborazione [11].

Yu et al. indicano come scelte più comuni alcune famiglie di Vision Transformer, quali ViT, CLIP, SigLIP e DINOv2, distinguendo due principali modalità di pre-addestramento [10].

- Metodo Contrastivo, il più comune CLIP (Contrastive Language-Image Pre-Training), consiste nella creazione di coppie Immagini-Didascalia, distinte in due diverse categorie.

  Positive: accoppiamento di un'immagine e relativa didascalia che la descrive in maniera veritiera.

  Negative: la stessa immagine viene associata alle didascalie delle altre immagini presenti nel minibatch.

  Il modello viene sottoposto ad un elevato numero di esempi positivi e negativi e impara a produrre rappresentazioni vicine per le coppie (immagine-didascalia) corrispondenti e rappresentazioni distanti per le coppie non corrispondenti [15].

- Modalità Auto-Supervisionata, propria del modello DINOv2, l'addestramento non utilizza immagini con un testo associato, il modello impara a "vedere" e comprendere le immagini attraverso compiti costruiti a partire dalle immagini stesse.

In entrambi i casi il risultato che si ottiene è la capacità del modello di fornire un'elaborazione generale, che si è dimostrata efficace anche su immagini o contenuti che non sono mai stati visionati durante l'addestramento, proprietà indicata come capacità zero-shot [10].

L'aspetto che i vari modelli vanno a trattare in maniera eterogenea è la gestione della risoluzione. Alcuni sistemi tendono a ridimensionare le immagini in un formato fisso, mentre altri adottano invece formati variabili. Gemma 4, ad esempio gestisce la risoluzione andando ad associare a ciascuna immagine un numero di token visivi, configurabile con valori compresi tra 70 e 1120, ottenendo un compromesso tra costo computazionale e livello di dettaglio [14].

2.2.3 Proiettore e spazio di rappresentazione condiviso

I vettori che sono stati prodotti dall'encoder non sono utilizzabili così come sono dall'LLM, questo perché appartengono ad uno spazio di rappresentazione diverso e non è quindi garantito che le dimensioni degli embedding testuali e visivi coincidano.

Il proiettore va a colmare questo divario, generalmente è una piccola rete che trasforma i token visivi in vettori che siano compatibili con quelli presenti nello spazio dell'LLM [10].

Da un lato il proiettore realizza una sorta di allineamento semantico tra le due diverse modalità, unificando all'interno di un solo spazio latente sia l'istruzione linguistica che la percezione visiva.

Tale allineamento stabilisce una corrispondenza diretta tra percezione e comandi, che supera l'ostacolo dell'avere le due capacità di analisi ed elaborazione in moduli separati [8].

D'altra parte il proiettore può anche ridurre la lunghezza della sequenza visiva. In Qwen2.5-VL, ad esempio, una rete a due strati permette di accorpare in gruppi quattro patch adiacenti per proiettarle insieme nella dimensione degli embedding testuali [16].

Poiché il carico computazionale dell'attenzione cresce con il quadrato della lunghezza della sequenza, tale compressione ha effetti netti sui tempi di risposta.

Quando vengono proiettati, i token visivi si concatenano a quelli di testo in una sequenza unica. Shao et al. descrivono l'effetto, a livello di token, di questa integrazione. Essendo elaborate dallo stesso meccanismo, le parole dell'istruzione possono indirizzare l'attenzione del modello verso specifiche porzioni dell'immagine [8].

2.2.4 LLM e generazione della risposta

Il modello linguistico è il centro del ragionamento semantico [10], riceve la sequenza mista di token visivi e di linguaggio, e genera la risposta con la modalità di autoregressione descritta precedentemente nella Sezione 2.1.2.

Generalmente si tratta di un modello linguistico già esistente, che viene utilizzato come base per il VLM. Ad esempio Qwen2.5-VL, inizializza questo componente usando gli stessi pesi del modello linguistico corrispondente [16].

L'effetto più rilevante che ne consegue è che l'uscita del sistema coincide con quella del modello linguistico, quindi con una sequenza di testo, la cui forma può essere regolata dall'utente nel prompt. Risulta quindi possibile richiedere risposte organizzate con una certa struttura prestabilita, come il formato JSON. Il report tecnico di Qwen2.5-VL esplicita proprio la possibilità di richiedere risposte in formato JSON tra le abilità del modello [16]. È proprio su questa possibilità che si basa l'architettura descritta successivamente nel Capitolo 4, i piani d'azione vengono forniti in uscita dal modello in formato JSON in modo che possano essere presi in input ed eseguiti dal sistema robotico.

Va però osservato che il rispetto del formato richiesto non è garantito dal modello, ma è soltanto un comportamento appreso. Si tratta di uno dei limiti discussi nel Sezione 2.4.2.

2.3 Addestramento e instruction tuning

Nel sistema descritto in questa tesi sono stati impiegati modelli già addestrati e resi disponibili dai rispettivi sviluppatori. Le fasi di addestramento vengono qui di seguito richiamate soltanto per cenni, al fine di spiegare in che misura un modello già addestrato sia in grado di seguire le istruzioni indicate all'interno del prompt.

- pre-addestramento: in questa prima fase il modello viene sottoposto a visione di una massiccia quantità di dati. Per la componente visiva, la presentazione di coppie immagine-didascalia permette di addestrare il modello ad elaborare rappresentazioni associate al significato linguistico, come mostrato precedentemente dall'impostazione contrastiva CLIP [15].

  In questa fase il modello non sviluppa la capacità di rispondere alle domande o eseguire le istruzioni fornite dall'utente, ma acquisisce conoscenze generali di base.

- Instruction tuning: il modello viene addestrato su insiemi di esempi, ciascuno dei quali presenta un accoppiamento immagine-istruzione e la risposta attesa, in modo da istruire il modello a reagire a richieste formulate liberamente.

  Shao et al. attribuiscono a questa fase lo sviluppo della capacità di generalizzare, trovando comunque il modo di reagire, a situazioni non incontrate durante l'addestramento. Gli autori riconoscono a LLaVA il merito di avere introdotto tale procedimento, con la costruzione di circa 150 000 esempi conversazionali partendo da coppie immagini-testo esistenti [8], [17].

Inoltre si è analizzato che ad incidere sul comportamento finale del modello è anche la qualità dei dati che vengono forniti, gli autori citano in questo senso l'adozione di misure di filtraggio automatiche per annotazioni rumorose come mezzo per ridurre le risposte considerate anomale [8].

La distinzione tra queste due fasi implica una conseguente distinzione pratica nei modelli che vengono distribuiti pubblicamente, in base al tipo di addestramento a cui sono stati sottoposti. Gli sviluppatori di norma rilasciano sia la versione che si è ottenuta solo mediante pre-addestramento sia quella che è stata successivamente sottoposta ad instruction tuning.

La model card di Gemma 4 distingue esplicitamente le due varianti [14], e la sigla it nell'identificativo gemma-4-e2b-it indica appunto il modello sottoposto ad instruction tuning. Solo questa variante è adatta ad un sistema che deve interagire con l'utente in linguaggio naturale.

Un modello già addestrato può essere ulteriormente adattato a un compito specifico, ciò avviene tramite particolari tecniche che vanno ad aggiornare tutti i parametri oppure, come nel caso di LoRA, si può andare ad aggiornare il modello con un numero ridotto di parametri aggiuntivi [18]. Questa è di fatto la strada che viene seguita dai sistemi robotici che apprendono direttamente le azioni, descritti in seguito nel Sezione 3.1.

Nel presente lavoro non si è fatto ricorso ad alcuna tecnica di aggiornamento, il comportamento del modello viene regolato unicamente dalle istruzioni fornite nel prompt di sistema, secondo le specifiche che vengono descritte nel Capitolo 4.

2.4 Capacità e limiti rilevanti per la robotica

2.4.1 Capacità

Si vanno di seguito ad analizzare le quattro capacità che vale la pena sottolineare quando si va ad utilizzare un VLM in un sistema robotico.

La prima è la comprensione di istruzioni libere. Il modello non ha necessità di ricevere comandi che rispettano una certa struttura o sintassi stabilita a priori, ma è in grado di interpretare testo in linguaggio naturale in base al contesto. Shao et al. attribuiscono tale proprietà alla conoscenza che viene acquisita nella fase di pre-addestramento e la contrappongono ai sistemi tradizionali, i quali hanno la tendenza a ricondurre le istruzioni a schemi fissi [8]. Per un robot che è destinato all'assistenza domestica si tratta quindi di un requisito essenziale, in quanto l'utente non ha la necessità di dover conoscere comandi o avere preconoscenze tecniche per andare ad utilizzare il sistema.

La seconda è la descrizione della scena. Il modello comprende l'immagine acquisita dalla telecamera ed è in grado di elencare gli oggetti che appaiono nell'immagine, in questo senso ha quindi la possibilità di rispondere a domande sul suo contenuto [11].

La terza è il riferimento a oggetti determinati. I modelli recenti non si limitano alla sola descrizione della scena che viene loro presentata, collegano le istruzioni a specifiche regioni dell'immagine. Il report tecnico di Qwen2.5-VL riporta, tra le capacità del modello, la possibilità di localizzare tramite coordinate riferite all'immagine la posizione degli oggetti nella scena [16].

La quarta, già richiamata nella Sezione 2.2.4, riguarda la produzione di un output strutturato, molto importante in quanto consente di indirizzare la risposta del modello come input per un altro programma anziché come testo destinato alla sola lettura [16].

2.4.2 Limiti

Queste capacità sono tuttavia affiancate da limiti altrettanto rilevanti, che vanno ad incidere sulla progettazione dell'intero sistema e saranno successivamente ripresi nel Capitolo 5.

Una prima osservazione deve essere fatta riguardo l'affidabilità del contenuto. La model card di Gemma 4 riconduce le prestazioni del modello a diversi fattori, quali la qualità e la copertura dei dati di addestramento, la complessità del compito, l'ambiguità del linguaggio e il contesto che viene fornito. Tra i limiti viene inoltre indicata l'accuratezza dei fatti riportati e il senso comune [14]. Un modello potrebbe quindi produrre una risposta che nella forma è plausibile, ma errata nel contenuto, riportando ad esempio la presenza di oggetti non presenti nell'immagine.

Un altro aspetto da tenere presente è la capacità di attenersi alla forma della risposta. Il rispetto del formato prestabilito non dipende da una particolare architettura, ma è un'abilità che il modello apprende nella fase di addestramento. In questo senso non è possibile avere garanzie dal punto di vista progettuale. In seguito a ciò è necessario andare sempre a validare la risposta del modello se si intende utilizzarla come dato di ingresso. L'assenza di tale controllo rappresenta quindi uno dei limiti dell'implementazione descritta in questa tesi.

Il terzo limite è la sensibilità al prompt. Bordes et al. osservano una tendenza frequente dei modelli a trascurare parte delle istruzioni che gli vengono assegnate. Diventa in questo senso necessario un consistente lavoro di ingegneria per quanto riguarda la messa a punto di un prompt che risulti efficace ad ottenere il lavoro richiesto [11]. Gli stessi autori segnalano difficoltà ricorrenti per quanto riguarda

la comprensione delle relazioni spaziali, il conteggio degli oggetti ed il riconoscimento di attributi [11].

Un ulteriore fattore da non trascurare è l'entità del carico computazionale, che si traduce in latenza. Yu et al. indicano nelle dimensioni dell'LLM quella che è la principale causa di inefficienza di questi sistemi, inoltre a rendere il problema più marcato è la grandezza delle sequenze di token visivi [10]. Nell'architettura descritta in questa tesi la latenza incide solo sul tempo di risposta percepito dall'utente, poiché la pianificazione è separata dall'esecuzione.

2.5 Modelli aperti ed esecuzione in locale

2.5.1 Motivazioni e requisiti dell'esecuzione locale

I modelli visione-linguaggio più noti sono accessibili come servizi remoti, attraverso interfacce di programmazione che vengono messe a disposizione dagli sviluppatori. In un sistema robotico assistivo, tale soluzione presenta alcuni inconvenienti, richiamati dagli autori Yu et al. che trattano le applicazioni di questi sistemi in impieghi domestici e sanitari [10]. Il primo aspetto da considerare è la riservatezza dei dati dell'utente, le immagini acquisite all'interno di un'abitazione e in generale le informazioni relative allo stato di salute della persona assistita, verrebbero trasmessi ad un servizio esterno, mentre l'elaborazione locale li mantiene sul dispositivo. Una seconda considerazione da fare è la dipendenza dalla connessione di rete, che rende il sistema inutilizzabile in caso di malfunzionamenti o assenza di essa. Infine la latenza introdotta dalla comunicazione di rete, che si va a sommare a quella del modello.

L'esecuzione locale d'altra parte comporta un certo vincolo di risorse. Gli stessi autori esplicitano che i modelli con parametri che rientrano nell'ordine dei miliardi presentano latenze elevate derivanti dalla massiva entità del carico computazionale. Nella rassegna, un modello da 7 miliardi di parametri viene indicato con un tempo di inferenza di circa 166 ms, mentre una variante da 55 miliardi può arrivare ad avere valori compresi tra 330 e 1000 ms [10].

Si tratta comunque di misure riferite a modelli VLA e all'hardware impiegato dagli autori, riportate qui soltanto come ordine di grandezza.

In letteratura si descrivono due strategie che si possono impiegare allo scopo di rendere sostenibile questo costo. La prima consiste nell'adottare dei modelli che abbiano dimensioni contenute, Yu et al. osservano che di recente molti sistemi diversi vengono strutturati a partire da VLM con pochi miliardi di parametri, tra i quali si indicano alcune varianti della famiglia Qwen [10].

La seconda strategia è impiegare quella che viene definita tecnica di quantizzazione. Consiste nell'andare a rappresentare i parametri del modello con un numero di bit ridotto, in modo tale da ridurre la memoria occupata, con conseguente ottimizzazione dei tempi di inferenza. Gli autori riportano un caso di quantizzazione a 4 bit che viene applicata in seguito all'addestramento, si ottiene un dimezzamento della memoria richiesta mantenendo prestazioni paragonabili con il modello iniziale [10].

Anche nella documentazione di Gemma 4 è previsto l'impiego di modelli a precisione predefinita 16 bit oppure a precisione ridotta tramite quantizzazione [14].

Nel presente lavoro la quantizzazione non è stata oggetto di studio e viene qui richiamata solo per completezza. I modelli sono stati impiegati nelle versioni rese disponibili dall'ambiente di esecuzione descritto successivamente nella Sezione 2.5.4.

Un'ultima condizione è invece di natura giuridica, l'esecuzione locale richiede modelli scaricabili e utilizzabili secondo i termini specifici di particolari licenze. I due modelli impiegati sono entrambi a pesi aperti e rientrano quindi in questa categoria, si procede a descriverli nei paragrafi seguenti.

2.5.2 Qwen2.5-VL

Qwen2.5-VL è una famiglia di VLM sviluppata da Alibaba e documentata da un report tecnico [16]. Questo modello è disponibile in tre versioni di dimensioni variabili, con 3, 7 e 72 miliardi di parametri, destinate a contesti che vanno dai dispositivi con risorse limitate a quelli che sono sistemi di calcolo ad alte prestazioni.

L'architettura segue lo schema precedentemente esposto nella Sezione 2.2, ovvero un modello linguistico, un encoder visivo di tipo Vision Transformer e un modulo di fusione basato su percettore multistrato [16]. Nel progetto viene impiegata la versione da 7 miliardi di parametri, che ha un encoder visivo composto da 32 strati e dimensione interna 1280, mentre l'LLM ha 28 strati e dimensione interna 3584 [16]. Al fine di mantenere sostenibile il costo dell'elaborazione visiva, ad applicare l'attenzione all'intera sequenza sono solo quattro strati dell'encoder, gli altri invece operano su finestre locali, così il costo cresce linearmente con il numero di patch [16].

Si sottolineano due caratteristiche proprie dell'architettura di questo modello. La prima è il già citato raggruppamento di quattro patch adiacenti a opera del modulo di fusione, che va a ridurre la lunghezza della sequenza visiva [16]. La seconda è il supporto esplicito alla produzione di risposte in formato JSON, il quale è un requisito alla base dell'architettura che si descrive nel Capitolo 4 [16].

La versione da 7 miliardi di parametri adattata a seguire istruzioni è distribuita con licenza Apache 2.0 [19].

2.5.3 Gemma 4

Gemma 4 è una famiglia di modelli a pesi aperti sviluppata da Google DeepMind e descritta in un report tecnico [20] e in una model card [14]. I modelli sono nativamente multimodali, possono ricevere in input testo e immagini per dare in uscita una risposta in linguaggio naturale. Alcune versioni possono anche ricevere in ingresso file audio [14]. La famiglia comprende sia architetture dense sia architetture di tipo *mixture-of-experts*, con un numero di parametri tra 2,3 e 31 miliardi [20].

Nel sistema oggetto di questa tesi è stata impiegata la versione E2B, adatta a seguire istruzioni. In base a quanto riportato nella model card dispone di 2,3 miliardi di parametri effettivi, che considerando anche gli embedding, diventano 5,1 miliardi. L'architettura è organizzata in 35 strati, possiede una finestra di contesto la cui capacità può arrivare fino a 128 000 token ed un encoder visivo di circa 150 milioni di parametri [14]. La distinzione tra parametri effettivi e parametri complessivi nasce in seguito all'impiego delle tabelle di embedding che sono associate ai singoli strati, esse sono consultate durante l'elaborazione, ma non attive come gli altri parametri. Questa caratteristica è il motivo della lettera *E* nel nome [14].

Tra le caratteristiche peculiari del modello vediamo inoltre la possibilità di gestire la risoluzione di un'immagine tramite un numero regolabile di token visivi, che varia tra 70 e 1120. Gemma 4 prevede inoltre il supporto nativo del ruolo di sistema, il che consente una separazione tra le istruzioni generali e la richiesta dell'utente [14].

Anche questa famiglia di modelli è distribuita con licenza Apache 2.0 [14].

Sarebbe interessante poter operare un confronto diretto tra i due modelli, ma sulla base dei dati dichiarati dai rispettivi sviluppatori, questo non è possibile. I risultati sono stati ottenuti in momenti diversi e con molta probabilità in condizioni non uguali. Un confronto significativo richiede misure che siano raccolte sullo stesso sistema e sugli stessi compiti.

2.5.4 Esecuzione tramite server locale con API compatibile OpenAI

Al fine di poter utilizzare modelli a pesi aperti, oltre a dover disporre dei parametri, è necessario che un programma si occupi del caricamento in memoria, che ne gestisca l'elaborazione e si occupi di rendere i risultati disponibili alle applicazioni. Il modello può in questo modo essere usato sulla macchina dell'utente e le sue funzioni vengono presentate attraverso un'interfaccia di programmazione locale, accessibile su una porta di rete della macchina.

Questa interfaccia di programmazione generalmente risulta essere compatibile con quella resa nota da OpenAI, la richiesta contiene i messaggi suddivisi per ruolo ed eventuali immagini, mentre la risposta ha la struttura attesa dai client già definiti per un determinato servizio. Dal punto di vista pratico, si ha il vantaggio che un'applicazione ha la possibilità di passare da un modello remoto ad uno locale, e viceversa, tramite la sola modifica dell'indirizzo del servizio e del nome del modello.

Durante l'implementazione di questo sistema è stato impiegato LM Studio, che offre la possibilità di eseguire i modelli sulla macchina Windows ed espone l'interfaccia locale utilizzata dal backend del sistema [21]. La configurazione delle porte ed il formato delle richieste verranno successivamente descritti nel Capitolo 4.

2.6 Sintesi del capitolo

Il capitolo ha ricostruito che cosa sia un modello visione-linguaggio e quali proprietà ne giustifichino l'impiego in un sistema robotico assistivo. Si è visto che tali modelli derivano dai modelli linguistici di grandi dimensioni, ai quali affiancano un percorso di elaborazione visiva, e che la loro architettura si articola in un encoder visivo, un proiettore e un modello linguistico. Poiché la risposta è sempre testo e la sua forma può essere specificata nel prompt, un VLM può essere utilizzato non solo per descrivere ciò che osserva, ma anche per produrre strutture dati destinate a un altro programma.

Accanto a queste capacità sono stati indicati i limiti che ne condizionano l'uso, ovvero l'assenza di garanzie sulla correttezza del contenuto e sulla forma della risposta, la sensibilità al prompt e il costo computazionale. Proprio quest'ultimo, insieme alle esigenze di riservatezza tipiche dell'ambito assistivo, motiva la scelta di modelli a pesi aperti eseguiti su una macchina locale, quali Qwen2.5-VL e Gemma 4.

Resta però aperta la questione principale: un modello di questo tipo produce testo, mentre un robot richiede comandi. Il capitolo successivo si apre su questo divario e presenta le tecnologie impiegate per colmarlo.

3 Tecnologie per il controllo robotico

Il Capitolo 2 si è chiuso su una constatazione: un VLM produce testo, mentre un robot ha bisogno di comandi. Questo capitolo presenta gli strumenti che colmano tale distanza. La Sezione 3.1 inquadra il problema e colloca HIMA rispetto agli approcci presenti in letteratura, i paragrafi successivi descrivono il middleware ROS 2 (3.2), lo stack di navigazione (3.3) e gli ambienti di simulazione e le piattaforme robotiche impiegate (3.4). L'implementazione dei componenti di HIMA è trattata nel Capitolo 4.

3.1 Dal linguaggio all'azione

3.1.1 Il divario tra piano e comandi

Come osservato al termine della Sezione 2.6, un VLM produce testo, mentre un robot richiede comandi. La richiesta di un utente in linguaggio naturale, dal punto di vista semantico e strutturale, non può quindi essere passata direttamente come comando per un movimento robotico.

Un'istruzione come "vai in bagno e controlla che non ci sia nessuno" descrive lo scopo da raggiungere, ma non i passi per raggiungerlo. Per essere eseguita, deve essere scomposta in più passaggi ordinati che indicano azioni (spostarsi, osservare, riferire). A ciascun passo corrisponde una delle capacità del robot, che viene infine convertita in comandi elementari per gli attuatori, come la velocità lineare o angolare della base mobile.

Alla luce di ciò si può affermare che la richiesta dell'utente e il movimento del robot si trovano su livelli di astrazione diversi. Tra i due estremi si possono distinguere quattro livelli:

- istruzione: la richiesta in linguaggio naturale
- piano simbolico: sequenza di passi consecutivi, ciascuno con un nome e dei parametri (per esempio «naviga verso: bagno»)
- primitive: capacità del robot che possono essere eseguite in maniera autonoma, come afferrare un oggetto o raggiungere una posa
- comandi di controllo: segnali trasmessi agli attuatori in modo continuo e ad alta frequenza

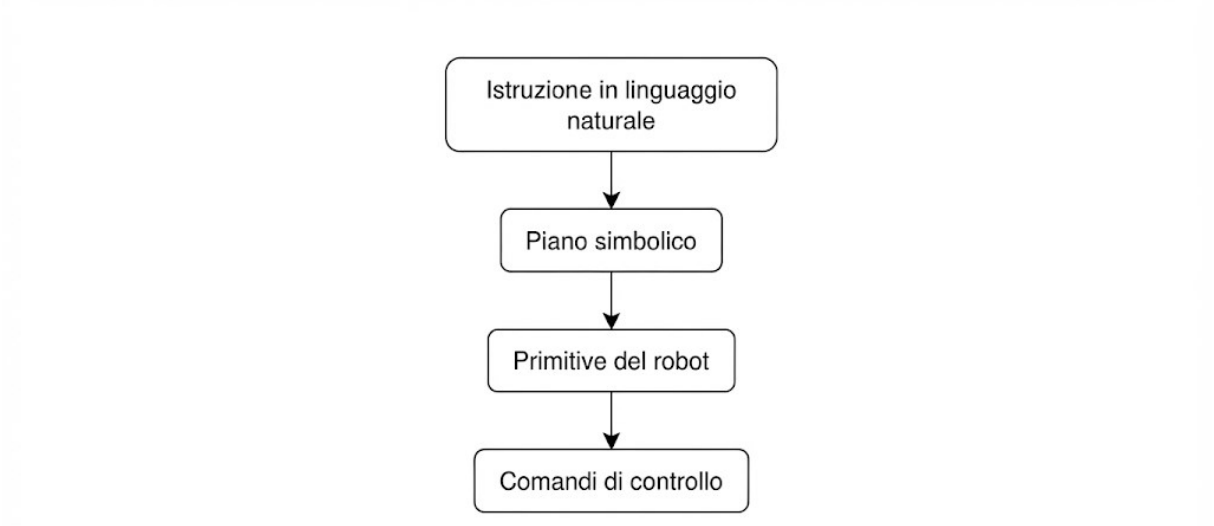


Figura 3.1 – Livelli di astrazione tra linguaggio e azione (immagine generata con IA)

La conversione della richiesta dell'utente in comandi di controllo presenta tre aspetti problematici.

Il primo è il radicamento (*grounding*), le parole devono essere associate a entità fisiche, per esempio, la parola "bagno" deve corrispondere a una posizione all'interno della mappa del robot.

Il secondo riguarda la scala temporale, ovvero i tempi di inferenza di un modello di grandi dimensioni non sempre sono compatibili con le alte frequenze richieste dal controllo [8, §3.1.3].

L'ultimo aspetto è la retroazione. Durante l'esecuzione dei comandi l'ambiente può cambiare, si possono incontrare ostacoli non previsti in fase di pianificazione e le azioni possono fallire. È quindi necessario un meccanismo in grado di osservare gli esiti delle azioni e di reagire a questi stimoli esterni.

In letteratura si individuano principalmente due tipi di approccio. Negli approcci end-to-end un unico modello apprende come produrre direttamente le azioni, Shao et al. definiscono questi modelli monolitici [8]. Negli approcci gerarchici, invece, un pianificatore genera una rappresentazione intermedia, che viene passata a un livello inferiore incaricato di tradurla in comandi [8]. HIMA appartiene a questa seconda famiglia, con le caratteristiche discusse in 3.1.4.

3.1.2 Approcci end-to-end: i modelli VLA

I modelli Vision-Language-Action (VLA) possono essere considerati un'estensione dei VLM, in quanto non si limitano a produrre testo in linguaggio naturale, ma sono in grado di generare direttamente azioni [10].

L'architettura di partenza è quella del VLM descritta nella Sezione 2.2, l'encoder visivo converte le immagini in token, il proiettore li porta nello spazio del backbone linguistico e il backbone elabora congiuntamente sia i token visivi sia quelli testuali provenienti dalle istruzioni. A questi elementi si aggiunge un modulo che amplia le capacità del VLM ed è proprio ciò che contraddistingue i VLA, come anticipato nella Sezione 2.2.1, il decodificatore di azioni. Nella rassegna di Yu et al. [10, §2.1.1] questo modulo riceve gli stati nascosti h del backbone e, elaborandoli, restituisce in uscita una sequenza di azioni, secondo l'espressione:

$$a_{1:T} = D_{act}(h;\, \theta_{act})$$

dove $a_{1:T}$ rappresenta un blocco di azioni relative a T istanti successivi e θ_{act} indica i parametri del decodificatore. Le azioni sono grandezze continue e direttamente eseguibili, possono essere pose dell'organo terminale da raggiungere oppure comandi della pinza. Il decodificatore può essere realizzato con una rete MLP, con una generazione autoregressiva di token d'azione oppure con modelli generativi basati su diffusione o *flow matching* [10].

Nella rassegna di Shao et al. [8, §3] sono definiti monolitici i modelli in cui percezione, ragionamento e generazione delle azioni sono integrati.

A loro volta, i modelli monolitici possono essere a singolo modulo, in cui un unico modello elabora tutte le modalità e decodifica le azioni, oppure duali. Questi ultimi suddividono il carico di lavoro, un VLM di grandi dimensioni si occupa del ragionamento, mentre un modello più leggero è dedicato alla sola generazione di azioni a frequenza elevata. Questa separazione nasce dall'esigenza di conciliare il ragionamento con tempi di generazione dei comandi di movimento compatibili con le alte frequenze di controllo [8, §3.2].

Per costruire la conoscenza alla base di questi modelli sono richiesti grandi dataset contenenti traiettorie robotiche, reali o simulate, da cui i modelli apprendono il comportamento da tenere e le azioni da eseguire in diversi contesti [10, §2.1.2].

Per adattare questi modelli a un nuovo robot o a un nuovo compito si ricorre al fine-tuning, già discusso nella Sezione 2.3, spesso con tecniche che aggiornano solo una parte mirata dei parametri, come LoRA [10, §4], [18].

Il principale vantaggio dell'approccio end-to-end è che l'intera catena, dalla percezione al controllo, viene ottimizzata congiuntamente, senza interfacce progettate a mano.

Tuttavia, questo approccio comporta anche dei limiti da considerare per un progetto come HIMA:

- le rappresentazioni che passano tra i livelli intermedi restano confinate nello spazio latente e sono quindi difficili da ispezionare [8, §4.3]
- servono raccolte di dati di dimostrazione specifiche per il robot e risorse di calcolo per l'addestramento
- il costo dell'inferenza a ogni passo di controllo è elevato, e gran parte della ricerca si concentra sulle tecniche per ridurlo [8, §3.1.3], [10].

3.1.3 Approcci gerarchici

La modellazione gerarchica costituisce un paradigma fondamentale nei modelli VLA basati su VLM di grandi dimensioni, ed è particolarmente utile negli scenari che richiedono ragionamento a lungo termine, astrazione spaziale o decomposizione delle azioni [8, §4].

Il principio alla base consiste nel separare in due moduli il ragionamento, affidato al VLM che svolge il ruolo di pianificatore di alto livello, e la generazione delle azioni, per la quale si impiega una politica di basso livello.

Il flusso di dati segue uno schema in cui l'interpretazione delle istruzioni e delle osservazioni spetta unicamente al pianificatore, che le converte in rappresentazioni intermedie. La politica riceve queste rappresentazioni e, a partire da esse, genera una sequenza di azioni direttamente eseguibili dal robot.

Questa suddivisione permette ai due moduli di operare su piani separati, con completa indipendenza nel proprio campo d'azione. L'architettura risulta quindi flessibile e consente diverse combinazioni dei due moduli.

In questo senso i modelli gerarchici possono essere suddivisi in due categorie, i modelli con solo pianificatore, che si concentrano sulla realizzazione del pianificatore e utilizzano un insieme di capacità già disponibili. I modelli pianificatore + politica, che sviluppano e integrano entrambi i moduli [8, §4.1–4.2].

La peculiarità dei modelli gerarchici è che le rappresentazioni intermedie non restano più confinate nello spazio latente, interne al modello, ma sono consultabili e interpretabili anche da una persona. Il piano può quindi essere letto, controllato e, se necessario, modificato prima dell'esecuzione. Il livello inferiore può essere costituito da una pipeline robotica tradizionale [8, §4.3].

3.1.4 L'approccio di HIMA e le sue conseguenze

HIMA adotta un'architettura gerarchica all'interno della quale il VLM svolge il ruolo di pianificatore. Nella classificazione di Shao et al. [8] il sistema può essere ricondotto ai modelli con solo

pianificatore, in quanto l'esecuzione è affidata a componenti esterni già esistenti, senza l'impiego di una politica appresa. Il VLM utilizzato non integra un decodificatore di azioni e non viene riaddestrato, ma come anticipato nella Sezione 2.3, il suo comportamento è regolato unicamente dalle specifiche del prompt di sistema.

La rappresentazione intermedia è un piano d'azione in formato JSON. Ogni azione appartiene a un vocabolario chiuso e riporta i propri parametri. Il piano viene inviato al livello fisico, dove un nodo ROS 2 ne gestisce l'esecuzione in modo sequenziale e affida la navigazione allo stack Nav2 (Sezione 3.3).

Il funzionamento e le connessioni dei vari componenti sono illustrati nel Capitolo 4.

Figura 3.2 – Posizione del VLM nella catena di HIMA (immagine generata con IA)

Questa struttura comporta tre conseguenze principali.

Separazione delle scale temporali**.** Il VLM viene interrogato solo quando deve essere prodotto un piano, su richiesta dell'utente o nella modalità autonoma, e resta quindi escluso dai cicli di controllo. Durante l'esecuzione del piano il controllo del moto avviene interamente all'interno di Nav2, con una propria frequenza. Come discusso nella Sezione 2.4.2, la latenza del VLM incide quindi solo sul tempo che intercorre tra la richiesta e l'inizio del movimento, senza ripercussioni sulla stabilità del controllo.

Leggibilità del piano**.** Il piano JSON può essere registrato, consultato e controllato prima dell'esecuzione, coerentemente con i vantaggi dei modelli gerarchici descritti in 3.1.3. Nella versione attuale, tuttavia, il piano non è sottoposto a una validazione dello schema (Capitolo 4).

Dipendenza dall'interfaccia. Le capacità del sistema dipendono dal vocabolario di azioni eseguibili, e la correttezza del piano dipende dal rispetto del formato da parte del modello, cioè da quanto il modello segue le specifiche del prompt. L'efficacia del prompt è quindi un requisito necessario. Inoltre, l'esito dell'esecuzione del piano non viene restituito al pianificatore, questo limite è discusso nei Capitoli 4 e 5.

3.2 ROS 2

Il livello fisico di HIMA è costruito su ROS 2, il framework open source più diffuso per lo sviluppo di software robotico [22]. Questo paragrafo ne introduce l'organizzazione e i meccanismi di comunicazione che il sistema utilizza: nodi e topic, trasformazioni tra sistemi di riferimento, servizi e action ed esecuzione asincrona.

3.2.1 Da ROS a ROS 2: il middleware

ROS non si comporta come un classico sistema operativo, in quanto non gestisce direttamente processi e risorse, ma offre delle vie di comunicazione e permette quindi di creare una rete di lavoro tra più componenti, appoggiandosi al sistema operativo di uno o più calcolatori [23]. In questo senso

ROS permette di scomporre il software di un robot in componenti indipendenti e riutilizzabili, sviluppati da gruppi diversi e in grado di scambiarsi informazioni attraverso interfacce comuni.

La prima versione, presentata nel 2009 [23], si è affermata soprattutto nella ricerca. Il suo impiego in prodotti commerciali ne ha fatto emergere i limiti, tra cui la presenza di un nodo centrale necessario per mettere in contatto gli altri componenti, lo scarso supporto per reti poco affidabili e per i sistemi con vincoli temporali, oltre una gestione insufficiente della sicurezza informatica. Allo scopo di superarli è stata sviluppata la versione ROS 2 [22].

Questo nuovo software ha come principale aggiornamento la configurazione del middleware. ROS 2 si basa su DDS (*Data Distribution Service*), uno standard industriale per lo scambio di dati tra processi distribuiti, disponibile in più implementazioni [24]. Questo servizio si occupa della scoperta reciproca dei componenti, della serializzazione dei messaggi e del trasporto in rete.

Tramite DDS non è più necessario un nodo centrale che si occupi dell'instradamento dei dati tra i componenti, ogni nodo si connette direttamente con gli altri e il guasto di un componente non isola i restanti.

Il software ROS 2 è organizzato a strati. Il codice dell'utente si appoggia a una libreria client, rclcpp per C++ o rclpy per Python. Entrambe si basano su una libreria comune scritta in C, che comunica con il middleware attraverso un'interfaccia astratta (*ROS Middleware Interface*, RMW) [25]. Il vantaggio di questa architettura sta nella possibilità di sostituire l'implementazione DDS senza modificare il codice applicativo, per le distribuzioni binarie quella predefinita è Fast DDS [24].

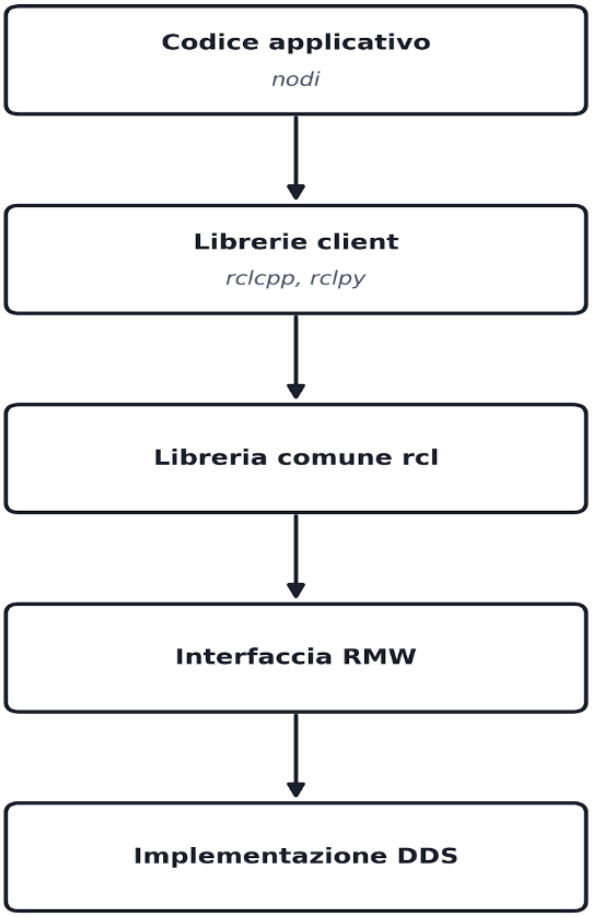


Figura 3.3 – Organizzazione a strati di ROS 2 (immagine generata con IA)

DDS mette a disposizione anche la qualità del servizio (*Quality of Service*, QoS). Ogni canale di comunicazione può essere configurato con un profilo che stabilisce, tra l'altro [26]:

- la quantità di messaggi che devono essere conservati in coda (*history* e profondità)
- se la consegna deve essere confermata da un messaggio di *acknowledgement*, con eventuali ritrasmissioni, oppure tentata una sola volta (*reliability*)
- se i messaggi inviati devono restare disponibili per i componenti che si collegano in seguito alla loro trasmissione (*durability*).

Produttore e consumatore di dati possono collegarsi solo se c'è compatibilità tra i rispettivi profili [26]. La scelta del profilo dipende dal tipo di dato, in un flusso di misure frequenti la perdita di un dato è accettabile, mentre una mappa pubblicata una sola volta deve raggiungere tutti i componenti, compresi quelli che si collegano dopo.

ROS 2 viene rilasciato con diverse distribuzioni. HIMA utilizza Humble Hawksbill nelle configurazioni dei test, distribuzione con supporto a lungo termine rilasciata nel maggio 2022 e supportata fino a maggio 2027 [27], installata su Ubuntu 22.04 in WSL2. Le prove con Stretch in MuJoCo sono state condotte con Jazzy Jalisco, rilasciata nel maggio 2024 [27]. Non tutti i componenti di HIMA sono nodi ROS 2, il backend, eseguito su Windows, comunica con il livello fisico tramite una connessione TCP, descritta nel Capitolo 4.

3.2.2 Nodi, topic e messaggi

L'unità che sta alla base di un'applicazione ROS 2 è il nodo, un componente che svolge una funzione ben definita, come l'acquisizione di dati o il calcolo di un percorso [28]. I nodi comunicano nello stesso modo indipendentemente dalla loro collocazione, sia che si trovino nello stesso processo, in processi diversi o su calcolatori diversi. L'insieme dei nodi e delle loro connessioni di comunicazione costituisce il grafo ROS. Il comportamento di ogni nodo viene regolato da una serie di parametri, modificabili durante l'esecuzione [28].

Tra i canali di comunicazione, quello più comunemente usato è il topic, che si basa sul modello publisher/subscriber [29]. L'identificativo di un topic è il nome e attraverso di esso un nodo pubblica dati su uno specifico topic (*publisher*) e un altro vi si abbona per raccoglierli (*subscriber*). Ogni messaggio raggiunge tutti i nodi abbonati, senza che questi conoscano gli altri abbonati né chi pubblica.

Questo disaccoppiamento, tra le entità che producono e quelle che consumano i dati, comporta due effetti. Un componente può essere sostituito da un altro equivalente, come un sensore simulato al posto di uno reale, senza che il resto del sistema debba essere modificato. Inoltre è possibile inserire nodi con il solo scopo di registrare dati, senza che questi partecipino alla catena di scambio e mantenendo inalterato il flusso tra i nodi esistenti [29].

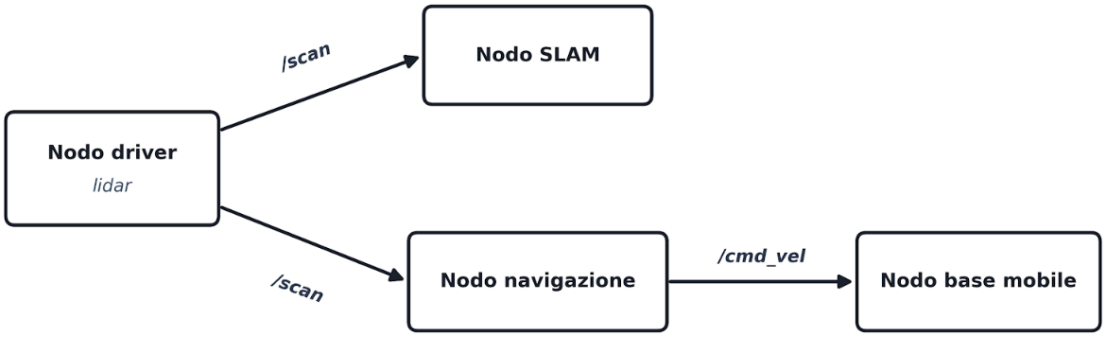


Figura 3.4 – Esempio di grafo ROS con nodi connessi su due topic diversi (immagine generata con IA)

I dati che vengono scambiati sui topic sono dei messaggi con un formato che ne definisce la struttura e i campi. Ogni tipo è descritto in un'interfaccia contenente i campi e il loro tipo, e il rispetto di questi viene controllato dal sistema [29]. ROS 2 fornisce un insieme di tipi standard [30], tra cui quelli più rilevanti per la navigazione:

Tipo di messaggio	Contenuto
`sensor_msgs/LaserScan`	una scansione di un sensore laser (lidar)
`sensor_msgs/Image`	un'immagine di una telecamera
`nav_msgs/Odometry`	posa e velocità stimate dall'odometria
`nav_msgs/OccupancyGrid`	una mappa a griglia di occupazione
`geometry_msgs/Twist`	velocità lineare e angolare, usata per i comandi di moto
`geometry_msgs/PoseStamped`	una posa con riferimento temporale e sistema di riferimento

Tabella 3.1 – Tipi di messaggio standard usati nella navigazione

3.2.3 Sistemi di riferimento e TF2

Le grandezze geometriche che indicano la posizione di un robot sono tutte espresse rispetto a un sistema di riferimento (*frame*). Il lidar misura le distanze nel proprio frame, la base si muove rispetto all'ambiente e la mappa viene considerata con un'origine fissa. Per stabilire la posizione di un ostacolo rilevato dal sensore occorre quindi comporre più trasformazioni tra i vari sistemi di riferimento, alcune fisse e altre variabili nel tempo. Nelle prime versioni di ROS la gestione delle trasformazioni era una frequente fonte di errori ed è stata perciò affidata a una libreria dedicata, tf [31].

Con lo sviluppo di ROS 2 la libreria è stata aggiornata e questo compito viene affidato a tf2 [32]. La libreria gestisce le relazioni tra i diversi frame all'interno di una struttura ad albero, di cui conserva la storia recente. Le trasformazioni di punti, vettori e pose si possono quindi calcolare non solo nell'istante corrente, ma anche riferendosi a un istante passato, per esempio quello in cui è stata acquisita una misura. Le relazioni che restano invariate, come quella di un sensore solidale alla base mobile, vengono pubblicate come trasformazioni statiche [32].

Per i robot mobili la convenzione REP 105 definisce tre frame principali [33]:

Frame	Significato	Proprietà
`base_link`	solidale alla base del robot	si muove con il robot
`odom`	fisso nell'ambiente, stimato dall'odometria	posa continua, ma soggetta a deriva nel tempo
`map`	fisso nell'ambiente, legato alla mappa	posa senza deriva significativa, ma con possibili salti discreti

Tabella 3.2 – Frame definiti dalla convenzione REP 105

Le due stime della posa indicate nella tabella sono tra loro complementari. Il frame `odom` si utilizza per il controllo locale ed è affidabile nel breve periodo, il frame `map` è il riferimento corretto nel lungo periodo, ma la posa può cambiare a scatti quando la localizzazione la ricalcola dalle misure dei sensori.

Ogni frame ha un solo genitore, quindi i frame sono concatenati tra loro, `map` è genitore di `odom`, che a sua volta è genitore di `base_link` [33]. La trasformazione da `odom` a `base_link` è pubblicata dalla sorgente di odometria. La localizzazione stima la posa del robot nella mappa, ma pubblica la correzione da `map` a `odom`, così che l'albero resti coerente [33].

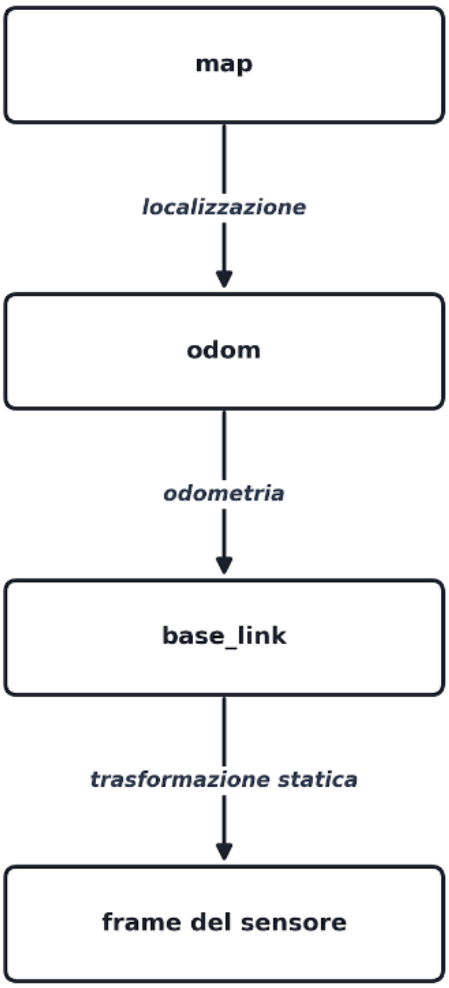


Figura 3.5 – Catena dei frame di un robot mobile (immagine generata con IA)

L'albero dei frame è un prerequisito della navigazione, l'obiettivo che viene assegnato è una posa espressa in un frame, di solito map, e lo stack di navigazione deve poterla mettere in relazione con la posizione del robot e con le misure dei sensori. Se l'albero è incompleto, queste trasformazioni non possono essere calcolate e di conseguenza la navigazione non è possibile.

3.2.4 Servizi e action

I topic trasmettono dati senza attendere risposta. Quando un nodo deve chiedere a un altro di svolgere un'operazione e riceverne l'esito, ROS 2 mette a disposizione due meccanismi di tipo richiesta/risposta: i servizi e le action.

Un servizio realizza una chiamata di procedura remota [34]. I nodi che utilizzano questa modalità di comunicazione si dividono in due ruoli, ovvero un nodo *client* invia la richiesta, invece un nodo *server* esegue l'operazione e restituisce la risposta. Per ogni nome di servizio esiste un solo server, mentre i nodi client possono essere molteplici. I servizi sono progettati per operazioni brevi, in quanto dopo la chiamata il client resta in attesa del risultato e non può interrompere l'operazione [34].

Un'action è anch'essa una chiamata di procedura remota, ma adatta a operazioni di lunga durata [35]. Il funzionamento è simile a quello del servizio, con due caratteristiche aggiuntive, il server invia al client feedback periodici sull'avanzamento e il client ha la possibilità di annullare l'obiettivo o di sostituirlo con uno nuovo durante l'esecuzione. La definizione di un'action è quindi composta di tre parti, quali l'obiettivo (*goal*), il risultato (*result*) e il feedback. La documentazione ufficiale porta come esempio proprio la navigazione verso un punto, che ha durata variabile da alcuni secondi ad alcuni minuti [35].

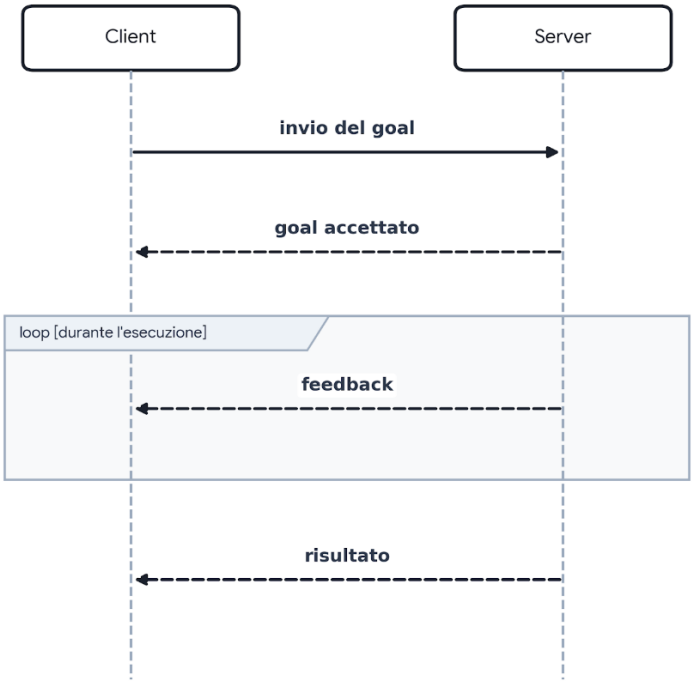


Figura 3.6 – Scambio di messaggi in un'action (immagine generata con IA)

	Topic	**Servizio**	**Action**
Modello	pubblicazione/sottoscrizione	richiesta/risposta	goal/feedback/risultato
Durata tipica	flusso continuo	breve	lunga
Feedback intermedio	no	no	sì
Annullamento	non applicabile	no	sì
Esempio	dati di un sensore	lettura di uno stato	raggiungimento di una posa

Tabella 3.3 – Confronto tra i meccanismi di comunicazione di ROS 2 [29], [34], [35]

In HIMA le action sono il punto di contatto tra il piano e il moto del robot. Nav2 espone come action la navigazione verso una posa (Sezione 3.3) e il nodo che esegue il piano ne è un client. I dettagli sono descritti nel Capitolo 4.

3.2.5 Esecuzione asincrona

Il funzionamento di un nodo ROS 2 non è di tipo sequenziale, ma orientato agli eventi. Un evento scatenante, che può essere l'arrivo di un messaggio, lo scadere di un timer, una richiesta di servizio o la risposta di un server di action, attiva il nodo, che in assenza di eventi resta in attesa. A ciascun evento è associata una funzione di *callback*. L'esecuzione delle callback viene affidata a un componente chiamato executor, il quale utilizza uno o più *thread* del sistema operativo [36]. Quando il nodo viene messo in esecuzione (operazione detta *spin*), l'executor interroga ripetutamente il middleware e richiama le callback degli eventi disponibili. La procedura va avanti finché il nodo non viene chiuso. I messaggi destinati al nodo restano nel middleware fino al momento in cui non vengono prelevati da una callback, secondo le impostazioni di QoS [36].

ROS 2 propone diverse tipologie di executor [36]. L'executor a singolo thread, usato in modo predefinito, esegue una callback alla volta, invece l'executor multi-thread può eseguire più callback in parallelo. Il parallelismo effettivo viene regolato dai gruppi di callback, le funzioni di un gruppo mutuamente esclusivo non vengono mai eseguite contemporaneamente, quelle di un gruppo rientrante possono invece essere eseguite nello stesso momento e le callback di gruppi diversi possono sempre sovrapporsi [36].

Da questo modello deriva un vincolo pratico. Se una funzione di callback resta bloccata in attesa, per esempio del completamento di una navigazione, e si utilizza un executor a singolo thread, tutti gli altri eventi del nodo non vengono gestiti.

Per superare questa limitazione le librerie client offrono chiamate asincrone. In rclpy l'invio di un goal a un server di action restituisce immediatamente una *future*, ovvero un oggetto che rappresenta il risultato di un'operazione in esecuzione, non ancora disponibile fino al suo completamento [37]. Il nodo non resta quindi bloccato e può continuare a elaborare gli altri eventi, per poi gestire l'accettazione del goal e il risultato finale, quando arrivano, tramite ulteriori callback [37].

Le chiamate sincrone restano possibili, ma una configurazione errata dei gruppi di callback può portare a situazioni di blocco (*deadlock*) [38].

Il tema è rilevante per HIMA, in cui una navigazione può durare a lungo mentre il nodo che esegue il piano deve continuare a ricevere dati e nuove richieste. La soluzione adottata è descritta nel Capitolo 4.

3.3 Navigazione autonoma

La Sezione precedente ha descritto gli strumenti con cui i componenti di un sistema ROS 2 comunicano. Questo paragrafo presenta il software che li utilizza per muovere il robot, gli elementi del problema della navigazione, gli algoritmi che li risolvono e lo stack Nav2, che in HIMA riceve gli obiettivi prodotti dal pianificatore.

3.3.1 Il problema della navigazione

Con il termine navigazione si intende portare il robot dalla sua posizione attuale a una posa obiettivo evitando collisioni con gli ostacoli presenti nell'ambiente circostante. Il problema può essere suddiviso in quattro sottoproblemi principali.

- Rappresentazione dell'ambiente: si rappresenta la realtà di interesse con una mappa in cui siano evidenziate e ben definite le zone libere, quelle occupate e quelle non ancora osservate.

- Localizzazione: la stima della posa del robot rispetto alla mappa.
- Pianificazione: il calcolo di un percorso privo di collisioni dalla posa attuale all'obiettivo.
- Controllo: la generazione dei comandi di velocità che fanno seguire il percorso al robot,deve tenere conto degli ostacoli rilevati durante la navigazione e prevedere una reazione a questi.

Queste sono le operazioni alla base della navigazione, a cui si deve aggiungere un sistema per la gestione dei fallimenti. Possono esserci casi in cui il percorso designato non è attuabile, ostacoli improvvisi che si interpongono nella traiettoria dopo la pianificazione oppure casi in cui la localizzazione può degradarsi. Una rassegna degli algoritmi disponibili in ROS 2 per ciascuno di questi compiti è proposta in [39].

I quattro sottoproblemi lavorano su piani diversi. La pianificazione ragiona sull'intera mappa e viene ripetuta con intervalli solitamente lunghi. Il controllo considera solo le zone in prossimità del robot e richiede frequenze elevate di produzione dei comandi. Questa separazione ritorna nell'organizzazione di Nav2 (Sezione 3.3.5).

3.3.2 SLAM e localizzazione

SLAM (*Simultaneous Localization and Mapping*) indica il problema di costruire la mappa di un ambiente sconosciuto e, contemporaneamente, stimare la posa del robot al suo interno. Negli ambienti interni la soluzione più comune impiega un lidar bidimensionale e produce una mappa a griglia di occupazione.

Il pacchetto di riferimento in ROS 2 è SLAM Toolbox [40]. La rappresentazione della mappa è un grafo di pose: ogni nodo corrisponde a una posa del robot con la scansione acquisita in quel punto, gli archi ai vincoli tra le pose successive. Quando il robot ritorna in una zona già osservata, il riconoscimento della corrispondenza permette di correggere l'intero grafo.

SLAM Toolbox mette a disposizione una modalità di mappatura e una di localizzazione per una mappa già costruita e salvata.

Per la stima della posa su una mappa già disponibile si utilizza un metodo di sola localizzazione. In ROS 2 il più comune è AMCL, un filtro particellare che mantiene un insieme di ipotesi sulla posa a partire dal confronto tra le scansioni del lidar e la mappa [39]. Come descritto nella Sezione 3.2.3, il componente di localizzazione pubblica la correzione tra i frame `map` e `odom`.

In HIMA la mappa e la posa stimata vengono lette dal nodo che esegue il piano e inoltrate al backend, che le mostra nell'interfaccia. Il percorso dei dati è descritto nel Capitolo 4.

3.3.3 Mappe di costo

La mappa a griglia indica quali celle sono libere, ma non è sufficiente per pianificare un movimento sicuro. Il robot ha una sua dimensione, quindi per evitare gli ostacoli non basta che il percorso non attraversi le celle occupate, ma occorre mantenere una distanza da esse. La navigazione lavora, a questo scopo, su una mappa di costo (*costmap*), una griglia in cui le celle sono associate a un valore che rappresenta quanto è sconveniente attraversare la cella stessa [41].

La mappa di costo si costruisce su livelli sovrapposti [39]:

- il livello statico riporta la mappa nota dell'ambiente
- il livello degli ostacoli aggiorna la griglia con le misure dei sensori, l'aggiornamento aggiunge i nuovi ostacoli rilevati nelle celle interessate e rimuove quelli non più presenti su di esse

- il livello di inflazione aggiunge un margine di costo attorno agli ostacoli, in funzione della dimensione del robot, per fare sì che i percorsi calcolati restino a distanza di sicurezza dagli ostacoli

Nav2 mantiene due mappe di costo distinte, ovvero una mappa globale, che si estende a tutta la realtà di interesse ed è usata per la pianificazione, l'altra locale, che si limita a una finestra nelle vicinanze del robot e si aggiorna con i dati forniti dai sensori per il controllo [41].

3.3.4 Pianificazione globale e controllo locale

Il pianificatore globale si occupa di elaborare il percorso dalla posa attuale all'obiettivo, calcolato sulla base della mappa di costo globale. Comunemente questo procedimento avviene tramite algoritmi di ricerca del cammino minimo. In alternativa si impiegano algoritmi che considerano i vincoli cinematici del robot e restituiscono percorsi che siano effettivamente attuabili da basi mobili con sterzo limitato [39].

Il controllo locale riceve il percorso e produce i comandi di velocità, che vengono modificati e corretti in base alle informazioni fornite dalla mappa di costo locale. Uno degli approcci principali prevede di generare e valutare un insieme di traiettorie candidate, scegliendo quella che viene valutata con il punteggio migliore. Un altro algoritmo insegue il percorso con una legge di controllo geometrica, ne è un esempio la *pure pursuit* regolata, che riduce la velocità in curva e in prossimità degli ostacoli [42].

In Nav2 pianificatore e controllore sono realizzati come plugin intercambiabili, scelti tramite file di configurazione [39], [41]. Gli algoritmi impiegati nelle prove di HIMA sono quelli della configurazione usata per il robot simulato, riportata nel Capitolo 5.

3.3.5 Nav2 e il suo behavior tree

Nav2 è lo stack di navigazione di ROS 2 ed è il successore di quello sviluppato per ROS 1 [41]. Non si tratta di un unico programma, ma di un insieme di nodi specializzati, quali un server per la pianificazione, uno per il controllo e uno per i comportamenti di recupero, i nodi che mantengono le mappe di costo e quello che coordina il tutto.

Il coordinamento è affidato a un behavior tree [41]. Questo componente è una struttura gerarchica, descritta in un file XML, che stabilisce l'ordine in cui devono essere invocate le azioni e la reazione prevista per il loro esito. L'albero predefinito richiama la pianificazione, avvia il controllo, ripete la pianificazione a intervalli regolari e, se un'azione fallisce, attiva i comportamenti di recupero: svuotare le mappe di costo, ruotare su sé stesso, arretrare o attendere prima di riprovare.

Questa impostazione permette di modificare la logica di navigazione intervenendo sulla struttura dell'albero, senza intervenire sul codice dei singoli server [43].

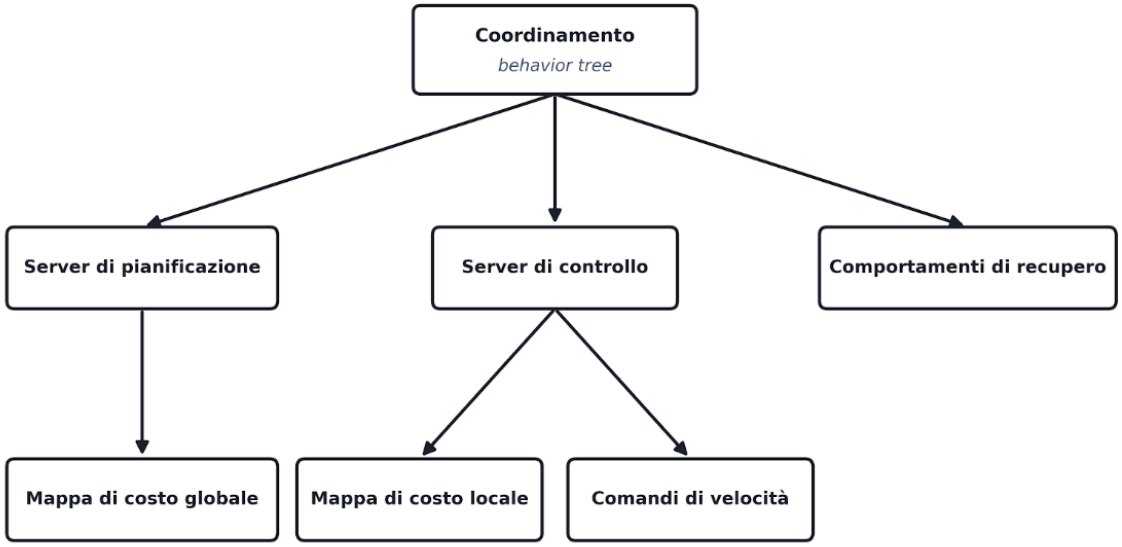


Figura 3.7 – Componenti principali di Nav2 (immagine generata con IA)

3.3.6 L'interfaccia NavigateToPose

Nav2 rende disponibili i propri compiti sotto forma di action (Sezione 3.2.4). Quella usata per raggiungere una destinazione è `NavigateToPose`: l'obiettivo contiene la posa che si vuole raggiungere, espressa con riferimento temporale e sistema di riferimento. Il server, durante l'esecuzione, invia messaggi di feedback sull'avanzamento. Al termine restituisce l'esito, che riporta se la navigazione ha avuto successo o meno. Il client ha inoltre la possibilità di annullare l'obiettivo in qualsiasi momento.

Nell'implementazione di HIMA il piano prodotto dal VLM si traduce in movimento tramite questa interfaccia, il nodo esecutore si occupa di convertire il nome della destinazione indicato nell'azione in una posa e invia il goal a Nav2.

3.4 Simulazione e piattaforme

Lo sviluppo e la verifica di un sistema robotico richiedono molte prove ripetute. Queste si effettuano in ambienti virtuali simulati, per evitare di sottoporre i prototipi a urti o possibili danneggiamenti quando il sistema è ancora in fase di collaudo. In HIMA tutte le prove sono state condotte in simulazione.

3.4.1 Simulatori: Gazebo e MuJoCo

Gazebo è un simulatore tridimensionale nato per la robotica mobile e sviluppato in stretto rapporto con l'ecosistema ROS [44]. In questo ambiente sono riprodotte la dinamica dei corpi rigidi e i contatti e sono integrati i modelli dei sensori, come il lidar e le telecamere, le cui misure vengono pubblicate sui topic come avviene per i dispositivi reali. Gli ambienti e i robot sono descritti in file di testo strutturati. Questa integrazione lo rende la scelta abituale per provare uno stack di navigazione, in quanto i nodi di SLAM e di Nav2 non distinguono un sensore simulato da uno reale.

MuJoCo è un motore fisico pensato per il controllo basato su modello [45]. Permette di rappresentare la dinamica dei sistemi articolati in coordinate generalizzate e, a vantaggio della velocità e della stabilità di simulazione, offre algoritmi efficienti per il calcolo dei contatti. Nel mondo della ricerca il suo impiego è molto diffuso per il controllo e l'apprendimento, anche per i modelli VLA descritti nella Sezione 3.1. Non nasce però come simulatore integrato in ROS, per utilizzarlo risulta quindi necessario l'impiego di componenti che traducano lo stato simulato in messaggi e i comandi in ingressi del simulatore.

	Gazebo	**MuJoCo**
Ambito di origine	robotica mobile, ecosistema ROS	controllo basato su modello
Punti di forza	modelli di sensori, mondi complessi	dinamica articolata, contatti, velocità
Integrazione con ROS 2	nativa	tramite componenti dedicati
Uso in HIMA	prove con TurtleBot3	prove con Stretch

Tabella 3.4 – Confronto tra i due simulatori impiegati

3.4.2 Piattaforme: TurtleBot3 e Stretch

TurtleBot3 è una piattaforma mobile compatta e a basso costo, diffusa nella didattica e nel mondo della ricerca [46]. Si basa su una trazione differenziale a due ruote ed è equipaggiata con un lidar a scansione completa, un'unità inerziale ed encoder sulle ruote. Hardware, firmware e software sono aperti. La sua diffusione è dovuta anche alla disponibilità di configurazioni già pronte per SLAM e Nav2, che ne fanno una scelta abituale per le prove di navigazione in simulazione. Nel progetto questa piattaforma è stata utilizzata nelle prove in Gazebo.

Stretch è un manipolatore mobile compatto e leggero, progettato per ambienti domestici [47]. È composto da una base mobile a trazione differenziale, alla quale si integrano una colonna di sollevamento e un braccio telescopico, che consente il movimento lungo direzioni cartesiane, e la capacità di manipolazione degli oggetti tramite un polso e una pinza cedevole. Dimensioni e peso sono molto contenuti, il che lo rende adatto a impieghi di assistenza domestica, stesso campo di interesse per cui è stato pensato HIMA. Nel progetto è stato usato in simulazione con MuJoCo.

	TurtleBot3	**Stretch**
Tipo	base mobile	manipolatore mobile
Locomozione	trazione differenziale	trazione differenziale
Manipolazione	assente	colonna, braccio telescopico, pinza
Ambito d'uso	didattica e ricerca	assistenza in ambienti domestici
Uso in HIMA	prove di navigazione in Gazebo	prove in MuJoCo

Tabella 3.5 – Confronto tra le due piattaforme

L'ambiente di simulazione è un'approssimazione della realtà di interesse, in cui la dinamica, l'attrito e il rumore dei sensori sono ricostruiti con modelli semplificati. Un comportamento corretto in simulazione non garantisce quindi lo stesso risultato nell'ambiente reale. Questo limite va tenuto presente nella lettura dei risultati del Capitolo 5.

4 Architettura del sistema

In questo capitolo si descrive l'architettura del sistema HIMA. Il progetto ha attraversato diverse versioni di sviluppo, fino all'implementazione corrente, che è quella riportata in questa tesi. Della versione precedente si richiamano alcuni elementi nella Sezione 4.7.1, al solo scopo di inquadrare lo sviluppo dei nodi di navigazione realizzati nell'ambito di questo lavoro.

Il capitolo si apre con una panoramica dell'organizzazione complessiva (4.1). Seguono il livello cognitivo, costituito dal backend e dal modello visione-linguaggio (4.2), l'interfaccia utente (4.3), i canali di comunicazione tra i due ambienti di esecuzione (4.4) e il livello fisico, con il simulatore e Nav2 (4.5). Le Sezioni 4.6 e 4.7 trattano l'esecuzione delle azioni e i nodi di navigazione sviluppati. La Sezione 4.8 chiude il capitolo seguendo il percorso di un comando dall'interfaccia utente fino al robot.

4.1 Panoramica

4.1.1 Organizzazione su due ambienti

Il sistema si distribuisce su due ambienti di esecuzione presenti sulla stessa macchina. Su Windows gira una parte dei componenti del sistema, il modello visione-linguaggio, eseguito localmente da LM Studio [21], il backend implementato in Python con il framework FastAPI [48] e l'interfaccia web realizzata con React [49]. In WSL2, il sottosistema Linux di Windows [50], girano invece ROS 2, Nav2, il simulatore Gazebo con il TurtleBot3 e il nodo `tcp_listener`. La suddivisione è dettata da ragioni di praticità legate alla macchina utilizzata e non da un vincolo dell'architettura, i componenti dei due lati comunicano attraverso socket TCP e possono quindi essere eseguiti anche su un unico sistema operativo.

4.1.2 I tre livelli e il bridge

Dal punto di vista funzionale, l'architettura di HIMA può essere divisa in tre livelli. Il livello cognitivo, realizzato nel backend, interpreta ed elabora le richieste dell'utente attraverso il modello visione-linguaggio. Su questo livello vengono generati i piani d'azione e viene costruita una memoria spaziale persistente degli oggetti osservati. Il livello fisico comprende ROS 2, Nav2 e il robot simulato, cioè tutte le parti del sistema che portano al movimento nell'ambiente. Il livello di interfaccia è costituito dalla dashboard web, con cui l'utente invia i comandi, riceve le risposte e consulta mappa, posa del robot e immagini.

Tra backend e livello fisico non esiste un collegamento ROS 2 diretto, i due ambienti comunicano attraverso socket TCP, passando per il bridge, il nodo `tcp_listener`, che riceve i piani, fornisce mappa, posa e immagini e traduce le azioni in comandi ROS 2. La Tabella 4.1 riassume i componenti principali del sistema e l'ambiente in cui vengono eseguiti.

Componente	File o strumento	Ruolo	Ambiente
Interfaccia	frontend/ (React, Vite)	Chat, mappa, immagini, controlli	Windows
Backend	`backend/main.py` (FastAPI)	Coordina i moduli, espone REST e WebSocket	Windows
Modulo cognitivo	`modules/brain.py`	Classificazione, piani, risposte, percezione	Windows
Memoria spaziale	`modules/memory.py`	Oggetti osservati e zone	Windows
Visione	`modules/vision.py`	Immagini da webcam o dal robot	Windows
Sintesi vocale	`modules/speech.py`	Pronuncia delle risposte	Windows
Ponte verso ROS 2	`modules/ros_bridge.py`, `modules/map_server.py`	Invio dei piani; lettura di mappa e posa	Windows
Modello	LM Studio (porta 1234)	Esecuzione locale del modello	Windows
Bridge ed esecutore	`tcp_listener.py`	Server TCP, esecuzione delle azioni	WSL2
Navigazione	Nav2, AMCL, `map_server`	Localizzazione, pianificazione, controllo	WSL2
Simulazione	Gazebo, TurtleBot3 burger	Robot e ambiente simulati	WSL2

Tabella 4.1 – Componenti principali del sistema e rispettivo ambiente di esecuzione

4.2 Livello cognitivo

Il livello cognitivo poggia interamente sul backend che, attraverso un'applicazione FastAPI, coordina alcuni moduli specializzati. Tra questi il modulo cognitivo `brain.py`, che ha il compito di interrogare il modello, gli altri si occupano della memoria spaziale, della visione, della sintesi vocale e del collegamento con il bridge. In questa sezione si descrivono il backend, il dialogo con il modello e i singoli moduli.

4.2.1 Il backend

All'avvio il backend attiva una serie di attività periodiche, riportate nella Tabella 4.2, il cui scopo è mantenere aggiornata l'interfaccia con i dati del robot e con lo stato interno del sistema. Le comunicazioni con la dashboard avvengono sulla porta 8000, attraverso un canale WebSocket e alcuni endpoint REST descritti nella Sezione 4.3.

Attività	Periodo	Contenuto
Invio dei fotogrammi	0,2 s	Ultima immagine disponibile, codificata in JPEG e base64
Invio della memoria	2 s	Oggetti e zone della memoria spaziale
Stato della modalità autonoma	1 s	Stato corrente e ultimo ragionamento del modulo autonomo
Mappa e posa	0,5 s	Posa del robot e stato del collegamento con il bridge
Lettura di mappa e posa dal bridge	0,5 s tra un ciclo e l'altro	Richieste a `tcp_listener` (Sezione 4.4.3)

Tabella 4.2 – Attività periodiche del backend

A ogni messaggio inviato dall'utente tramite la dashboard il backend segue una sequenza fissa di passaggi. Per prima cosa classifica con il modello il tipo di richiesta ricevuta, se si tratta di un'azione genera un piano e lo invia al bridge, mentre negli altri casi produce direttamente una risposta testuale. In tutti i casi la risposta viene pronunciata dalla sintesi vocale e restituita alla dashboard insieme alla

categoria della richiesta e all'eventuale piano d'azione. Le chiamate al modello avvengono in modo sincrono all'interno del gestore del WebSocket, ovvero finché il modello non risponde, il backend non può servire altre richieste. La limitazione diventa rilevante quando i tempi di inferenza sono lunghi, perché per tutta la durata della generazione il sistema resta di fatto bloccato.

4.2.2 Interfaccia con il modello

Per il dialogo con LM Studio il backend utilizza l'interfaccia compatibile con le API di OpenAI, attraverso la libreria client ufficiale. Il modello da impiegare è indicato in una variabile d'ambiente del backend, per cui la sua scelta è indipendente dal codice che gestisce la comunicazione ed è possibile sostituirlo senza modificare il resto del sistema, purché sia servito da LM Studio. Il modulo cognitivo non conserva la cronologia della conversazione, ogni chiamata è indipendente dalla precedente e costruisce ogni volta un prompt completo, che include il contesto della memoria spaziale. Non è previsto alcun limite sulla lunghezza della risposta né sui tempi di attesa, è invece variabile la temperatura, impostata più bassa per le funzioni che devono produrre un output strutturato (Tabella 4.3).

Funzione	Temperatura	Immagine	Output
`classify_intent`	non impostata	No	Una categoria: action, `visual_query` o conversation
`generate_task_plan`	0,2	No, nell'uso del backend	Piano JSON con l'elenco delle azioni
`get_vlm_response`	0,7	Solo per le domande visive	Risposta testuale breve
`perceive_and_react`	0,3	Sì	Percezioni, ragionamento, frase da pronunciare e azioni
`extract_objects_for_memory`	0,1	Sì	Elenco di oggetti con il loro stato

Tabella 4.3 – Funzioni del modulo cognitivo che interrogano il modello

4.2.3 Classificazione dell'intento

Il passo iniziale è la classificazione della richiesta dell'utente in una di tre categorie: `action`, per i comandi che richiedono movimento o manipolazione, `visual_query`, per le domande su ciò che il robot vede, `conversation`, per i dialoghi generici che non implicano richieste sulle funzionalità del sistema. Il prompt di sistema impone al modello di rispondere con una sola parola e la risposta viene poi interpretata per sottostringa, se contiene «action» la richiesta è trattata come azione, se contiene «visual» come domanda sulla visione, altrimenti come conversazione. Anche un errore nella chiamata porta alla categoria `conversation`.

4.2.4 Generazione del piano

Le richieste classificate come azioni sono gestite dalla funzione `generate_task_plan`, che chiede al modello un piano in formato JSON. Il prompt contiene la richiesta dell'utente e il contesto della memoria spaziale, e indica al modello alcune regole di pianificazione, come il fatto che le richieste di prendere e portare un oggetto devono contenere un'azione di presa prima di quella di consegna, le richieste di controllo di un luogo devono seguire uno schema di perlustrazione prefissato, ogni piano composto da più passi deve terminare con il ritorno verso l'utente e un messaggio vocale. Il modello deve inoltre usare soltanto i nomi di azione e di luogo elencati nel prompt. Un messaggio di sistema

ribadisce che il campo `actions` deve essere un elenco piatto di stringhe nella forma `azione(parametro)`.

Il vocabolario è chiuso e comprende 18 verbi, tra cui `navigate_to`, `pick_up`, `place_down`, `scan_area` e `speak`, oltre a 11 luoghi. L'elenco dei luoghi è però fissato nel codice e non viene letto dal file `locations.json` usato per la navigazione, solo cinque nomi compaiono in entrambi, sei luoghi proposti al modello non corrispondono ad alcun waypoint e sei waypoint non vengono mai proposti. Il luogo `user`, punto di rientro fissato al termine di ogni piano, non compare nell'elenco del prompt, ma è imposto dalle regole di pianificazione ed è presente tra i waypoint.

La generazione del piano avviene senza immagine, il backend passa alla funzione soltanto il testo della richiesta, anche se questa potrebbe riceverne una. Il piano dipende quindi dalla richiesta, dalle regole di pianificazione e dalla memoria spaziale, e non da ciò che il robot osserva in quel momento.

Il piano deve essere trasmesso agli altri componenti in formato JSON, a questo scopo la funzione `_extract_json` estrae i comandi dal testo prodotto dal modello. La funzione cerca dapprima un blocco di codice marcato come JSON, poi il primo oggetto racchiuso tra parentesi graffe e, se non trova nulla che corrisponda, utilizza l'intero testo. Dal JSON viene letto il campo `actions`, se la lettura fallisce, il piano risulta vuoto. Non è prevista alcuna validazione dei verbi, dei luoghi o del formato delle singole azioni, né un nuovo tentativo in caso di errore, l'unico controllo effettivo avviene al momento dell'esecuzione, nel bridge (Sezione 4.6).

In seguito all'invio del piano il backend genera una risposta di conferma con una seconda chiamata, questa volta alla funzione `get_vlm_response`, alla quale passa una frase che riporta la richiesta dell'utente e chiede di confermarla. Il piano non viene passato a questa chiamata, la conferma è prodotta senza sapere in che cosa consiste la sequenza di azioni stabilita, e viene generata anche quando il piano è vuoto o l'invio al bridge non è riuscito.

Per quanto riguarda le altre categorie di richiesta, le domande visive vengono inviate a `get_vlm_response` insieme all'ultima immagine disponibile, mentre le richieste di conversazione vengono inviate senza immagine. Il messaggio di sistema di questa funzione presenta il robot come un assistente mobile capace di muoversi e manipolare oggetti, include la memoria spaziale e chiede risposte brevi, sotto le cinquanta parole.

4.2.5 Memoria spaziale

La memoria spaziale è gestita dal modulo `memory.py`, che conserva in un file JSON gli oggetti riconosciuti dal robot, con il loro stato, l'istante dell'ultima osservazione e, se disponibile, una posa. Gli oggetti vengono raggruppati in zone di interesse con un algoritmo di clustering: un oggetto non ancora assegnato apre una nuova zona, alla quale vengono poi aggiunti tutti gli altri oggetti non assegnati che si trovano entro 1,5 m da esso. I waypoint non partecipano alla formazione delle zone, ma servono a dare loro un nome riconoscibile, utilizzabile come destinazione per la navigazione. Ad ogni zona viene associato il waypoint più vicino al suo centroide, a condizione che si trovi entro 2 m.

Il limite di questo sistema sta nel fatto che gli oggetti vengono estratti dalle immagini soltanto durante la modalità autonoma e che a tutti gli oggetti visti in uno stesso fotogramma viene assegnata la posa del robot in quel momento, invece della loro posizione effettiva.

4.2.6 Visione e sintesi vocale

Il modulo di visione gestisce due sorgenti di immagini, ovvero la webcam del computer, acquisita con OpenCV [51] a circa 10 fotogrammi al secondo, ridimensionata a 320×240 pixel e compressa in JPEG, e la camera del robot, i cui dati vengono trasferiti dal bridge sulla porta 9092. La sorgente attiva viene selezionata dall'utente nella dashboard, quando il bridge risulta raggiungibile, viene scelta automaticamente la camera del robot. Il TurtleBot3 utilizzato in simulazione non dispone però di una camera, per cui in questa configurazione dal robot non arrivano immagini.

La sintesi vocale usa la libreria pyttsx3 [52], eseguita in un thread dedicato con una coda di messaggi, in modo che la pronuncia non blocchi il resto del backend. Vengono pronunciate le risposte della chat e le frasi prodotte dalla modalità autonoma.

4.2.7 Modalità autonoma

La modalità autonoma viene attivata dall'utente dalla dashboard ed esegue un ciclo ogni 30 s. A ogni ciclo, se è disponibile un'immagine, estrae gli oggetti visibili e aggiorna la memoria, poi interroga il modello con la funzione `perceive_and_react` per ottenere una valutazione della scena ed eventualmente un piano. Il prompt di questa funzione fissa regole diverse da quelle dei comandi, per esempio se nella scena viene riconosciuta una persona il robot deve avvicinarsi e salutarla, se viene rilevato un oggetto sul pavimento deve raccoglierlo e riporlo. Le azioni proposte non vengono inviate tutte insieme, il modulo le passa al bridge una alla volta, a intervalli di 5 s, senza attendere l'esito della precedente e senza alcun coordinamento con i comandi inviati dall'utente.

4.3 Interfaccia utente

L'interfaccia è un'applicazione React eseguita dal server di sviluppo Vite [53] sulla porta 3000. La schermata principale comprende tre pannelli, quali la chat, in cui i messaggi possono essere inseriti come testo oppure dettati tramite il riconoscimento vocale del browser, il flusso di immagini, con la scelta della sorgente, i controlli, con l'interruttore della modalità autonoma, il ragionamento prodotto dal modulo autonomo, l'ultimo piano ricevuto e gli oggetti in memoria (Figura 4.1). La mappa si apre a schermo intero da un pulsante dell'intestazione (Figura 4.2).

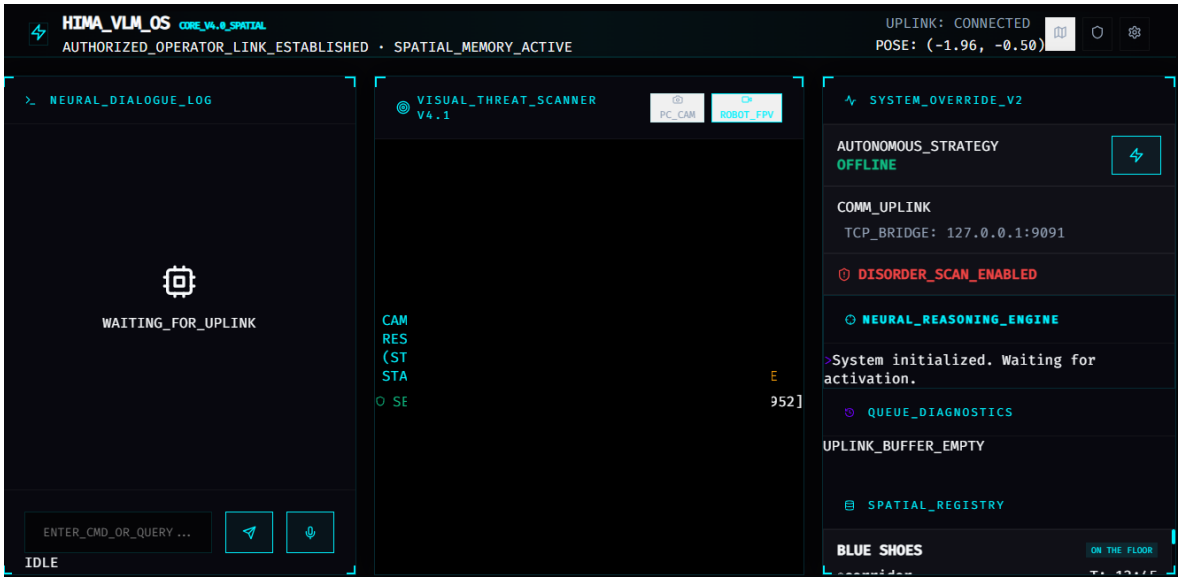


Figura 4.1 – Schermata principale della dashboard: chat, flusso di immagini e pannello dei controlli. Il pannello centrale è vuoto perché il robot simulato non dispone di una camera.

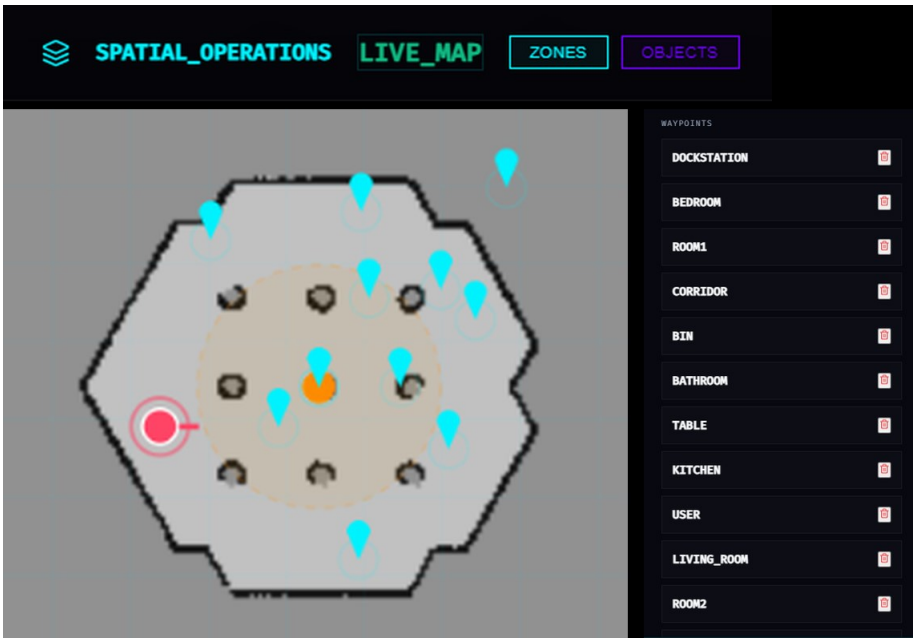


Figura 4.2 – Mappa a schermo intero: i segnaposto indicano i waypoint, il cerchio con la freccia la posa del robot e il cerchio tratteggiato una zona della memoria spaziale, i punti scuri corrispondono alle colonne dell'ambiente simulato (adattata con IA).

Il backend fornisce i dati alla dashboard attraverso due canali. Sul WebSocket arrivano le risposte della chat, i fotogrammi, lo stato della modalità autonoma, la memoria e la posa del robot. Nella direzione opposta viaggiano i messaggi della chat, l'attivazione della modalità autonoma e la scelta della sorgente video. Mappa, waypoint e memoria vengono inoltre letti periodicamente attraverso gli endpoint REST della Tabella 4.4, rispettivamente ogni 5, 10 e 3 s.

Metodo	**Percorso**	**Scopo**
GET	/map-image	Immagine PNG della mappa (dal robot o di ripiego)
GET	/map-metadata	Risoluzione, origine e dimensioni della mappa
GET	/robot-pose	Ultima posa nota del robot
GET	/bridge-status	Stato del collegamento con il bridge
GET, POST, DELETE	/waypoints	Lettura, creazione e cancellazione dei luoghi
GET	/memory	Contenuto della memoria spaziale
POST	/memory/`recluster`	Ricalcolo delle zone

Tabella 4.4 – Principali endpoint REST del backend

Il pannello della mappa disegna su un unico canvas, sempre nello stesso ordine, l'immagine della mappa, una griglia, le zone della memoria come cerchi tratteggiati, gli oggetti con posa, i waypoint e infine la posa del robot, con una freccia che ne indica l'orientamento. Per passare dalle coordinate del mondo a quelle in pixel si utilizzano la risoluzione e l'origine della mappa fornite dal backend. L'utente può inoltre aggiungere un nuovo waypoint selezionando un punto sulla mappa, il waypoint viene salvato nel file `locations.json` e diventa subito disponibile per la navigazione (Sezione 4.6.3).

4.4 Comunicazione tra i sistemi

4.4.1 Canali e porte

I componenti del sistema comunicano attraverso quattro porte, riassunte nella Tabella 4.5. Il bridge è l'unico componente che accede a ROS 2 e comunica con il backend, il backend, a sua volta, dialoga con LM Studio e, in modo bidirezionale, con la dashboard.

Porta	Estremi	Protocollo	Contenuto	Connessione
8000	Dashboard ↔ backend	HTTP (REST) e WebSocket	Chat, stato, immagini, mappa, posa, memoria	WebSocket persistente; richieste REST brevi
1234	Backend → LM Studio	HTTP, API compatibile OpenAI	Prompt e risposte del modello	Una richiesta per chiamata
9091	Backend → `tcp_listener`	TCP, messaggi JSON	Piani d'azione; richieste di mappa e posa	Una connessione breve per richiesta
9092	`tcp_listener` → backend	TCP, dati binari	Immagini JPEG della camera	Persistente

Tabella 4.5 – Canali di comunicazione del sistema

4.4.2 Invio dei piani

L'invio dei piani d'azione è affidato al modulo `ros_bridge.py`. Per ogni piano il modulo apre una nuova connessione TCP verso l'indirizzo locale sulla porta 9091, con un tempo massimo di 4 s. Il messaggio è un oggetto JSON contenente il solo campo `actions`, nel quale è riportato l'elenco delle azioni da eseguire in forma testuale. Per evitare che gli invii si sovrappongano, ciascuno di essi avviene sotto un lock. Con un periodo di 5 s il modulo verifica inoltre che il bridge sia raggiungibile, tentando una connessione con un tempo massimo di 2 s.

Il protocollo non prevede alcuna risposta da parte del bridge, la connessione viene chiusa subito dopo la ricezione (Sezione 4.6.1). La funzione di invio restituisce soltanto l'informazione che il messaggio è stato consegnato oppure no, ma questo valore non viene utilizzato dal backend, che prosegue in ogni caso.

4.4.3 Mappa e posa

Le informazioni relative a mappa e posa sono gestite dal modulo `map_server.py`, che interroga il bridge in un ciclo continuo, con una pausa di 0,5 s tra un ciclo e il successivo. Per ogni richiesta apre una nuova connessione, invia un messaggio contenente un oggetto JSON con il campo `cmd` e ne legge la risposta, con un tempo massimo di 4 s.

La richiesta `get_pose` restituisce un oggetto JSON con le coordinate x e y e l'orientamento del robot. La richiesta `get_map` restituisce invece un'intestazione di 16 caratteri, composta da due numeri decimali di 8 cifre ciascuno che indicano la lunghezza dei metadati e dell'immagine. All'intestazione seguono i metadati in formato JSON e l'immagine PNG della mappa. Se il bridge non ha ancora ricevuto una mappa, risponde con il messaggio di errore `ERROR_NO_MAP`.

La mappa di occupazione che il bridge riceve da ROS 2 viene convertita in un'immagine in scala di grigi, con livelli diversi per celle libere, occupate e sconosciute. Se non riceve alcuna mappa, il backend genera una soluzione di ripiego, costruendo un'immagine a partire dai waypoint memorizzati. Il collegamento viene considerato attivo non appena la richiesta di posa va a buon fine, anche quando la mappa non è ancora disponibile.

Per raggiungere il bridge, `map_server.py` effettua un primo tentativo con l'indirizzo locale, in caso di fallimento ricava gli indirizzi di WSL e li prova uno alla volta, ripetendo il controllo ogni 30 s. Questo meccanismo di ripiego è previsto solo per mappa e posa, l'invio dei piani e la ricezione delle immagini usano sempre l'indirizzo locale.

4.4.4 Immagini della camera

Per la condivisione delle immagini acquisite dalla camera, il bridge sottoscrive i topic ROS 2 di immagini compresse più comuni e conserva l'ultimo fotogramma ricevuto, che trasmette a ogni client connesso circa dieci volte al secondo, preceduto da un'intestazione di 4 byte che ne indica la lunghezza.

Il modulo di visione del backend mantiene una connessione persistente, legge i fotogrammi e scarta quelli incompleti, in caso di errore chiude la connessione e si riconnette dopo 2 s.

4.5 Livello fisico

Il livello fisico comprende il robot simulato, i nodi ROS 2 per la localizzazione e la navigazione e la sequenza con cui vengono avviati. Di seguito si descrive la configurazione adottata.

4.5.1 Robot e simulatore

Il robot utilizzato in simulazione è un TurtleBot3 nella versione burger, avviato in Gazebo nell'ambiente `turtlebot3_world`, costituito da un'area esagonale con una griglia di colonne. Il modello è a trazione differenziale ed è dotato di un sensore laser e di un'unità inerziale, ma non di una camera. Il robot compare nel simulatore nella posizione (−2,0; −0,5) m.

4.5.2 Mappa e localizzazione

Il sistema viene avviato da uno script che lancia Cartographer [54], il quale costruisce la mappa dell'ambiente durante il movimento, e inizializza Nav2 in modalità di localizzazione, con il server della mappa che carica la mappa statica predefinita del pacchetto TurtleBot3. In parallelo viene avviato AMCL [55], che stima la posa del robot rispetto a questa mappa. AMCL non riceve però automaticamente una posa iniziale, finché questa non viene impostata dall'utente, la posa non viene stimata e il bridge fornisce al backend la posizione predefinita (0; 0).

4.5.3 Nav2

Nav2 è configurato con un file di parametri, i cui valori principali sono riportati nella Tabella 4.6. Il pianificatore globale calcola il percorso sulla mappa con l'algoritmo di Dijkstra, il controllore locale DWB genera i comandi di velocità valutando un insieme di traiettorie possibili, il behavior tree di navigazione ricalcola il percorso durante il movimento e attiva i comportamenti di recupero, come la rotazione sul posto o l'arretramento, quando il robot non riesce ad avanzare.

Elemento	Valore
Pianificatore globale	NavFn, algoritmo di Dijkstra, tolleranza sulla destinazione 0,5 m
Controllore locale	DWB, frequenza 10 Hz, velocità lineare massima 0,6 m/s
Criterio di arrivo	0,25 m in posizione e 0,25 rad in orientamento
Controllo dell'avanzamento	Almeno 0,1 m di spostamento ogni 10 s
Costmap locale	Finestra mobile di 3 × 3 m, risoluzione 0,05 m
Robot e ostacoli	Raggio del robot 0,1 m, zona di rispetto intorno agli ostacoli 0,5 m
Behavior tree	Navigazione con ripianificazione e recupero (predefinito di Nav2)

Tabella 4.6 – Parametri principali di Nav2

4.5.4 Sequenza di avvio

Il lato WSL del sistema viene avviato da uno script che esegue i passaggi in un ordine prefissato, attendendo che ogni componente sia pronto prima di passare al successivo. Lo script carica l'ambiente ROS 2 e gli spazi di lavoro, chiude eventuali processi rimasti attivi e avvia Gazebo con il robot. Lancia quindi Cartographer e attende che venga pubblicata una mappa. Successivamente avvia Nav2 con il file di parametri configurato, apre RViz per la visualizzazione e infine avvia `tcp_listener`. I percorsi delle cartelle sono scritti all'interno dello script e vanno adattati alla macchina su cui viene eseguito.

4.6 Esecuzione delle azioni

Il piano prodotto dal livello cognitivo deve essere tradotto in azioni eseguibili dal robot. Nella versione finale del progetto questa trasformazione è affidata a un unico componente, il nodo `tcp_listener.py`, avviato nell'ambiente WSL insieme a Nav2 e al simulatore. Il piano arriva come una lista di azioni ancora in formato testuale, il nodo le interpreta e, per ciascuna, attiva la funzionalità ROS 2 corrispondente, cioè un goal di navigazione, una chiamata di servizio oppure una sequenza di comandi di velocità.

Le sottosezioni seguenti descrivono prima la struttura del nodo (4.6.1) e la procedura con cui esegue un piano (4.6.2), poi la navigazione (4.6.3), le altre azioni (4.6.4) e il caso di più piani inviati insieme (4.6.5). L'ultima (4.6.6) è dedicata al pacchetto `hima_bt_runner`, che fa parte del progetto ma non viene usato nella versione finale.

4.6.1 Il nodo bridge tcp_listener

Il nodo, denominato `hima_universal_bridge`, svolge due compiti contemporaneamente, ovvero scambia con il backend i dati del robot (mappa, posa e immagini, il cui formato è descritto nella Sezione 4.4) ed esegue i piani d'azione. In ROS 2 si presenta come un nodo scritto con la libreria `rclpy` che, all'avvio, crea le interfacce riportate nella Tabella 4.7.

Interfaccia	Nome	Tipo di messaggio	Uso
Sottoscrizione	/map	`nav_msgs/OccupancyGrid`	Conversione della mappa in un'immagine PNG da inviare al backend; durabilità transient local, per ricevere anche l'ultima mappa pubblicata prima della connessione
Sottoscrizione	`/amcl_pose`, /pose	`geometry_msgs/PoseWithCovarianceStamped`	Posa stimata del robot (x, y e orientamento)
Sottoscrizione	tre topic di immagini compresse	`sensor_msgs/CompressedImage`	Ultimo fotogramma della camera, se il robot ne dispone
Publisher	`/cmd_vel`	`geometry_msgs/Twist`	Comandi di velocità per i movimenti non gestiti da Nav2
Client di action	`navigate_to_pose`	`nav2_msgs/NavigateToPose`	Invio dei goal di navigazione a Nav2

Tabella 4.7 – Interfacce ROS 2 del nodo tcp_listener nella versione finale

Lo scambio di dati con il backend avviene su due canali separati e insieme al nodo ROS 2, il programma avvia due server TCP, ciascuno in un proprio thread e in ascolto su una porta diversa. Il primo, sulla porta 9091, riceve i piani d'azione e le richieste di mappa e posa, il secondo, sulla porta 9092, trasmette le immagini della camera. Il thread principale resta dedicato all'esecuzione dei callback del nodo (`rclpy.spin`), in modo che la ricezione di mappa, posa e immagini prosegua indipendentemente dalle connessioni TCP. Ogni connessione in ingresso sulla porta 9091 è gestita da un thread dedicato, che legge un messaggio JSON e ne riconosce il tipo, le richieste `get_map` e `get_pose` ricevono una risposta immediata, mentre i messaggi che contengono il campo `actions` vengono trattati come piani da eseguire.

Quando arriva un piano, il server non attende che venga eseguito, ma ne affida l'esecuzione a un nuovo thread e chiude subito la connessione con il backend, prima ancora che inizi la prima azione. In questo modo una navigazione lunga non impedisce di rispondere alle richieste di mappa e posa. Lo svantaggio è che, una volta chiusa la connessione, il backend non riceve alcuna conferma né sull'avvio del piano né sull'esito delle singole azioni. Le conseguenze di questa scelta sul comportamento complessivo del sistema sono discusse nel Capitolo 5.

4.6.2 La procedura execute_plan

Il metodo `execute_plan` lavora in modo sequenziale, cioè riceve la lista delle azioni generata dal modello, ad esempio `["navigate_to(kitchen)", "pick_up(bottle)", "navigate_to(user)"]`, e le esegue una alla volta nell'ordine in cui compaiono.

Ogni azione viene scomposta in verbo, la parola che precede la prima parentesi aperta, e un parametro ovvero tutto il testo compreso tra le parentesi, trattato come un'unica stringa e senza rimuovere eventuali apici. Se la stringa non termina con una parentesi chiusa, viene interpretata come un verbo privo di parametro.

Il verbo determina il ramo di esecuzione, il codice confronta il verbo con quelli che sa gestire e, per ciascuno, richiama il metodo corrispondente (navigazione, presa, rilascio, percorso circolare,

esplorazione). I verbi per cui non è previsto un gestore, compresi alcuni di quelli ammessi dal prompt del modello, vengono registrati nel log e saltati (Sezione 4.6.4).

L'esecuzione di ogni azione è racchiusa in un blocco di gestione delle eccezioni, per cui un eventuale errore viene registrato nel log e il ciclo passa all'azione successiva. Lo stesso accade per gli insuccessi che non generano eccezioni, come un luogo inesistente o un goal rifiutato da Nav2, il metodo corrispondente scrive un messaggio di log e restituisce il controllo. Di conseguenza il piano viene sempre percorso fino all'ultima azione, qualunque sia l'esito delle precedenti. Se per esempio la navigazione verso la cucina fallisce, l'azione di presa successiva viene tentata comunque nel punto in cui si trova il robot.

4.6.3 Navigazione verso un luogo

L'azione `navigate_to` è gestita dal metodo `_navigate_to_named`, che traduce il nome di un luogo in un goal per Nav2. I luoghi noti al sistema sono memorizzati nel file `locations.json`, che associa a ogni nome le coordinate x e y nel sistema di riferimento della mappa e un orientamento θ. Il file viene riletto a ogni richiesta di navigazione, senza conservarne una copia in memoria. Poiché l'utente può aggiungere o rimuovere luoghi direttamente dalla mappa della dashboard, e il backend modifica lo stesso file, le modifiche sono subito disponibili all'esecutore, senza necessità di riavviarlo.

La ricerca del luogo inizia con un processo di normalizzazione, in cui il nome ricevuto viene convertito in minuscolo e gli spazi vengono sostituiti da trattini bassi. Il nome così ottenuto viene confrontato con quelli presenti nel file e, se non esiste una corrispondenza esatta, viene accettato il primo luogo il cui nome contiene il parametro. Questa corrispondenza parziale rende il sistema tollerante a piccole variazioni del nome, ma può dare risultati inattesi. Un parametro come `room` viene associato al primo luogo che lo contiene, come `bedroom`, scelto in base all'ordine dei nomi nel file e non alla loro somiglianza, invece un parametro vuoto corrisponde a qualunque nome e viene quindi associato al primo luogo del file. Al contrario, poiché gli apici non vengono rimossi, un parametro scritto tra apici, come in `navigate_to('kitchen')`, non corrisponde ad alcun luogo. Se la ricerca non dà risultati, il metodo registra un avviso con l'elenco dei luoghi disponibili e non invia alcun goal a Nav2.

Individuate le coordinate, l'esecutore verifica che il server d'azione `navigate_to_pose` di Nav2 sia disponibile, attendendo al massimo 10 s, e costruisce un goal di tipo `NavigateToPose` nel sistema di riferimento `map`. Le coordinate x e y vengono prese dal file `locations.json` così come sono, mentre l'angolo θ viene convertito nel quaternione che rappresenta una rotazione attorno all'asse verticale. Il goal viene inviato in modo asincrono. Poiché il thread principale è già impegnato nell'esecuzione dei callback del nodo, il metodo non può usare le funzioni di attesa bloccante di `rclpy` e ricorre a un'attesa attiva. Ogni 0,05 s controlla lo stato delle future, concedendo al massimo 5 s per l'accettazione del goal e 120 s per il suo completamento.

Questa procedura presenta però due punti deboli. Allo scadere dei tempi massimi il goal non viene cancellato, ma il metodo termina e il piano prosegue, mentre Nav2 può continuare a guidare il robot verso la destinazione. Inoltre, al termine della navigazione il metodo non controlla lo stato finale restituito da Nav2, per cui il messaggio di completamento viene registrato sia quando il robot arriva a destinazione sia quando la navigazione viene interrotta. L'unico insuccesso riconosciuto esplicitamente è il rifiuto del goal da parte del server, in tutti gli altri casi il piano prosegue indipendentemente dall'esito.

4.6.4 Altre azioni

Oltre alla navigazione, l'esecutore gestisce solo una parte dei verbi che il prompt consente al modello di usare. La Tabella 4.8 riporta, per ciascun verbo del vocabolario, il comportamento effettivo.

Verbo	Gestione in `tcp_listener`	Effetto sul robot
`navigate_to`	Metodo _navigate_to_named	Goal `NavigateToPose` verso un luogo di `locations.json`
`pick_up`, grasp	Chiamata del servizio `/grasp_object/trigger` con il comando ros2 service call (tempo massimo 15 s)	Nessuno: il servizio non è presente nei pacchetti installati
`place_down`, release	Chiamata del servizio `/place_object/trigger` (tempo massimo 15 s)	Nessuno: il servizio non è presente
`circular_path`	Comandi `Twist` su `/cmd_vel` (0,25 m/s; 0,6 rad/s; 10 Hz) per N giri	Movimento circolare ad anello aperto
`scan_area`, `search_for_object`, `identify_object`, `find_person`	Un giro completo sul posto (0,8 rad/s)	Rotazione, senza elaborazione di immagini
speak	Solo messaggio di log	Nessuno (la sintesi vocale avviene nel backend)
follow, wander, `move_forward`, `make_eye_contact`, greeting, nod, `gesture_with_arm`	Nessun gestore	Azione saltata con un messaggio di log

Tabella 4.8 – Gestione dei verbi del vocabolario da parte di tcp_listener

Le azioni di presa e rilascio sono tradotte in chiamate a servizi di tipo `Trigger`, eseguite lanciando il comando `ros2 service call` come processo separato. I servizi richiamati, però, non esistono tra i pacchetti installati. Poiché il comando resta in attesa finché il servizio non diventa disponibile, ciascuna di queste azioni termina allo scadere del limite di 15 s, viene registrato un errore nel log e il piano prosegue. Inoltre il TurtleBot3 usato in simulazione non dispone di un braccio, quindi le azioni di manipolazione non potrebbero comunque avere un effetto fisico.

L'azione `circular_path` e le azioni di esplorazione non usano chiamate di servizio, ma pubblicano comandi di velocità direttamente sul topic `/cmd_vel`, senza l'ausilio di Nav2. La durata del movimento si ricava dalla velocità angolare impostata, dato che un giro completo richiede $2\pi/\omega$ secondi. Per tutta questa durata i comandi vengono inviati a 10 Hz e alla fine segue un comando di arresto. Si tratta quindi di un controllo ad anello aperto, che non tiene conto né dell'odometria né degli ostacoli. Le quattro azioni percettive, in pratica, fanno solo ruotare il robot di un giro sul posto e l'esecutore non acquisisce né analizza immagini, quindi la "ricerca" non produce alcun risultato che possa influire sul resto del piano.

Infine, l'azione `speak` si limita a scrivere un messaggio nel log del nodo. La voce del robot è gestita dal backend (Sezione 4.2), che però pronuncia la conferma prodotta dal modello con una chiamata separata, non il testo delle azioni `speak` contenute nel piano.

4.6.5 Piani concorrenti

L'esecutore non prevede alcun meccanismo di esclusione tra piani, ma ogni piano ricevuto avvia un proprio thread, anche se un piano precedente è ancora in esecuzione. Due piani inviati insieme o a breve distanza l'uno dall'altro, come una richiesta dell'utente e un'azione dettata dalla modalità

autonoma, vengono quindi eseguiti in parallelo. Per la navigazione, un nuovo goal inviato mentre un altro è attivo dovrebbe sostituire il precedente, secondo il comportamento abituale del server d'azione di Nav2. I comandi inviati direttamente sul topic `/cmd_vel`, infine, si sovrappongono a quelli prodotti da Nav2 senza alcun arbitraggio.

Il problema non viene evitato nemmeno dal lato del backend, che non tiene traccia dei piani in corso e, in modalità autonoma, invia le azioni una alla volta ogni 5 s senza aspettare che la precedente sia conclusa.

4.6.6 Il pacchetto hima_bt_runner

Il pacchetto `hima_bt_runner` è ancora presente nel progetto, ma nella versione finale non partecipa all'esecuzione dei piani, lo script di avvio lancia il convertitore `tcp_action_bridge`, al quale però nessun componente invia più piani e non avvia l'esecutore `bt_executor`. Il pacchetto viene descritto brevemente perché documenta un'impostazione alternativa, ripresa nel Capitolo 6 tra gli sviluppi futuri.

Il pacchetto definisce messaggi di tipo `ActionSequence`, che rappresenta un piano completo con un identificativo, la sorgente e l'elenco dei passi, `ActionStep`, che descrive un singolo passo con nome, parametri e testo originale e infine `ExecutionEvent`, un messaggio di notifica che indica lo stato di un passo (in coda, in esecuzione, riuscito, fallito). Il convertitore `tcp_action_bridge` riceve il piano in formato JSON sulla porta 9090 e lo trasforma in un messaggio `ActionSequence`, nel farlo il suo parser separa i parametri in corrispondenza delle virgole, per cui un'azione come `navigate_to(1.0,2.0)` produce due parametri distinti.

L'esecutore `bt_executor` raccoglie i piani ricevuti in una coda e li esegue uno alla volta. Per ciascun piano costruisce un behavior tree [56] formato da un nodo di sequenza con un figlio per ogni azione, l'albero viene valutato con una frequenza di 10 Hz e ogni azione ha a disposizione al massimo 300 s. Se un'azione fallisce, il nodo di sequenza lo segnala e il piano si interrompe senza tentare i passi successivi. L'esito di ogni passo viene pubblicato con un messaggio `ExecutionEvent` sul topic `/hima/execution_events`. Rispetto al bridge della versione finale, questo esecutore offre quindi la serializzazione dei piani, l'arresto al primo errore e una notifica esplicita degli esiti. Nel progetto, tuttavia, nessun componente del backend sottoscrive questi eventi.

4.7 Sviluppo dei nodi di navigazione

In questa sezione si presenta il contributo personale al progetto, ovvero la creazione di due nodi ROS 2 scritti per collegare i piani d'azione alla navigazione autonoma, sviluppati su una versione precedente del sistema, richiamata nella Sezione 4.7.1. Il primo, `simple_navigator.py`, è un prototipo con finalità didattiche, il secondo, `hima_navigator.py`, si inserisce nel flusso dei piani di quella versione. Nella versione finale la navigazione viene eseguita direttamente dal bridge (Sezione 4.6) e i due nodi non fanno più parte della configurazione in uso. Il confronto tra le due soluzioni è riportato nella Sezione 4.7.4. Il codice viene mostrato per estratti, con i numeri di riga dei file originali.

4.7.1 Analisi del flusso della versione precedente

Il punto di partenza è stata l'analisi del percorso seguito da un comando dell'utente nella versione iniziale del sistema, precedente a quella descritta finora. In quella configurazione il nodo

`tcp_listener` svolgeva la sola funzione di bridge, quindi riceveva i piani sulla porta 9091 e, tramite una connessione TCP dedicata, li inoltrava al convertitore `tcp_action_bridge` in ascolto sulla porta 9090, che a sua volta li pubblicava sul topic `/hima/action_sequence` come messaggi `ActionSequence`. Da quel punto l'esecuzione spettava all'esecutore `bt_executor` del pacchetto `hima_bt_runner`, basato su behavior tree, il bridge creava un client per l'action di Nav2, ma non lo utilizzava.

Da questa analisi sono stati ricavati i requisiti di un nodo che si occupasse della navigazione in modo indipendente dall'esecutore basato su behavior tree, così da completare la catena di comando fino a Nav2. Il nodo doveva raccogliere i piani dal topic `/hima/action_sequence`, riconoscere le azioni di navigazione e tradurre la destinazione, espressa come nome di un luogo o come coppia di coordinate, in una posa nel sistema di riferimento della mappa. Ottenuta la destinazione, doveva inviare il goal a Nav2 e verificarne l'esito al termine del movimento.

4.7.2 Il prototipo simple_navigator

Prima di intervenire sul flusso del sistema è stato realizzato un prototipo, `simple_navigator.py`, un nodo minimo, con il solo scopo di verificare l'interazione con Nav2 attraverso l'interfaccia action di ROS 2. Il nodo crea un client per l'action `navigate_to_pose`, invia un singolo goal con coordinate fissate nel codice e ne attende l'esito prima di terminare. La Figura 4.3 riporta il metodo che costruisce e invia il goal.

```
22      def send_goal(self, x: float, y: float, theta: float = 0.0):
23          """Invia un goal a Nav2 e aspetta il risultato."""
24          # Aspetta che il server Nav2 sia pronto
25          if not self._action_client.wait_for_server(timeout_sec=5.0):
26              self.get_logger().error('Nav2 action server non disponibile!')
27              return False
28
29          # Crea il messaggio goal
30          goal_msg = NavigateToPose.Goal()
31          goal_msg.pose.header.frame_id = 'map'
32          goal_msg.pose.header.stamp = self.get_clock().now().to_msg()
33          goal_msg.pose.pose.position.x = x
34          goal_msg.pose.pose.position.y = y
35
36          # Converti theta (yaw) in quaternione
37          goal_msg.pose.pose.orientation.z = math.sin(theta / 2.0)
38          goal_msg.pose.pose.orientation.w = math.cos(theta / 2.0)
39
40          self.get_logger().info(f'Invio goal: x={x}, y={y}, theta={theta}')
41
42          # Invia il goal in modo asincrono
43          future = self._action_client.send_goal_async(goal_msg)
44
45          # Aspetta che il goal venga accettato
46          rclpy.spin_until_future_complete(self, future)
47          goal_handle = future.result()
48
49          if not goal_handle.accepted:
50              self.get_logger().error('Goal rifiutato!')
51              return False
52
53          self.get_logger().info('Goal accettato, in esecuzione...')
54
55          # Aspetta il risultato finale
56          result_future = goal_handle.get_result_async()
57          rclpy.spin_until_future_complete(self, result_future)
58          result = result_future.result().result
59          status = result_future.result().status
60
61          if status == 4:  # SUCCEEDED
62              self.get_logger().info('Navigazione completata con successo!')
63              return True
64          else:
65              self.get_logger().error(f'Navigazione fallita! Status: {status}')
66              return False
```

Figura 4.3 – Metodo send_goal di simple_navigator.py (righe 22–66)

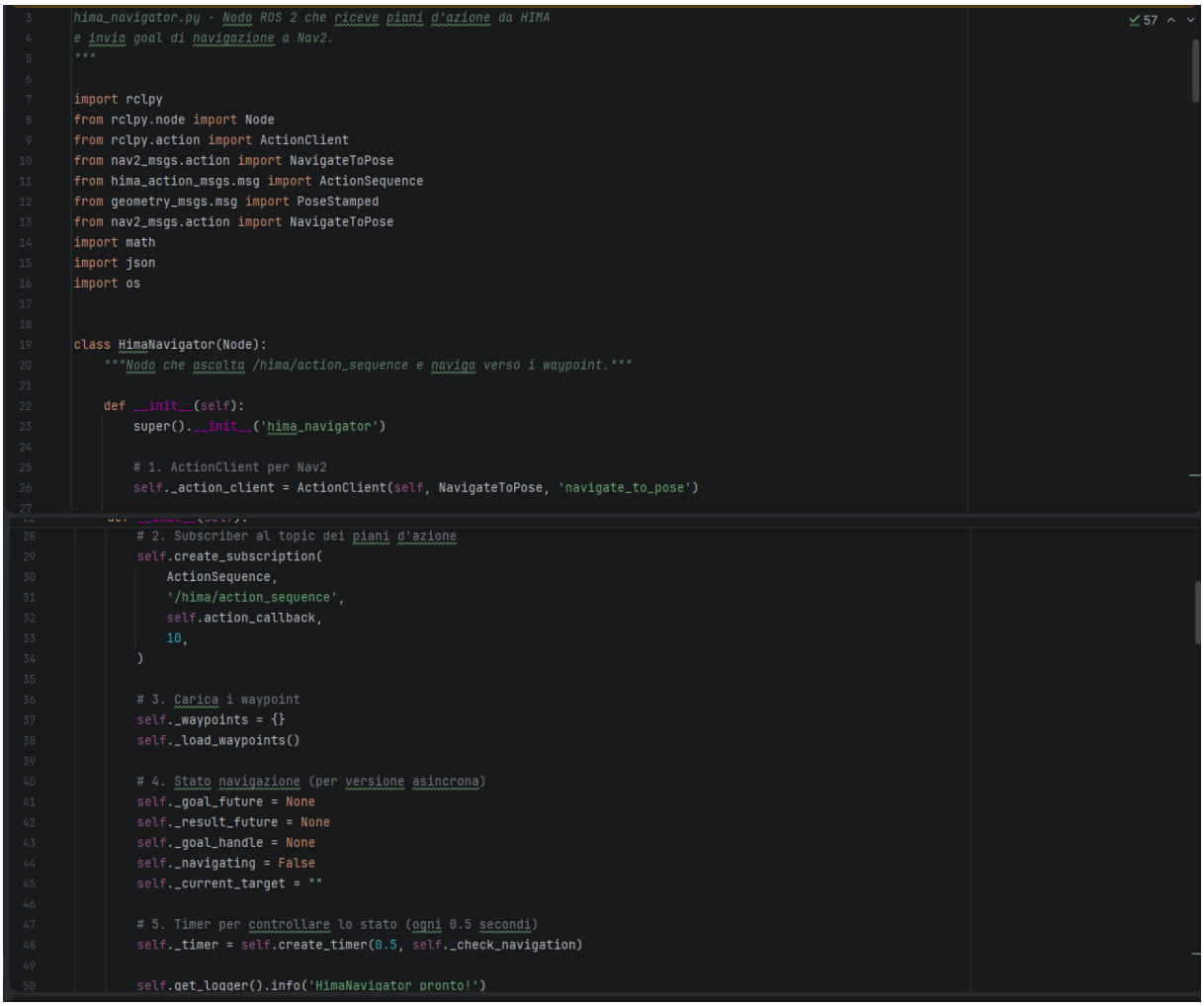
Il metodo attende innanzitutto che il server d'azione di Nav2 sia disponibile, per un tempo massimo di 5 s (righe 25–27). Successivamente costruisce il goal, un messaggio `NavigateToPose.Goal` che contiene una posa obiettivo espressa nel sistema di riferimento della mappa (righe 30–34). L'orientamento θ, cioè l'angolo di rotazione attorno all'asse verticale, viene convertito in quaternione, con componenti $z = \sin(\theta/2)$ e $w = \cos(\theta/2)$ (righe 37–38). L'invio del goal avviene in modalità asincrona con la funzione `send_goal_async`, che restituisce una future (riga 43).

L'aspetto che caratterizza questo prototipo è il modo in cui attende l'esito. Dopo l'invio, il metodo chiama `rclpy.spin_until_future_complete` per ottenere l'accettazione del goal (riga 46) e, subito dopo, una seconda volta per attendere il risultato (riga 57). Questa seconda chiamata ritorna solo quando Nav2 ha concluso la navigazione, e fino a quel momento il nodo resta bloccato e non elabora altri messaggi. Al termine, lo stato restituito da Nav2 viene confrontato con il valore 4, che corrisponde a SUCCEEDED, per distinguere una navigazione riuscita da una fallita (righe 61–66). Nel programma principale il goal è fissato alle coordinate (0,5; 0,5) m, con orientamento di 1,0 rad.

Questa struttura va bene per un programma che invia un solo goal e poi termina, ma non per un nodo che deve restare in ascolto dei piani successivi mentre il robot si muove. Da questa considerazione nasce l'esecuzione non bloccante adottata nel nodo sviluppato in seguito.

4.7.3 Il nodo hima_navigator

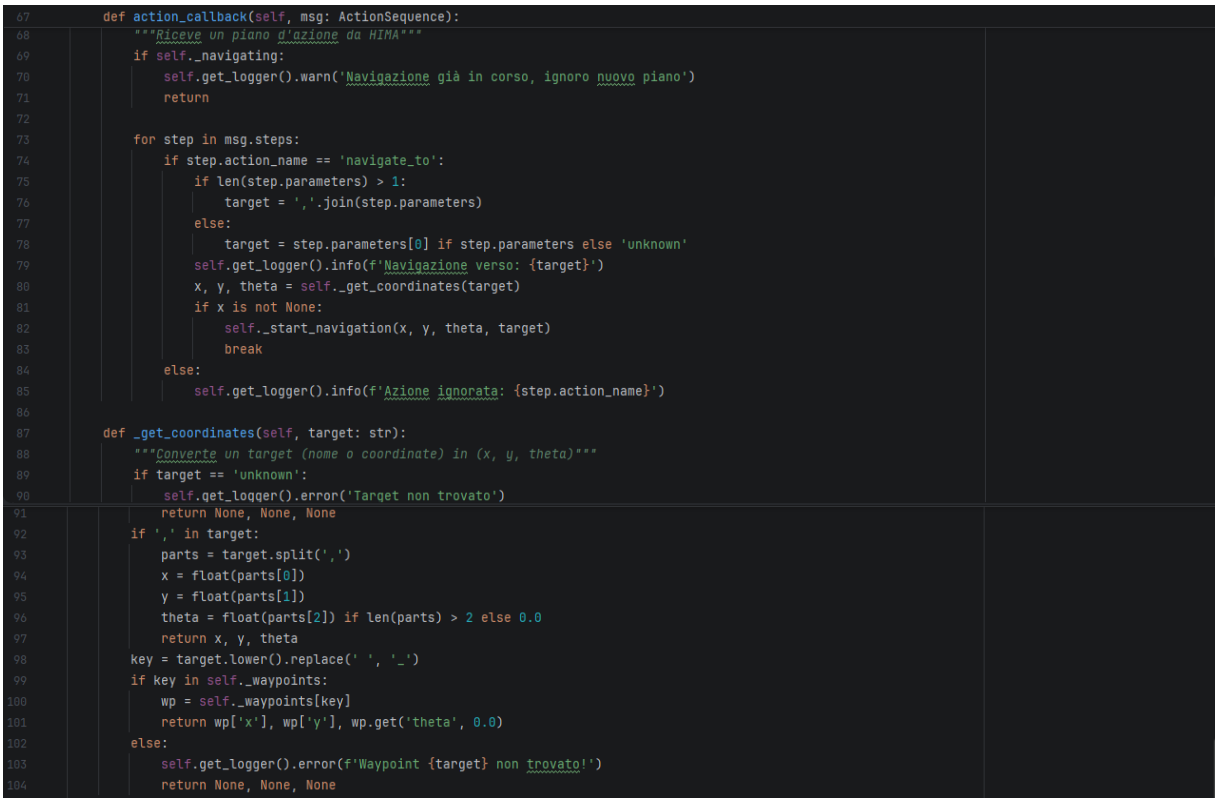
Il nodo `hima_navigator.py` applica al flusso della versione precedente i requisiti individuati nella Sezione 4.7.1. Il costruttore (Figura 4.4) crea il client per l'action di Nav2 (riga 26) e la sottoscrizione al topic `/hima/action_sequence`, associata alla funzione di callback che elabora ogni piano ricevuto (righe 29–34). I luoghi noti, che permettono all'utente di indicare una destinazione senza specificarne le coordinate, vengono caricati una sola volta, all'avvio del nodo, dal file `locations.json` usato anche dal backend (righe 37–38). Alle righe 41–45 sono definite le variabili che descrivono lo stato della navigazione, ovvero le future relative all'accettazione del goal e al risultato, il riferimento al goal accettato, un indicatore di navigazione in corso e la destinazione corrente. Infine, un timer richiama ogni 0,5 s il metodo che controlla l'avanzamento della navigazione (riga 48).



```python
"""
hima_navigator.py - Nodo ROS 2 che riceve piani d'azione da HIMA
e invia goal di navigazione a Nav2.
"""

import rclpy
from rclpy.node import Node
from rclpy.action import ActionClient
from nav2_msgs.action import NavigateToPose
from hima_action_msgs.msg import ActionSequence
from geometry_msgs.msg import PoseStamped
from nav2_msgs.action import NavigateToPose
import math
import json
import os


class HimaNavigator(Node):
    """Nodo che ascolta /hima/action_sequence e naviga verso i waypoint."""

    def __init__(self):
        super().__init__('hima_navigator')

        # 1. ActionClient per Nav2
        self._action_client = ActionClient(self, NavigateToPose, 'navigate_to_pose')

        # 2. Subscriber al topic dei piani d'azione
        self.create_subscription(
            ActionSequence,
            '/hima/action_sequence',
            self.action_callback,
            10,
        )

        # 3. Carica i waypoint
        self._waypoints = {}
        self._load_waypoints()

        # 4. Stato navigazione (per versione asincrona)
        self._goal_future = None
        self._result_future = None
        self._goal_handle = None
        self._navigating = False
        self._current_target = ""

        # 5. Timer per controllare lo stato (ogni 0.5 secondi)
        self._timer = self.create_timer(0.5, self._check_navigation)

        self.get_logger().info('HimaNavigator pronto!')
```

Figura 4.4 – Costruttore di hima_navigator.py (righe 22–50)

La Figura 4.5 riporta il callback `action_callback`, che scarta i piani ricevuti mentre una navigazione è ancora in corso (righe 69–71) e scorre i passi del piano alla ricerca delle azioni `navigate_to`. le altre azioni vengono registrate nel log e ignorate (righe 84–85). Per ogni azione di navigazione il nodo ricostruisce la destinazione a partire dai parametri del messaggio. Una destinazione espressa come coordinate, ad esempio `navigate_to(1.0,2.0)`, arriva infatti divisa in più parametri, perché il parser del convertitore li separa in corrispondenza delle virgole (Sezione 4.6.6), il nodo li ricompone in un'unica stringa (righe 75–78).

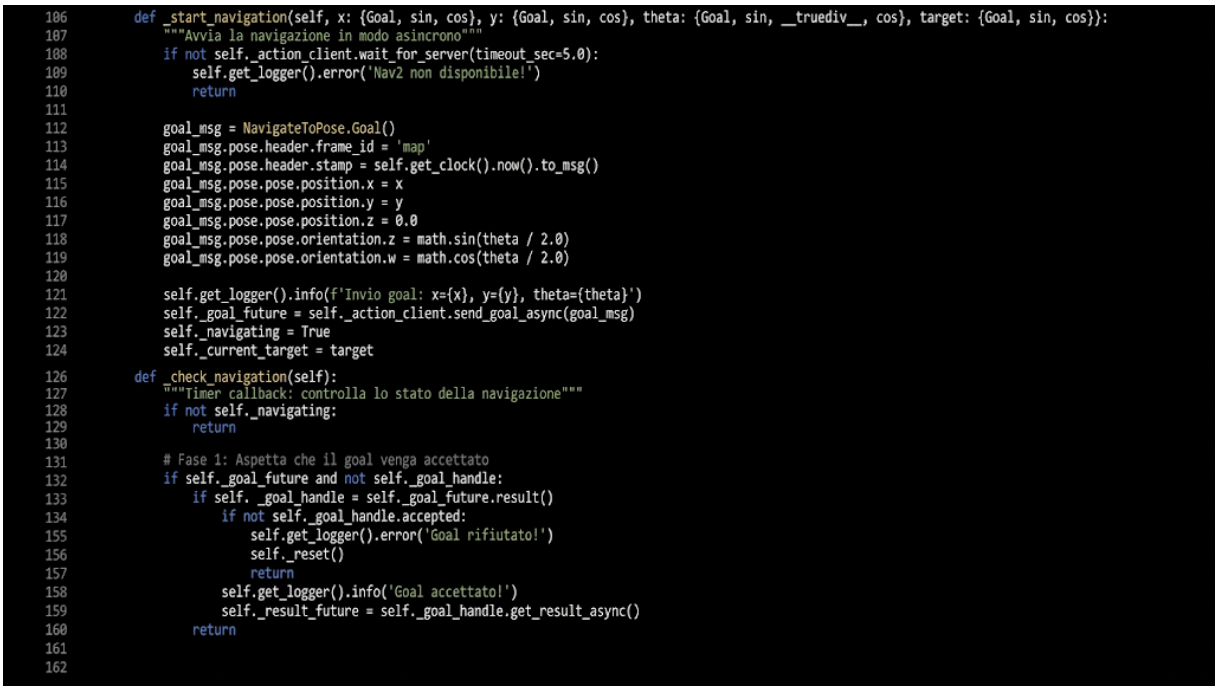


```
67      def action_callback(self, msg: ActionSequence):
68          """Riceve un piano d'azione da HIMA"""
69          if self._navigating:
70              self.get_logger().warn('Navigazione già in corso, ignoro nuovo piano')
71              return
72
73          for step in msg.steps:
74              if step.action_name == 'navigate_to':
75                  if len(step.parameters) > 1:
76                      target = ','.join(step.parameters)
77                  else:
78                      target = step.parameters[0] if step.parameters else 'unknown'
79                  self.get_logger().info(f'Navigazione verso: {target}')
80                  x, y, theta = self._get_coordinates(target)
81                  if x is not None:
82                      self._start_navigation(x, y, theta, target)
83                      break
84              else:
85                  self.get_logger().info(f'Azione ignorata: {step.action_name}')
86
87      def _get_coordinates(self, target: str):
88          """Converte un target (nome o coordinate) in (x, y, theta)"""
89          if target == 'unknown':
90              self.get_logger().error('Target non trovato')
91              return None, None, None
92          if ',' in target:
93              parts = target.split(',')
94              x = float(parts[0])
95              y = float(parts[1])
96              theta = float(parts[2]) if len(parts) > 2 else 0.0
97              return x, y, theta
98          key = target.lower().replace(' ', '_')
99          if key in self._waypoints:
100             wp = self._waypoints[key]
101             return wp['x'], wp['y'], wp.get('theta', 0.0)
102         else:
103             self.get_logger().error(f'Waypoint {target} non trovato!')
104             return None, None, None
```

Figura 4.5 – Ricezione del piano e risoluzione della destinazione in hima_navigator.py (righe 67–104)

La conversione della destinazione in coordinate è affidata al metodo `_get_coordinates` (righe 87–104), che accetta due formati. Se la stringa contiene una virgola, viene interpretata come coppia di coordinate x e y, seguita eventualmente dall'orientamento θ (righe 92–97), altrimenti viene normalizzata come nome di luogo, in minuscolo e con trattini bassi al posto degli spazi, e cercata in `locations.json`, accettando solo la corrispondenza esatta (righe 98–104). Quando la destinazione è stata risolta, la navigazione viene avviata e il ciclo si interrompe con l'istruzione `break` (righe 81–83), il nodo esegue così la prima azione di navigazione con una destinazione valida e non considera le azioni successive del piano.

Il metodo `_start_navigation` (Figura 4.6) costruisce il goal con lo stesso schema del prototipo, ma lo invia in modo asincrono, registra lo stato della navigazione (righe 122–124) e restituisce subito il controllo all'executor del nodo. L'avanzamento viene seguito dal metodo `_check_navigation`, richiamato dal timer ogni 0,5 s, che lavora in due fasi. Finché il goal non è stato accettato, controlla la future di invio e, se il goal viene rifiutato, riporta il nodo allo stato iniziale (righe 131–142). Dopo l'accettazione attende la future del risultato e ne esamina lo stato, distinguendo il successo (stato 4) dagli altri esiti (righe 144–153). In entrambi i casi il metodo `_reset` rende il nodo di nuovo disponibile per un nuovo piano (righe 155–161).



```
106         def _start_navigation(self, x: {Goal, sin, cos}, y: {Goal, sin, cos}, theta: {Goal, sin, __truediv__, cos}, target: {Goal, sin, cos}):
107             """Avvia la navigazione in modo asincrono"""
108             if not self._action_client.wait_for_server(timeout_sec=5.0):
109                 self.get_logger().error('Nav2 non disponibile!')
110                 return
111 
112             goal_msg = NavigateToPose.Goal()
113             goal_msg.pose.header.frame_id = 'map'
114             goal_msg.pose.header.stamp = self.get_clock().now().to_msg()
115             goal_msg.pose.pose.position.x = x
116             goal_msg.pose.pose.position.y = y
117             goal_msg.pose.pose.position.z = 0.0
118             goal_msg.pose.pose.orientation.z = math.sin(theta / 2.0)
119             goal_msg.pose.pose.orientation.w = math.cos(theta / 2.0)
120 
121             self.get_logger().info(f'Invio goal: x={x}, y={y}, theta={theta}')
122             self._goal_future = self._action_client.send_goal_async(goal_msg)
123             self._navigating = True
124             self._current_target = target
126         def _check_navigation(self):
127             """Timer callback: controlla lo stato della navigazione"""
128             if not self._navigating:
129                 return
130 
131             # Fase 1: Aspetta che il goal venga accettato
132             if self._goal_future and not self._goal_handle:
133                 if self. _goal_handle = self._goal_future.result()
134                     if not self._goal_handle.accepted:
155                         self.get_logger().error('Goal rifiutato!')
156                         self._reset()
157                         return
158                     self.get_logger().info('Goal accettato!')
159                     self._result_future = self._goal_handle.get_result_async()
160                 return
161 
162 
```

Figura 4.6 – Avvio e controllo asincrono della navigazione in hima_navigator.py (righe 106–161)

In questo modo il nodo non resta mai bloccato durante il movimento, ma tra un controllo e il successivo l'executor continua a elaborare i messaggi in arrivo. Il prezzo di questa soluzione è un ritardo nel rilevare l'esito, fino a 0,5 s, pari al periodo del timer.

Il nodo presenta però alcuni limiti, il principale dei quali è che funziona correttamente solo con piani che contengono una sola navigazione:

- esegue solo la prima navigazione con destinazione valida e trascura il resto del piano
- non ha una coda: i piani che arrivano durante una navigazione vanno persi
- non fissa tempi massimi: se Nav2 non risponde, il nodo rimane nello stato di navigazione e rifiuta tutti i piani successivi
- dalla lettura del codice risulta che le coordinate malformate non vengono gestite e provocherebbero un'eccezione all'interno del callback
- l'attesa del server d'azione, che può durare fino a 5 s, avviene dentro il callback e per quel tempo blocca l'elaborazione degli altri messaggi.

4.7.4 Confronto con la soluzione finale

Il nodo sviluppato e il bridge della versione finale (Sezione 4.6) affrontano lo stesso problema in modi diversi. La Tabella 4.9 mette a confronto le due soluzioni.

Aspetto	`hima_navigator`	`tcp_listener` (versione finale)
Collocazione	Nodo dedicato, a valle del topic /hima/`action_sequence`	Nel bridge, a valle della porta 9091
Azioni gestite	Solo la prima `navigate_to` con destinazione valida	Tutte le azioni del piano, in sequenza
Formato della destinazione	Nome di luogo o coordinate x,y[,θ]	Solo nome di luogo
Ricerca del luogo	Corrispondenza esatta; file letto all'avvio	Esatta, poi parziale; file riletto a ogni navigazione
Attesa dell'esito	Timer a 0,5 s sulle future	Thread dedicato con controllo ogni 0,05 s
Tempi massimi	Nessuno	5 s per l'accettazione, 120 s per il risultato
Verifica dello stato finale	Sì (stato 4 = SUCCEEDED)	No
Piani sovrapposti	Scartati durante una navigazione	Eseguiti in parallelo

Tabella 4.9 – Confronto tra hima_navigator e l'esecutore della versione finale

Rispetto al nodo sviluppato, il bridge della versione finale copre più casi, esegue tutto il piano, comprese le azioni diverse dalla navigazione e pone un limite alla durata delle attese. Il nodo sviluppato mantiene però due caratteristiche che nel bridge mancano, quali la possibilità di accettare destinazioni espresse come coordinate e controllare lo stato con cui Nav2 conclude la navigazione, distinguendo così un arrivo a destinazione da un'interruzione. In entrambe le soluzioni l'esito viene atteso controllando periodicamente le future e in nessuna delle due viene comunicato al backend. Su questi aspetti si torna nel Capitolo 6.

4.8 Flusso di un comando

Per riassumere il funzionamento del sistema si segue il percorso del comando «go to the kitchen» dalla dashboard fino al robot. I passaggi sono rappresentati nel diagramma di sequenza della Figura 4.7.

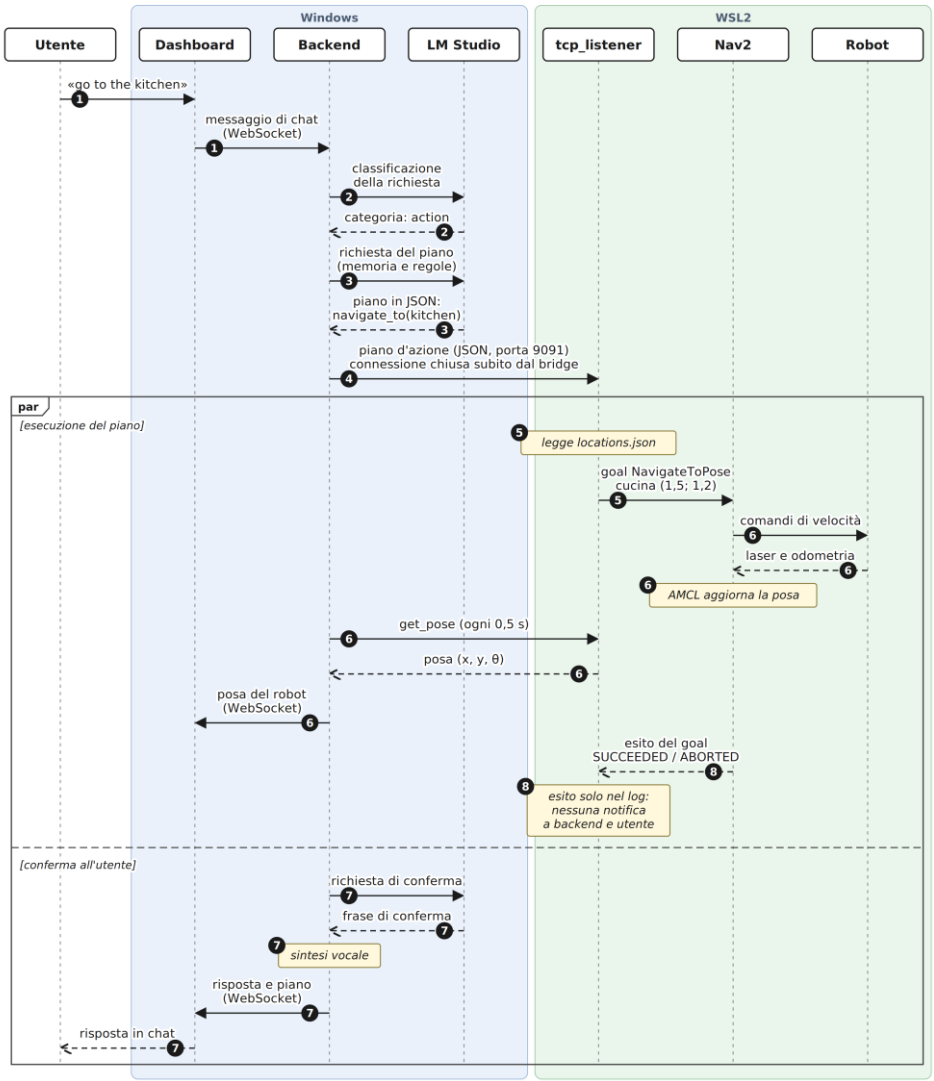


Figura 4.7 – Diagramma di sequenza del comando «go to the kitchen». Le frecce tratteggiate indicano le risposte e il blocco par le attività eseguite in parallelo; i numeri corrispondono ai passi elencati nel testo (generata con IA).

1. L'utente scrive il comando nella chat, la dashboard lo invia al backend sul WebSocket come messaggio di tipo `chat`.
2. Il backend chiede al modello di classificare la richiesta, che viene riconosciuta come `action`.
3. Il backend chiede al modello un piano, fornendo la richiesta, la memoria spaziale e le regole di pianificazione. Il modello risponde con il piano `{"actions": ["navigate_to(kitchen)"]}`, da cui viene estratto l'elenco delle azioni.
4. Il piano viene inviato a `tcp_listener` sulla porta 9091, con una connessione che il bridge chiude subito dopo averlo ricevuto.
5. Il bridge avvia un thread per il piano ed esegue `navigate_to(kitchen)`, legge il file dei luoghi, trova la cucina alle coordinate (1,5; 1,2) m e invia a Nav2 il goal corrispondente.
6. Nav2 calcola il percorso e guida il robot, mentre AMCL aggiorna la stima della posa, il backend legge la posa ogni 0,5 s e la dashboard mostra il robot in movimento sulla mappa.
7. Nel frattempo il backend, che ha già inviato il piano, chiede al modello una frase di conferma, la fa pronunciare dalla sintesi vocale e la restituisce alla dashboard insieme al piano.

8. Al termine della navigazione il bridge registra l'esito nel proprio log, il backend e l'utente non ricevono alcuna notifica.

Come mostrato in questo esempio il piano è generato a partire dalla sola richiesta testuale, l'esecuzione delle azioni è completamente separata dal backend e la conferma all'utente è prodotta in modo indipendente sia dal piano sia dal suo esito.

5 Test e validazione

Nel seguente capitolo si presentano le prove sperimentali svolte per osservare il funzionamento del sistema HIMA. Tali prove sono state condotte in ambiente di simulazione, con la configurazione dell'architettura di sistema descritta nel Capitolo 4, adattata alla macchina disponibile. Gli adattamenti e le differenze che ne derivano rispetto alla versione finale sono discussi nella Sezione 5.2.

Si definiscono inizialmente gli obiettivi e l'impostazione generale della validazione (5.1). In seguito si mostra la configurazione adottata (5.2), i vari scenari di prova con la relativa procedura (5.3) e le modalità con cui i risultati ottenuti sono stati valutati (5.4). Nelle Sezioni 5.5 e 5.6 si riportano rispettivamente i dati raccolti per le prove sulla navigazione e per il livello cognitivo. Il capitolo si conclude con la discussione dei risultati, esposta nella Sezione 5.7, mentre nella Sezione 5.8 si espongono i limiti della validazione.

5.1 Obiettivi e impostazione della validazione

La validazione ha lo scopo di verificare se il flusso di comandi descritto nel Capitolo 4, il quale ha come punto di partenza la richiesta dell'utente in linguaggio naturale per arrivare ad ottenere il movimento del robot, segue il comportamento atteso, oltre a dare una quantificazione dei tempi e dei limiti sulla macchina utilizzata. Si propone una separazione tra la valutazione per il livello fisico e quella per il livello cognitivo, con prove e metriche specifiche. Questo perché il percorso di un comando passa per diversi componenti all'interno del sistema, ciascuno dei quali introduce un proprio fattore di errore, come osservato nella Sezione 4.6.3 per il log del bridge. Si adotta tale separazione quindi al fine di limitare la somma degli errori dei diversi componenti e valutarne le capacità individuali.

I test per il livello fisico prevedono la verifica che un piano di navigazione, inviato al bridge nello stesso formato utilizzato dal backend, abbia come risultato lo spostamento del robot nella posizione richiesta. L'esito di ogni prova non è stato ricavato dal log del bridge, ma direttamente da Nav2. Questa scelta deriva dal fatto che, come descritto nella Sezione 4.6.3, `tcp_listener` registra il completamento della navigazione senza controllare lo stato finale del goal. Il messaggio del log risulta quindi insufficiente per decretare il raggiungimento del luogo indicato. Oltre all'esito, si misurano la distanza tra la posizione raggiunta e la destinazione richiesta e la durata della navigazione. Si osserva infine il comportamento del sistema quando il goal non può essere inviato, con prove in cui Nav2 è inattivo oppure la destinazione non è presente nel file dei luoghi.

Nei test effettuati per il livello cognitivo si analizza invece la risposta del modello alla stessa sequenza di chiamate che il backend esegue per un messaggio di chat, ovvero classificazione dell'intento, generazione del piano e risposta di conferma. Su queste prove si vanno a valutare la correttezza della classificazione, la validità dei piani rispetto al vocabolario e a ciò che il bridge è in grado di eseguire, la coerenza tra il piano e la conferma restituita all'utente e la latenza di ogni chiamata. Quest'ultima è un aspetto centrale, dato che il modello è eseguito in locale e senza il supporto di una scheda grafica dedicata.

Le prove sono state organizzate in due blocchi, eseguiti in momenti diversi. Il primo blocco è dedicato alla navigazione, in queste prove il modello non è caricato e i piani sono inviati direttamente al bridge. Il secondo blocco è dedicato al livello cognitivo, l'ambiente di simulazione è spento e i piani generati

non vengono inviati al robot. Modello e simulazione sono eseguiti sulla stessa macchina e, in una prova preliminare con la simulazione attiva, i tempi di inferenza sono risultati sensibilmente più elevati di quelli misurati nei test (Sezione 5.6). Mantenere tutti i componenti attivi avrebbe quindi reso i tempi dipendenti dalla contesa per processore e memoria più che dal comportamento del sistema, in questo senso la separazione è ulteriormente motivata dalla condivisione delle risorse. La controparte della configurazione adottata è che i due blocchi riproducono le interfacce tra i livelli, ma non verificano il funzionamento del sistema nel suo insieme.

Le prove sono state condotte esclusivamente in simulazione e si è utilizzato solo il modello Gemma 4 E2B, presentato nel Capitolo 2 ed eseguito sul processore della macchina. Non sono stati condotti esperimenti con misure rilevanti sul robot reale e non sono stati eseguiti test di validazione con modelli di dimensioni maggiori, come Qwen2.5-VL-7B, che richiederebbero un hardware più efficiente di quello disponibile. Tutti questi aspetti vengono ripresi nel Capitolo 6, che tratta gli sviluppi futuri. Il numero di ripetizioni per scenario risulta contenuto, per le motivazioni esposte nella Sezione 5.3. I risultati vengono quindi presentati in forma descrittiva, utilizzando conteggi e calcolando mediane e intervallo tra minimo e massimo, senza trarre conclusioni di carattere statistico.

5.2 Configurazione sperimentale

I vari test sono stati svolti su un unico personal computer, con i due ambienti di esecuzione descritti nella Sezione 4.1.1, Windows, su cui è eseguito LM Studio, e il sottosistema Linux WSL2, nel quale sono installati ROS 2, Nav2, il simulatore e il bridge. Le caratteristiche della macchina e le versioni del software utilizzate sono riportate nella Tabella 5.1.

Elemento	**Configurazione**
Processore	Intel Core i5–7200U, 2,50 GHz
Memoria	8 GB
Scheda grafica	Intel HD Graphics 620 integrata; nessuna scheda dedicata
Sistemi operativi	Windows 10 Pro 22H2 (build 19045.6466); WSL2 con Ubuntu 22.04.5
ROS 2 e navigazione	ROS 2 Humble; Nav2 con localizzazione AMCL
Simulatore	Gazebo 11.10.2; TurtleBot3 burger nel mondo `turtlebot3_world`
Esecuzione del modello	LM Studio 0.4.12 (Build 1), su Windows
Modello	Gemma 4 E2B (`google/gemma-4-e2b-it`); quantizzazione Q4_K_M; contesto 4096 token
Backend e script di misura	Python, eseguiti in WSL2

Tabella 5.1 – Configurazione hardware e software utilizzata nelle prove

L'elemento che incide di più è la potenza di calcolo della macchina. In assenza di una scheda grafica dedicata il modello viene eseguito solo sul processore, dove si trova a condividere la memoria con il simulatore e con tutti gli altri componenti. Durante le prove con la simulazione attiva il processore è rimasto al massimo del carico e la memoria quasi interamente occupata. Si tratta di osservazioni qualitative, non misurate in modo sistematico, che motivano tuttavia la separazione dei blocchi di prova descritta nella Sezione 5.1.

Il modello eseguito da LM Studio viene interrogato con l'interfaccia compatibile con le API di OpenAI (Sezione 4.2), all'indirizzo con il quale l'host Windows è raggiungibile da WSL2. Le

chiamate usano le impostazioni del backend senza modifiche, quindi temperatura 0,2 per la generazione del piano e 0,7 per le risposte testuali, mentre per la classificazione dell'intento si applica il valore predefinito del server. Per le risposte non viene impostato alcun limite di lunghezza. Con una temperatura diversa da zero la generazione non è deterministica e lo stesso comando può dare origine a risposte diverse, per questo ogni comando viene ripetuto più volte (Sezione 5.3).

La configurazione utilizzata nelle prove, rispetto alla descrizione riportata nel Capitolo 4, presenta alcune differenze dovute alla macchina utilizzata. Non è stato impiegato lo script di avvio della versione finale, poiché contiene percorsi specifici di un'altra macchina e prevede di avviare in via prioritaria ROS 2 Jazzy, mentre in WSL2 è installata soltanto la distribuzione Humble. Il lato WSL è stato per questo motivo avviato con uno script semplificato, predisposto in precedenza sulla stessa macchina, del quale è stata modificata soltanto la riga che avvia il simulatore, per impiegare il mondo `turtlebot3_world`. Lo script avvia in sequenza Gazebo con il TurtleBot3 burger in tale mondo, Nav2 attraverso il file di avvio del pacchetto `turtlebot3_navigation2` con la mappa statica predefinita e infine `tcp_listener`, del quale registra l'output in un file di log. L'operazione di localizzazione è gestita da AMCL sulla stessa mappa statica caricata dal server della mappa nella versione finale (Sezione 4.5.2), senza la costruzione della mappa da parte di Cartographer.

Un'altra differenza che risulta opportuno segnalare riguarda i parametri di Nav2. Il file dei parametri della versione finale è scritto per distribuzioni di ROS 2 successive a Humble, per alcuni plugin usa una sintassi che, come indicato nei commenti del file stesso, è stata introdotta dopo la distribuzione Iron. Nelle simulazioni Nav2 è stato quindi avviato con il file di parametri predefinito del pacchetto TurtleBot3 per il modello burger e la distribuzione Humble. All'interno di questo file alcune sezioni, tra cui quelle relative ai comportamenti di recupero e al behavior tree, utilizzano nomi propri di distribuzioni precedenti e vengono ignorate, per cui si applicano i valori predefiniti di Humble. La Tabella 5.2 confronta i parametri principali delle due configurazioni e indica, per ciascun valore della configurazione di prova, se è stato verificato sul sistema in esecuzione oppure letto direttamente dal file.

Elemento	**Configurazione di prova**	**Versione finale (Tabella 4.6)**	**Fonte del valore di prova**
Criterio di arrivo	0,25 m; 0,25 rad	0,25 m; 0,25 rad	Sistema in esecuzione
Velocità lineare massima	0,22 m/s	0,6 m/s	Sistema in esecuzione
Velocità angolare massima	1,0 rad/s	2,0 rad/s	File
Accelerazione lineare massima	2,5 m/s²	3,0 m/s²	File
Controllo dell'avanzamento	0,5 m ogni 10 s	0,1 m ogni 10 s	File
Zona di rispetto, costmap locale	1,0 m	0,5 m	File
Zona di rispetto, costmap globale	0,55 m	0,5 m	File
Pianificatore globale	NavFn, tolleranza 0,5 m	NavFn, tolleranza 0,5 m	File
Behavior tree	Predefinito di Humble, con ripianificazione e recupero	Predefinito, con ripianificazione e recupero	Sistema in esecuzione

Tabella 5.2 – Parametri principali di Nav2 nella configurazione di prova e nella versione finale

Restano coerenti con la versione riportata nel Capitolo 4 il raggio del robot (0,1 m), le dimensioni e la risoluzione della costmap locale e i criteri di valutazione delle traiettorie del controllore DWB. Tra le due configurazioni cambia inoltre, in entrambe le costmap, la rapidità con cui il costo decresce allontanandosi dagli ostacoli, mentre variano in misura meno rilevante per le prove la tolleranza temporale sulle trasformazioni usata da AMCL (1,0 s contro 0,5 s) e la frequenza attesa del pianificatore (20 Hz contro 10 Hz). Si può quindi dire che la configurazione di prova limita il robot a velocità più basse e mantiene una zona di rispetto più ampia attorno agli ostacoli nella costmap locale. I tempi di navigazione e il comportamento in prossimità degli ostacoli sono riportati nella Sezione 5.5 e vanno letti tenendo presenti queste differenze, la cui trattazione sarà ripresa nella Sezione 5.8.

Il backend è stato eseguito in WSL2 invece che in Windows e questo è solo uno degli adattamenti utilizzati nella configurazione della validazione. Nell'ambiente di prova non si dispone di una webcam per l'acquisizione delle immagini e il modulo di visione è stato quindi adattato in modo che restituisca un'immagine nera di 320×240 pixel quando la webcam non risponde, unica modifica apportata al codice della versione finale. La sintesi vocale è stata disattivata, in quanto non disponibile in WSL2. Questi adattamenti non influiscono sui blocchi di test riportati nel capitolo, nei quali il backend non viene avviato.

5.3 Scenari e procedura

Per ottenere le misure dei primi due blocchi sono stati impiegati due script di misura, ognuno dedicato al blocco in questione, i quali riproducono le interfacce dei componenti coinvolti senza modificare i file del sistema e registrano i risultati al termine di ogni prova, in modo che un'eventuale interruzione non comporti la perdita dei dati già raccolti. Per entrambi i blocchi si riportano qui di seguito gli scenari considerati e la procedura seguita.

5.3.1 Blocco di navigazione

Le prove sono svolte nel mondo `turtlebot3_world`, con il robot che ogni volta parte dalla stessa posa predefinita (−2,0; −0,5) m con orientamento nullo. Le destinazioni vengono prese dal file dei luoghi (Sezione 4.6.3) e la loro posizione nell'ambiente è mostrata nella Figura 5.1. I diversi scenari sono riassunti nella Tabella 5.3 e comprendono quattro luoghi raggiungibili, due luoghi problematici, un luogo non presente nel file e una situazione in cui Nav2 non è attivo. I luoghi problematici sono considerati tali perché ricadono all'interno dell'ingombro di una colonna sulla mappa, `table` dista circa 0,09 m dal centro di una delle colonne laterali, il cui diametro sulla mappa è di circa 0,35 m, mentre `user` coincide con il centro della colonna centrale. In tutti i casi è richiesto orientamento nullo.

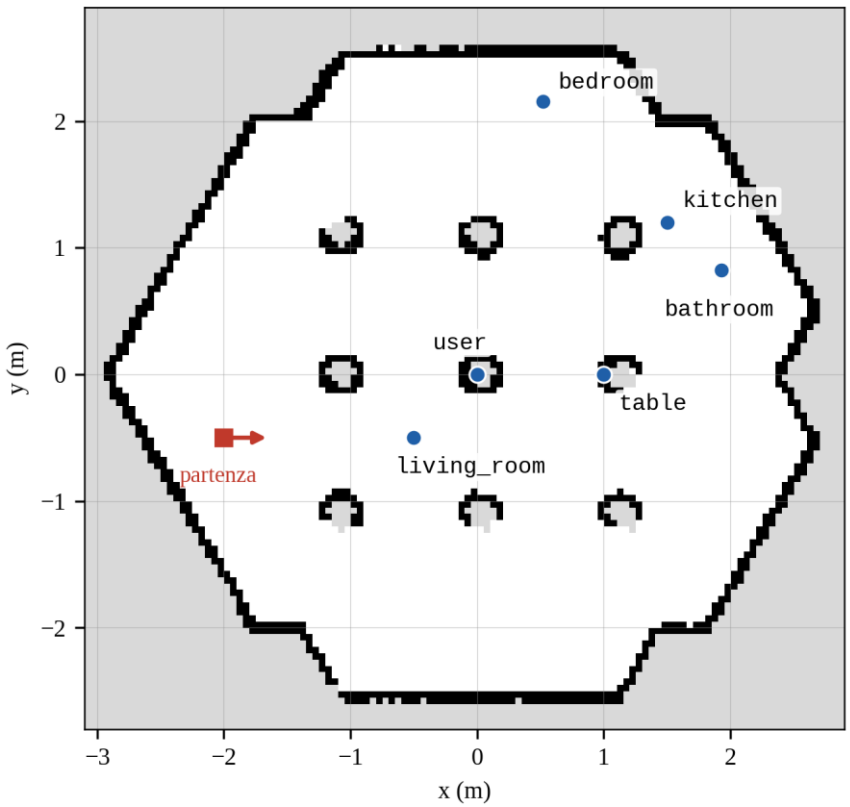


Figura 5.1 – Mappa dell'ambiente turtlebot3_world usata da Nav2, con i luoghi delle prove e la posa di partenza del robot

Luogo richiesto	Coordinate (m)	Tipo di scenario	Ripetizioni
`kitchen`	(1,50; 1,20)	Raggiungibile	10
`bedroom`	(0,52; 2,16)	Raggiungibile	10
`living_room`	(−0,50; −0,50)	Raggiungibile	10
`bathroom`	(1,93; 0,82)	Raggiungibile	10
`table`	(1,00; 0,00)	All'interno di una colonna laterale	10
`user`	(0,00; 0,00)	All'interno della colonna centrale	10
`garage`	—	Luogo non presente nel file	10
`kitchen`	(1,50; 1,20)	Nav2 non attivo	3

Tabella 5.3 – Scenari del blocco di navigazione

Quando viene avviata la serie di prove, lo script pubblica la posa iniziale sul topic `/initialpose`, utilizzando i valori della posa di partenza e una covarianza di 0,25 m² sulle coordinate x e y e di 0,0685 rad² sull'orientamento, quindi cancella il contenuto delle costmap. La posa non si imposta direttamente dall'interfaccia di RViz perché, in fase preliminare, l'impostazione manuale con la vista ruotata portava AMCL a stimare un orientamento di circa −90°, mentre la pubblicazione sul topic aveva prodotto una stima di (−1,97; −0,47) m con orientamento praticamente nullo. Un altro vantaggio è che la pubblicazione sul topic garantisce uniformità nella posa iniziale in ogni serie.

In ogni prova lo script invia al bridge, sulla porta 9091, un piano composto unicamente dall'azione `navigate_to` verso il luogo dello scenario, utilizzando lo stesso formato JSON previsto dal backend. Per la comparsa di un nuovo goal nello stato del server d'azione di Nav2 si imposta un

tempo massimo di attesa di 30 s quando il goal è atteso e di 20 s negli altri casi. Si segue poi l'evoluzione del goal fino allo stato finale, che lo riporta come completato, cancellato o interrotto. Se lo stato finale non viene raggiunto entro 300 s, il goal viene cancellato in automatico. Al termine della prova lo script registra i tempi, il numero di recuperi segnalato da Nav2, la posa stimata del robot e la parte del log del bridge scritta durante la prova.

In seguito ad ogni prova che ha portato a un effettivo movimento del robot, lo script lo riporta alla posa di partenza con un goal inviato direttamente a Nav2, senza passare per il bridge, e cancella le costmap prima e dopo il ritorno. In questo modo ogni prova inizia sempre nelle stesse condizioni, senza necessità di pubblicare nuovamente la posa iniziale. I sette scenari previsti con Nav2 attivo sono stati eseguiti in una sequenza fissa, ripetuta dieci volte, ottenendo un totale di 70 prove in un'unica serie. Lo scenario con Nav2 spento è stato invece organizzato in una serie separata di tre prove, eseguita dopo aver disattivato Nav2 e senza pubblicare la posa iniziale. Il numero di ripetizioni rientra nel limite di 20 per scenario fissato per la validazione e rappresenta un compromesso tra la durata della serie e il numero di osservazioni disponibili.

5.3.2 Blocco sul modello

Il secondo blocco valuta il modello senza avviare la simulazione, Gazebo e RViz sono mantenuti inattivi, mentre LM Studio ha in memoria il modello Gemma 4 E2B. Lo script di misura importa le funzioni del backend che interrogano il modello e ne riproduce la sequenza per un messaggio di chat (Sezione 4.2). Prevede la classificazione dell'intento e, in caso di intento `action`, la conseguente generazione del piano seguita dalla risposta di conferma, costruita con lo stesso testo usato dal backend. In tutti gli altri casi, invece, una risposta in linguaggio naturale al messaggio inviato. I prompt restano gli stessi della versione finale, compresa la memoria spaziale, di cui lo script conserva una copia all'inizio della serie. Dato che il piano non viene inviato al bridge, il robot non è coinvolto. Per misurare la durata delle operazioni e il numero di token, lo script intercetta le richieste inviate al modello durante l'esecuzione, senza apportare modifiche ai file del backend.

Gli otto comandi usati sono organizzati nella Tabella 5.4 insieme all'intento e al contenuto dei piani attesi. Questi comprendono quattro richieste di navigazione verso luoghi raggiungibili, tra cui una con il nome della destinazione composto da due parole, una richiesta di presa e consegna di un oggetto, una destinazione inesistente, un'azione di percezione e infine un messaggio che prevede una conversazione semplice. La sequenza degli otto comandi si ripete per tre volte, ottenendo un numero di 24 esecuzioni totali, con una durata complessiva di circa 63 minuti di inferenza. La durata di ciascuna esecuzione è il motivo per cui si è scelto di limitare il numero di ripetizioni. Inoltre, al fine di limitare l'effetto dell'avvio a freddo sulla prima misura, lo script esegue una classificazione preliminare non registrata.

ID	Comando	Intento atteso	Contenuto atteso del piano
C1	`go to the kitchen`	`action`	`navigate_to(kitchen)`
C2	`go to the bedroom`	`action`	`navigate_to(bedroom)`
C3	`go to the living room`	`action`	`navigate_to` verso `living_room`
C4	`go to the bathroom`	`action`	`navigate_to(bathroom)`
C5	`bring me a bottle of water from the kitchen`	`action`	`navigate_to(kitchen)`, `pick_up(…)`, `navigate_to(user)`
C6	`go to the garage`	`action`	Nessuno (luogo inesistente)
C7	`look around this room`	`action`	`scan_area(…)`
C8	`hello, how are you?`	`conversation`	Nessun piano

Tabella 5.4 – Comandi del blocco sul modello

La configurazione adottata per questi test non raggiunge una corrispondenza completa con il backend per i messaggi non classificati come azione, lo script non allega alcuna immagine, mentre il backend, per le richieste visive, prevede l'invio dell'ultimo fotogramma disponibile. Per il comando C7 risulta quindi significativa solo la classificazione, non la risposta né la sua durata.

5.4 Metriche e criteri di successo

Le metriche sono definite in maniera separata, dedicate e specifiche per ciascuno dei due blocchi di prove. Per ognuna delle grandezze designate si riportano la mediana e l'intervallo di valori tra minimo e massimo, arrotondati alla cifra indicata con le metà arrotondate per eccesso. La media non viene utilizzata perché, con un numero ridotto di valori per scenario, risente in modo marcato dei valori estremi.

5.4.1 Navigazione

Per la valutazione dell'esito di una prova si definiscono due criteri distinti. Il primo è lo stato finale del goal restituito da Nav2, la prova si considera riuscita se il goal termina nello stato di completamento. Il secondo è un criterio geometrico per cui la prova si considera riuscita se la distanza tra la posizione finale del robot e la destinazione indicata non supera 0,25 m, cioè la stessa tolleranza con cui Nav2 stabilisce l'arrivo. I due criteri possono dare esiti diversi, perché Nav2 può considerare raggiunto un goal anche quando la posizione finale non rispetta il limite di tolleranza rispetto alla destinazione richiesta, il caso è discusso nella Sezione 5.5.

La posizione finale è quella stimata da AMCL, letta come trasformazione tra il sistema di riferimento della mappa e quello del robot un secondo dopo il raggiungimento dello stato finale. Nella configurazione delle prove il simulatore non espone la posa reale del robot, per cui l'errore misurato comprende anche l'errore di localizzazione e non coincide con la distanza effettiva dalla destinazione. Si definisce l'errore di posizione come la distanza euclidea nel piano tra posizione stimata e destinazione richiesta. Come errore di orientamento si considera invece il valore assoluto della differenza tra orientamento stimato e orientamento richiesto.

La misura della durata della navigazione parte dall'invio del piano e arriva allo stato finale del goal, e viene effettuata sia sul tempo simulato sia sul tempo reale. La misura principale è il tempo riportato dall'orologio della simulazione, in quanto risulta indipendente dalla velocità con cui la macchina esegue la simulazione. Il rapporto tra i due tempi, indicato come fattore di tempo reale, quantifica il

rallentamento della simulazione. Si registrano inoltre il numero di comportamenti di recupero eseguiti durante la navigazione, ricavato dal contatore riportato da Nav2, e la presenza nel log del bridge del messaggio di completamento, che va poi confrontato con lo stato effettivo del goal. Per gli scenari in cui non è atteso un goal si verifica se un goal raggiunge comunque Nav2 e quale messaggio registra il bridge.

5.4.2 Livello cognitivo

Per il livello cognitivo si considera corretta una classificazione il cui intento restituito dal modello coincide con quello atteso (Tabella 5.4). Per ognuno dei piani generati si verifica che siano composti da una lista di azioni nella forma verbo(parametro), che i verbi appartengano al vocabolario delle azioni e che siano gestiti dall'esecutore del bridge. Le destinazioni di `navigate_to` devono inoltre essere risolvibili nel file dei luoghi con le stesse regole di `tcp_listener`, cioè per corrispondenza esatta e poi parziale (Sezione 4.6.3). Un piano non vuoto, nel formato corretto, con verbi gestiti dall'esecutore e destinazioni risolvibili si considera eseguibile. Separatamente, e indipendentemente dall'eseguibilità, si verifica che il piano contenga le azioni attese.

La coerenza tra piano e conferma si valuta leggendo le risposte di conferma relative ai comandi C1–C5. Una risposta si considera coerente se conferma l'esecuzione del comando, contraddittoria se nega di poterlo eseguire oppure se lo subordina a informazioni che il sistema possiede già, come la posizione del luogo, infine ambigua se dichiara di eseguirlo ma richiede comunque informazioni aggiuntive. La classificazione è stata eseguita manualmente. Le risposte relative al comando C6 sono trattate a parte, in quanto per un luogo inesistente non esiste un piano corretto con cui attuare un confronto.

Per ogni chiamata al modello si misura l'intervallo di tempo tra l'invio della richiesta e la ricezione della risposta, oltre al numero di token in ingresso e in uscita riportato dal server. Per i comandi di azione si calcola il tempo fino all'invio del piano, somma della classificazione e della generazione del piano, che corrisponde all'istante in cui il backend trasmetterebbe il piano al bridge. Si rileva infine il tempo complessivo, che tiene conto anche della conferma. La prima esecuzione della serie è esclusa dalle statistiche di latenza e riportata a parte, la classificazione preliminare eseguita prima della serie riguarda soltanto il prompt di classificazione, per cui la prima generazione di un piano risulta molto più lenta delle successive (Sezione 5.6).

5.5 Risultati della navigazione

Con il primo blocco di test sono state eseguite 70 prove con Nav2 attivo e 3 con Nav2 spento. In tutte le 60 prove che prevedono l'invio di un goal di navigazione Nav2 ha portato il goal allo stato di completamento e, una volta raggiunto tale stato, il robot è rientrato nella posa di partenza. Ogni prova è quindi stata eseguita con le medesime condizioni iniziali. Nella Tabella 5.5 si riportano i risultati per i diversi scenari. Nella Figura 5.2 si mostrano le posizioni finali rispetto alla destinazione richiesta, mentre nella Figura 5.3 si illustra la distribuzione degli errori e delle durate.

Scenario	Goal completati	Entro 0,25 m	Errore di posizione (m)	Durata simulata (s)	Prove con recuperi
`kitchen`	10/10	10/10	0,043 (0,005–0,068)	34,5 (29,0–52,9)	6/10
`bedroom`	10/10	10/10	0,038 (0,021–0,057)	25,1 (24,6–28,6)	0/10
`living_room`	10/10	10/10	0,224 (0,219–0,246)	6,4 (6,3–6,5)	0/10
`bathroom`	10/10	10/10	0,042 (0,017–0,049)	26,9 (25,4–29,9)	2/10
`table`	10/10	8/10	0,213 (0,156–0,283)	21,8 (18,1–38,5)	3/10
`user`	10/10	1/10	0,290 (0,199–0,336)	17,4 (13,1–23,7)	2/10

Tabella 5.5 – Risultati del blocco di navigazione per scenario: mediana e intervallo tra minimo e massimo

Sulla base del criterio inerente allo stato del goal si conclude che tutte le prove eseguite con Nav2 attivo sono riuscite. Secondo il criterio geometrico, invece, 49 prove su 60 giungono a completamento entro 0,25 m dal luogo richiesto. I due criteri risultano coincidenti nei quattro scenari in cui la richiesta è una destinazione libera da ostacoli, l'errore mediano è inferiore a 0,05 m per `kitchen`, `bedroom` e `bathroom`, mentre l'unico ad avere un risultato differente è lo scenario con `living_room`, in cui si attesta a 0,224 m, appena entro la soglia, con tutte le posizioni finali sul lato da cui il robot proviene. I due criteri divergono invece negli scenari con destinazione occupata.

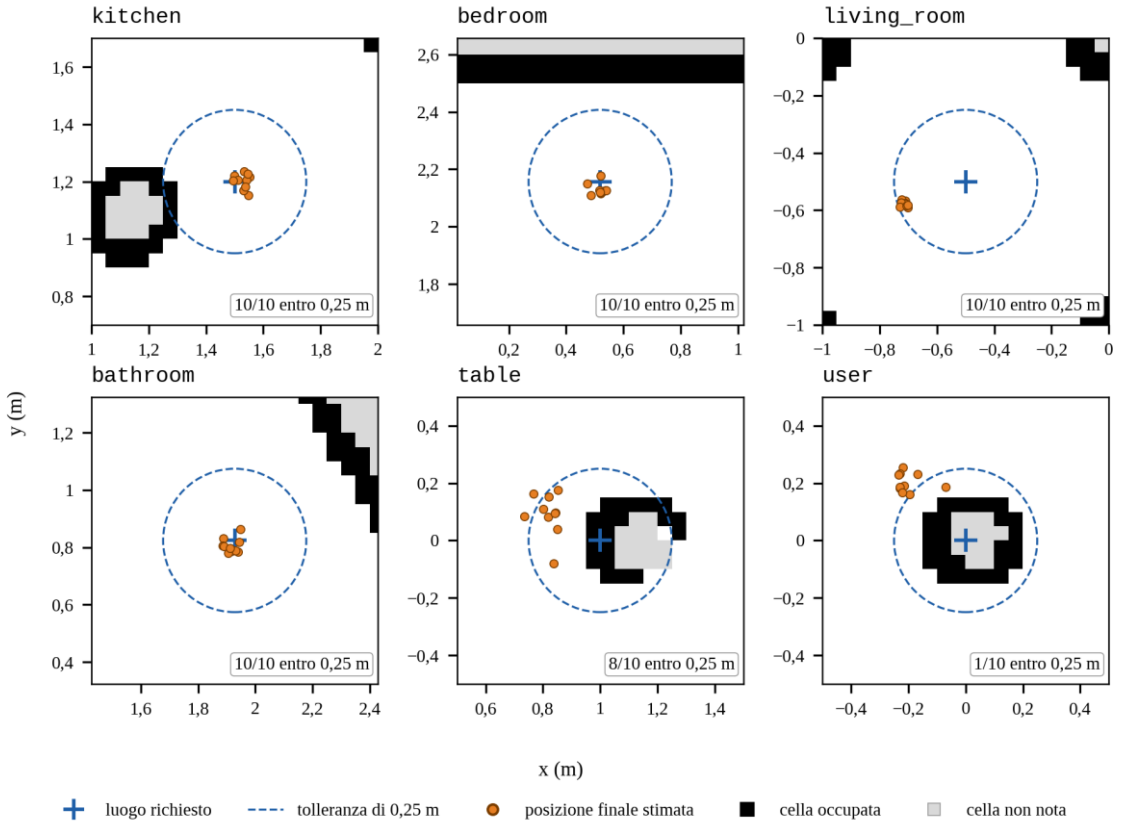


Figura 5.2 – Posizioni finali stimate nelle dieci prove di ciascuno scenario, rispetto al luogo richiesto e alla tolleranza di 0,25 m

Negli scenari `table` e `user` la destinazione indicata si trova all'interno della zona di ingombro di una colonna (Sezione 5.3.1), dove nessuna posa è raggiungibile. Nonostante questo Nav2 riporta uno stato di successo per tutti e venti i goal, anche se al termine della navigazione il robot si trova fermo al bordo dell'ostacolo, le posizioni finali arrivano a una distanza dal centro della colonna di 0,28 m (0,24–0,35) per `table` e di 0,31 m (0,21–0,35) per `user`. Il comportamento registrato risulta coerente con il funzionamento documentato dei componenti [43], il pianificatore globale non richiede che la destinazione indicata risulti libera da ingombri, ma produce un percorso che termina entro una tolleranza configurabile, pari a 0,5 m in entrambe le configurazioni (Tabella 5.2). Il criterio di arrivo del controllore, invece, confronta la posizione del robot con la fine del percorso pianificato, senza considerare la destinazione originale della richiesta. Il risultato è una navigazione dichiarata riuscita con il robot fermo a circa 0,3 m dal luogo richiesto, per `user` soltanto 1 prova su 10 si conclude entro la tolleranza di 0,25 m, per `table` 8 su 10, dato che la destinazione è più vicina al bordo della colonna. Questo è lo scarto principale che si è osservato tra il risultato effettivo e l'esito riportato dal sistema, e verrà ripreso nella Sezione 5.7.

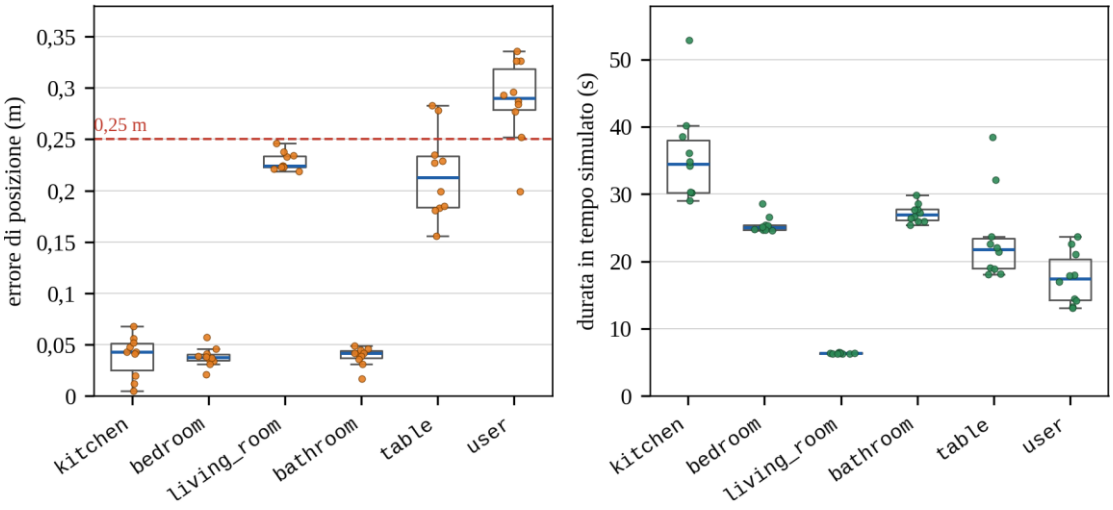


Figura 5.3 – Errore di posizione e durata in tempo simulato per scenario

Per quanto riguarda l'errore di orientamento, si osservano valori vicini tra loro in tutti gli scenari, con una mediana complessiva di 0,201 rad e un intervallo compreso tra 0,054 e 0,246 rad. Tutti i valori registrati rientrano nel limite di tolleranza impostato a 0,25 rad, il robot si arresta quindi appena entro il limite massimo, senza avvicinarsi ulteriormente all'orientamento richiesto. Il tempo impiegato per completare l'azione di navigazione riflette invece la distanza percorsa, si registrano 6,4 s per `living_room`, la destinazione più vicina al punto di partenza, e 34,5 s per `kitchen`, la più lontana. Il rapporto tra tempo simulato e tempo di orologio restituisce un fattore pari a 0,684 (0,547–0,715), la simulazione avanza quindi a poco più di due terzi della velocità del tempo reale. Questo dato conferma il peso del costo computazionale discusso nella Sezione 5.2.

Si è osservato il richiamo a comportamenti di recupero in 13 prove su 60, fino a un massimo di 3 per prova, con una netta concentrazione di questo meccanismo nello scenario con `kitchen` (9 dei 20 recuperi complessivi, distribuiti su 6 prove). Gli unici scenari in cui non sono stati registrati comportamenti di recupero sono quelli con `bedroom` e `living_room`. La destinazione `kitchen` può essere intesa come problematica sotto questo punto di vista, in quanto è quella più vicina a una cella occupata rispetto agli altri luoghi liberi, a circa 0,26 m. Si ipotizza, sulla base di questa osservazione, che la zona di rispetto di 1,0 m della costmap locale possa rendere più difficile

l'avvicinamento finale e richiamare per questo l'intervento dei recuperi. I recuperi, nelle prove interessate, hanno comunque contribuito al conseguimento del goal, a costo di durate maggiori, osservabili nei valori massimi di `kitchen` e `table`.

Un comportamento diverso del sistema si osserva negli scenari in cui non è atteso un goal. Per il luogo inesistente, in tutte le 10 prove il bridge cerca il nome indicato nel file dei luoghi e, non trovando alcuna corrispondenza, non invia alcun goal a Nav2. Nel log compare il messaggio di errore atteso. Il sistema riesce quindi a rilevare l'errore senza coinvolgere la navigazione. Nelle 3 prove eseguite con Nav2 disattivato nessun goal viene inviato, in 2 di queste il bridge segnala che il server di navigazione non è disponibile, mentre nella restante, la prima, non è stato registrato alcun messaggio che ne espliciti l'esito. In entrambe le situazioni i messaggi restano confinati nel log del bridge, senza che l'informazione sia passata al backend, il quale rimane ignaro dell'esito dell'azione (Sezione 4.6.1).

Un ultimo aspetto che vale la pena considerare riguarda la coerenza tra il log registrato e l'esito reale. In tutte le 60 prove con goal compare il messaggio che attesta il completamento dell'azione di navigazione, anche nelle 20 in cui il robot si è fermato al bordo di una colonna e nelle 11 che non rispettano la tolleranza geometrica. Il log del bridge riporta unicamente che l'azione è stata completata, senza segnalare alcuna distinzione tra una navigazione conclusa nella destinazione richiesta e una conclusa a 0,3 m di distanza.

5.6 Risultati del livello cognitivo

Il blocco di test relativo al livello cognitivo consiste in 24 esecuzioni, corrispondenti ai 3 cicli degli 8 comandi, le quali sono state completate senza errori di comunicazione con il server di inferenza. Nella Tabella 5.6 sono organizzati i risultati relativi all'interpretazione dei comandi da parte del modello.

ID	Intento restituito	Azioni del piano	Nel vocabolario	Destinazione risolta	Eseguibile
C1	`action` (3/3)	1	Sì	Sì	3/3
C2	`action` (3/3)	1	Sì	Sì	3/3
C3	`action` (3/3)	1	Sì	Sì (esatta)	3/3
C4	`action` (3/3)	1 o 2	Sì	Sì	3/3
C5	`action` (3/3)	5	Sì	Sì	3/3
C6	`action` (3/3)	1 o 2	Sì	No	0/3
C7	`visual_query` (3/3)	—	—	—	—
C8	`conversation` (3/3)	—	—	—	—

Tabella 5.6 – Intenti e piani generati nelle tre ripetizioni di ciascun comando

L'esito della classificazione coincide con il risultato atteso, e si può quindi considerare corretto, in 21 esecuzioni su 24. Si riscontrano difformità unicamente per il comando C7, che in tutte e 3 le ripetizioni viene classificato come richiesta visiva invece che come azione. L'esito di questa prova non è da considerare necessariamente un errore in senso stretto, in quanto il comando in questione chiede di osservare l'ambiente. Tale richiesta risulta ambigua e il sistema la gestisce comunque, dato che le richieste visive seguono un percorso diverso ma previsto (Sezione 4.2). La conseguenza è che l'azione `scan_area`, pur presente nel vocabolario, non viene mai generata.

I 18 piani generati per i comandi classificati come azione risultano tutti prodotti come liste di azioni nel formato atteso, con verbi appartenenti al vocabolario e correttamente gestiti dall'esecutore del bridge. Nelle 15 esecuzioni dei comandi C1–C5 il piano contiene le azioni attese. Si registra inoltre la presenza di azioni aggiuntive non richieste, che restano comunque coerenti con il comando, per il comando C4 viene prodotta, in 2 ripetizioni su 3, un'azione di annuncio vocale, mentre per il comando C5 il piano comprende stabilmente 5 azioni, tra cui il deposito dell'oggetto e l'annuncio finale, secondo le regole dichiarate nel prompt. Dei 18 piani quelli eseguibili sono 15, durante le tre esecuzioni del comando C6 si è registrata la produzione di un piano formalmente corretto, che punta però al luogo `garage`, destinazione assente dal file dei luoghi.

Il modello non segnala l'impossibilità dell'esecuzione di un comando e si limita a generare il piano, lasciando i controlli al bridge, il quale nelle prove di navigazione si comporta correttamente, scartando le destinazioni inesistenti senza inviare alcun goal (Sezione 5.5). Il comportamento del sistema risulta quindi coerente con l'assenza di una verifica del piano da parte del backend (Sezione 4.2) e con il fatto che l'elenco dei luoghi dichiarato al modello non coincide con quello usato dal bridge (Sezione 4.2.4).

Il risultato più rilevante che è stato ottenuto riguarda la risposta di conferma che viene restituita all'utente. Durante le 15 esecuzioni dei comandi C1–C5 la generazione del piano avviene correttamente, ma la conferma prodotta risulta contraddittoria in 14 casi. In 13 esecuzioni il modello dichiara l'impossibilità di raggiungere i luoghi indicati, mentre in un'altra subordina la navigazione a informazioni di cui il sistema è già in possesso, chiedendo ad esempio dove si trovi la camera. L'ulteriore esecuzione è caratterizzata da un esito ambiguo, il modello dichiara di essere pronto a eseguire la navigazione, ma nel messaggio di risposta richiede ugualmente le coordinate o una mappa. Si riporta un esempio relativo al comando C1, in cui la richiesta indica di andare in cucina e il modello risponde *"I am a mobile robot and do not have access to physical locations like a kitchen"*, mentre per lo scenario in cui si indica il soggiorno come destinazione (comando C3, terza ripetizione) la risposta è *"I am ready to navigate to the living room. Please provide coordinates or a map if available"*.

Da trattare a parte sono le conferme relative al comando C6 il quale, riguardando uno scenario con un luogo inesistente, non presenta un piano corretto da utilizzare come termine di confronto. In tutte e 3 le ripetizioni il modello nega la navigazione, motivandola in due casi con l'assenza del nome del luogo indicato all'interno della memoria spaziale a sua disposizione. La risposta generata risulta quindi appropriata, ma coincide con il comportamento del sistema solo per caso. Il modello genera comunque un piano che contiene l'azione di navigazione verso il garage, ed è il bridge a rilevare l'errore e a non inviare il goal.

L'origine di tale contraddizione è da ricercare nella modalità in cui la conferma viene prodotta. Come precedentemente descritto nella Sezione 4.2, la risposta si ottiene attraverso una chiamata separata e indipendente, che riceve unicamente il testo dato in ingresso dall'utente e la memoria spaziale. Non disponendo del piano appena generato né dell'elenco dei luoghi disponibili, il modello reagisce alla richiesta come se fosse interrogato sulle proprie capacità. La memoria spaziale, che riporta oggetti e zone osservate e non le destinazioni disponibili alla navigazione, lo porta inoltre a dichiarare di non conoscere il luogo richiesto. All'utente viene in questo modo restituito un messaggio che nega l'esecuzione di un'azione che il sistema sta invece avviando. Si tratta quindi di un difetto di progetto dell'interazione, indipendente dalla qualità dei piani, che viene ripreso nella Sezione 5.7.

Nella Tabella 5.7 e nella Figura 5.4 si riportano le latenze misurate durante le varie ripetizioni. Si osserva che la classificazione dell'intento richiede 1,54 s (1,38–2,54), mentre per la generazione del piano si registra un tempo pari a 100,8 s (98,8–129,9) e per la risposta di conferma 71,5 s (67,9–86,2). Il tempo per cui il sistema resta impegnato nell'eseguire un comando di azione corrisponde a 174,2 s (168,6–207,1) complessivi, di cui 102,3 s (100,3–131,6) prima che il piano venga passato al bridge. Il robot inizia quindi a muoversi circa un minuto e quaranta secondi dopo l'invio del comando, mentre la conferma viene restituita all'utente poco più di un minuto più tardi. Un messaggio di conversazione (comando C8) richiede invece 67,2 s (65,8–67,8).

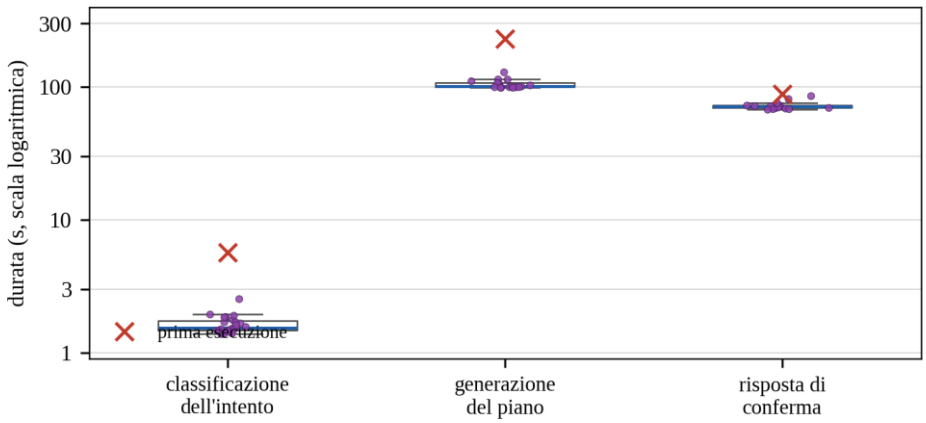


Figura 5.4 – Durata delle chiamate al modello per tipo, in scala logaritmica

Chiamata	Mediana (s)	Minimo (s)	Massimo (s)	Token in ingresso (mediana)
Classificazione dell'intento	1,54	1,38	2,54	150
Generazione del piano	100,8	98,8	129,9	1348
Risposta di conferma	71,5	67,9	86,2	925
Risposta di conversazione (C8)	67,2	65,8	67,8	915
Comando di azione completo	174,2	168,6	207,1	—

Tabella 5.7 – Latenza delle chiamate al modello, esclusa la prima esecuzione della serie

Le latenze misurate dipendono in misura dominante dalla lunghezza del prompt. La classificazione, che utilizza un prompt di circa 150 token, impiega circa 10 s ogni 1000 token ricevuti in ingresso, mentre la generazione del piano e la conferma, che hanno prompt rispettivamente di circa 1350 e 925 token, ne impiegano tra 75 e 77 s. Le risposte generate sono molto più brevi, tra i 14 e i 52 token per i piani e tra 18 e 38 per le conferme, e hanno quindi una bassa incidenza sul totale. La differenza tra le due classi di chiamate è di oltre un ordine di grandezza e indica che il carico di lavoro è dato dall'elaborazione del prompt più che dalla generazione delle risposte.

Da non lasciare in secondo piano è la latenza iniziale del sistema, quando si esegue il primo comando di una sessione. La prima esecuzione della serie è nettamente più lenta delle successive, arrivando a impiegare 5,65 s per la classificazione, 231,3 s per la generazione del piano e 88,4 s per la conferma, rispetto ai valori mediani di 1,54, 100,8 e 71,5 s calcolati sulle ripetizioni seguenti. Poiché la classificazione preliminare eseguita prima della serie riguarda soltanto il prompt di classificazione, la prima generazione di un piano risulta più che raddoppiata rispetto alle successive. Questi valori sono stati esclusi dalle statistiche generali, ma descrivono la latenza che si riscontra inviando il primo comando subito dopo l'avvio del sistema.

I valori riportati fanno riferimento alla configurazione descritta nella Sezione 5.2, che prevede di mantenere il modello in esecuzione sul processore e l'ambiente di simulazione disattivato. In una prova preliminare condotta mantenendo attiva anche la simulazione i tempi di inferenza erano risultati sensibilmente più lunghi di quelli riportati, a conferma della contesa per la potenza di calcolo della macchina descritta nella Sezione 5.1. Tale misura non è stata riportata in quanto la sequenza di chiamate non coincideva con quella del backend.

5.7 Discussione dei risultati

Nelle Sezioni 5.5 e 5.6 sono stati riportati i risultati ottenuti nelle prove del sistema, i due blocchi descrivono il comportamento di due livelli distinti. Il livello fisico si è dimostrato affidabile, in quanto in tutte le 60 prove eseguite con scenari contenenti una destinazione valida Nav2 ha portato il goal allo stato di completamento. Nelle prove di navigazione è stato registrato un errore mediano inferiore a 0,05 m nei 3 scenari privi di ostacoli in prossimità delle destinazioni e un comportamento stabile nelle varie ripetizioni. Dai test svolti per il livello cognitivo si evince un comportamento corretto del sistema nella produzione dei piani d'azione, i quali rispettano formalmente il formato atteso e le linee guida stabilite nel prompt, oltre a contenere le azioni attese in tutte le esecuzioni previste. I problemi riscontrati non riguardano quindi la capacità dei singoli componenti, ma i punti in cui le interfacce dei due livelli si incontrano, la verifica delle destinazioni, la restituzione dell'esito all'utente e i tempi di risposta.

5.7.1 Esito dichiarato ed esito effettivo

Si individuano due scarti tra il risultato ottenuto e quello che viene dichiarato dal sistema. Il primo riguarda lo stato di successo di un goal ed è emerso nelle prove di navigazione verso destinazioni che presentano un ostacolo nelle vicinanze. In questi scenari Nav2 ha dichiarato di aver completato tutti i goal con il robot fermo a circa 0,3 m dal punto indicato, e lo stesso messaggio compare nel log del bridge. Il comportamento, rispetto a come sono definiti i componenti, si può definire formalmente corretto, ma nel suo insieme il sistema non dispone di alcun controllo che confronti la posizione raggiunta con quella richiesta. L'informazione sulla distanza residua, che nelle prove è stata ricavata dalla posa stimata, non viene infatti utilizzata. In questo senso, una verifica basata su una soglia dichiarata permetterebbe di distinguere un arrivo effettivo da un arresto in prossimità dell'ostacolo e di segnalarlo all'utente.

Il secondo scarto riguarda l'assenza di un percorso inverso delle informazioni all'interno della catena di comando, comportamento che risulta più evidente nelle prove eseguite con Nav2 spento o con una destinazione non presente nel file dei luoghi. In questi scenari il bridge rileva correttamente la situazione e non invia alcun goal, ma l'informazione resta confinata nel suo log, senza che il backend riceva alcun riscontro. Anche l'utente rimane quindi all'oscuro dell'esito (Sezione 5.5). Il motivo per cui questa informazione si perde all'interno della catena di comando è che il collegamento tra backend e bridge è unidirezionale (Sezione 4.6.1). Per informare l'utente sarebbe sufficiente prevedere l'invio di un messaggio di esito sulla stessa connessione già stabilita tra i due componenti.

5.7.2 Coerenza dell'interazione

Il risultato più netto emerso dal blocco di test eseguito sul modello è la contraddizione tra il piano e la conferma, verificatasi in 14 delle 15 esecuzioni. Tra le cause è da escludere la qualità del modello, in quanto i comandi inviati ricevono una risposta corretta in termini di generazione del piano d'azione, il problema sta nella modalità con cui la conferma viene richiesta. Per l'invio del messaggio di

conferma è prevista una chiamata separata, alla quale non vengono comunicati il piano appena prodotto e l'elenco dei luoghi disponibili (Sezione 5.6). In assenza di queste informazioni l'utente riceve un messaggio che nega l'esecuzione di un'azione che il sistema sta invece avviando. Il problema in questione risulta particolarmente rilevante in un contesto assistivo, nel quale le uniche informazioni a disposizione dell'utente sono quelle che riceve dal dialogo presente nell'interfaccia, non ha quindi modo di verificare con chiarezza ciò che il sistema sta eseguendo.

Per correggere questo punto risulterebbe sufficiente intervenire sulla modalità con cui la conferma viene prodotta, in modo che questa preveda la consultazione del piano d'azione generato, oppure che venga composta direttamente a partire dal piano, senza ricorrere a una seconda chiamata al modello. Quest'ultima soluzione permetterebbe anche di eliminare la latenza della conferma, che come riportato nella Sezione 5.6 incide per oltre un terzo del tempo complessivo di un comando di azione. Lo stesso ragionamento è applicabile al problema che si riscontra negli scenari che prevedono destinazioni inesistenti, nei quali il modello genera comunque un piano d'azione verso un luogo che non conosce. Una verifica delle destinazioni sul file dei luoghi, con gli stessi criteri applicati dal bridge, prima dell'invio del piano consentirebbe di avvisare l'utente invece di lasciare che il comando si perda tra i componenti del sistema.

5.7.3 Tempi di risposta

Il limite più evidente per un uso reale sta nei tempi di latenza misurati, circa 102 s di attesa prima che il robot inizi a muoversi e circa 174 s per concludere un comando di azione, a fronte di una navigazione che richiede meno di 35 s. Da queste misure emerge che il tempo complessivo dell'esecuzione di un'azione dipende quasi esclusivamente dall'inferenza del modello e non dal movimento del robot. Le misure mostrano inoltre che il carico di lavoro dipende in misura dominante dalla lunghezza del prompt più che da quella delle risposte generate (Sezione 5.6).

Alla luce di quanto osservato si possono individuare tre direzioni di intervento, riprese tra gli sviluppi futuri nel Capitolo 6. La prima riguarda la macchina su cui il sistema viene eseguito, i tempi risulterebbero sensibilmente ridotti se il modello avesse a disposizione una scheda grafica dedicata. Le altre due prevedono invece di intervenire direttamente sul codice, riducendo la lunghezza dei prompt, e quindi il numero di token da elaborare a ogni comando, oppure limitando il numero di chiamate al modello, come osservato nella Sezione 5.7.2, la chiamata di conferma è ridondante e la sua eliminazione inciderebbe in maniera positiva sulla latenza complessiva.

Va comunque ricordato che le misure sono state raccolte tenendo la simulazione disattivata, tale configurazione non corrisponde quindi alla situazione reale di utilizzo del sistema. Mantenere tutti i componenti attivi sulla stessa macchina aumenterebbe ulteriormente i tempi di risposta, in quanto i vari componenti condividerebbero tra loro le risorse disponibili, gravando in misura maggiore sul carico computazionale.

5.7.4 Validità dei risultati

I valori riportati sono tutti riferiti alla configurazione descritta nella Sezione 5.2, che differisce dalla versione finale per alcuni parametri di Nav2 (Tabella 5.2). Questi parametri influenzano le durate della navigazione, la configurazione finale ammette velocità quasi triple e le durate sarebbero quindi sensibilmente minori di quelle misurate. Inoltre la costmap locale della configurazione di prova presenta una zona di rispetto più ampia, che rende il comportamento più prudente in presenza di ostacoli e può aver influito sui recuperi richiamati nelle prove. Restano invece equivalenti nelle due configurazioni il criterio di arrivo e la tolleranza del pianificatore globale, in quanto i parametri

corrispondenti sono identici, lo scarto tra esito effettivo ed esito dichiarato discusso nella Sezione 5.7.1 risulterebbe pertanto lo stesso anche nella versione finale.

I risultati dei test eseguiti per il livello cognitivo non risentono in alcun modo dei parametri di Nav2, ma dipendono dal modello impiegato e dai prompt utilizzati. Le latenze sono comunque da considerare relative alla macchina utilizzata, mentre la contraddizione che si è presentata tra piano e conferma deriva dalla struttura del backend e si manifesterebbe quindi con qualsiasi modello utilizzato con i medesimi prompt.

5.8 Limiti della validazione

La validazione presenta alcuni limiti, dei quali si è tenuto conto nella lettura dei risultati.

Il limite principale è l'assenza di una verifica sistematica del sistema nella sua totalità: l'esecuzione di un comando a partire dall'interfaccia web, mantenendo tutti i componenti attivi, non è stata oggetto di misure. Non è stato quindi possibile valutare il comportamento della catena completa, ma soltanto osservare le prestazioni dei due livelli, fisico e cognitivo, eseguiti separatamente.

Oltre a questo aspetto si deve tenere conto del fatto che le prove sono state eseguite unicamente in ambiente simulato, nel quale manca il rumore presente negli scenari reali e la mappa presenta pareti e ostacoli semplici e regolari, priva di persone, di oggetti mobili e delle condizioni di illuminazione di un ambiente reale. Il robot simulato, non avendo a disposizione una telecamera, non permette di esercitare le capacità percettive per cui il sistema è stato progettato. Il modulo di visione è stato adattato in modo da restituire un'immagine nera (Sezione 5.2) e, nelle prove sul modello, il comando percettivo è stato classificato come richiesta visiva senza che venisse generata l'azione corrispondente. Lo stesso vale per le azioni di presa e deposito, le quali sono presenti nei piani generati, ma non sono eseguibili sul TurtleBot3 e non sono state verificate.

Da tenere in considerazione è l'ulteriore errore di localizzazione (Sezione 5.4.1) che si origina dall'utilizzo della stima di AMCL per le posizioni finali, non essendo disponibile nella configurazione di prova la posa reale del robot. Tutte le prove sono inoltre partite dalla stessa posa, fornita ad AMCL mediante pubblicazione sul topic dedicato, e la mappa statica corrisponde esattamente all'ambiente simulato. Non risulta quindi oggetto di valutazione la robustezza della localizzazione a fronte di una posa iniziale errata o di una mappa non aggiornata.

Il numero delle ripetizioni è contenuto, 10 per scenario nel blocco di navigazione e 3 per comando nei test per il livello cognitivo, e gli 8 comandi inviati al modello sono formulati in inglese e con una sintassi di base. Le prove hanno avuto come soggetto di studio un solo modello, non è stato quindi possibile distinguere i limiti dell'architettura da quelli del modello impiegato. Uno degli scenari previsti per i test è stato infine escluso, perché le prove erano state eseguite con una configurazione non confrontabile con le altre della serie.

Resta infine da menzionare la differenza tra la configurazione finale del sistema e quella adottata nelle prove, discussa nella Sezione 5.7.4. Essendo stati i test eseguiti con la simulazione spenta, inoltre, le misure di latenza raccolte per il livello cognitivo non corrispondono a quelle dell'impiego reale del sistema.

6 Conclusioni e sviluppi futuri

Nel presente trattato è stato presentato il progetto HIMA, un sistema di robotica assistiva che, utilizzando un modello visione-linguaggio eseguito in locale, ha la capacità di interpretare le richieste dell'utente espresse in linguaggio naturale e tradurle in piani d'azione i quali, attraverso ROS 2 e Nav2, dettano il movimento di un robot mobile. Nell'introduzione è stato espresso l'obiettivo nel seguente lavoro di descrivere l'architettura del sistema nella sua versione finale e di andare a verificare il funzionamento di questo in ambiente simulato, al fine di individuare i punti di forza nella catena di comando e i fattori che ne limitano l'impiego. Di seguito si riportano i risultati principali ottenuti (Sezione 6.1), il contributo personale (Sezione 6.2), gli sviluppi futuri (Sezione 6.3) e le conclusioni tratte sulla base dei risultati (Sezione 6.4).

6.1 Risultati ottenuti

I risultati misurati nel Capitolo 5 possono essere organizzati in tre esiti.

Il primo esito si riferisce al livello fisico. Dai test condotti questa parte del sistema si è dimostrata affidabile. Il risultato più significativo è stato il fatto che nelle 60 prove in scenari che prevedevano una destinazione valida, Nav2 ha portato il goal designato a completamento. In tre dei quattro scenari con destinazione libera da ostacoli è stato misurato un errore di posizione mediano inferiore a 0,05 m e nel quarto un errore maggiore, ma comunque entro il limite di tolleranza (Sezione 5.5). Un comportamento affidabile è stato rilevato anche in quegli scenari in cui veniva richiesto come posizione finale un luogo non presente nel file `locations.json` e che pertanto dovrebbe risultare estraneo al sistema. Il bridge si è dimostrato in grado di riconoscere la situazione e non avviare l'azione di navigazione. Anche nello scenario in cui il server Nav2 è mantenuto disattivato non viene inviato alcun goal, risultato coerente con quello atteso.

Il secondo riguarda invece il livello cognitivo, testato con il modello Gemma 4 E2B eseguito in locale sul processore della macchina disponibile. I piani d'azione, generati in risposta alle richieste proposte nelle varie prove, si sono dimostrati corretti nel rispetto del formato e coerenti con il vocabolario delle azioni. Per tutti i comandi di cui era stabilito il contenuto atteso, i piani comprendono le azioni previste (Sezione 5.6). Un limite è invece stato rilevato per gli scenari che prevedono richieste che il sistema non può eseguire, come la navigazione verso un luogo inesistente. Il modello non segnala queste richieste e genera comunque il piano d'azione, che risulta formalmente corretto, ma non eseguibile e scartato solo a valle dal bridge.

Il terzo esito mostra che i problemi e i limiti riscontrati non dipendono dalle capacità dei singoli livelli, quanto più dai punti in cui questi si collegano e dai tempi di risposta (Sezione 5.7). Gli errori rilevati dal bridge restano all'interno del log, non visibili all'osservazione dell'utente perché non vengono comunicati al backend e di conseguenza non possono essere mostrati nella dashboard. La chiamata al modello, eseguita per ricevere il messaggio di conferma, si effettua senza che il piano d'azione appena generato venga riportato. A causa di questo, in 14 esecuzioni su 15, pur trattandosi di comandi con un piano corretto ed eseguibile, la conferma restituita all'utente contraddice il piano. A questi aspetti si aggiunge anche il fattore della latenza. L'esecuzione di un comando d'azione impegna il sistema per circa 174 s, e di questi sono stati misurati 102 s come tempo di attesa prima che il piano venga inviato al bridge, mentre la navigazione richiede in mediana non più di 35 s di tempo simulato per essere completata. Da questo risultato si evince che la latenza di esecuzione dipende quasi interamente dall'inferenza e, come mostrano le misure della Sezione 5.6, è dominata

dal tempo necessario per l'elaborazione dei prompt da parte del modello, in misura sensibilmente maggiore rispetto alla generazione delle risposte.

6.2 Contributo personale

Questa tesi descrive l'intera architettura del sistema HIMA ed è stata sviluppata nell'ambito di un progetto già avviato. Qui di seguito si descrive il flusso di lavoro ed il contributo personale apportato all'intero progetto, mentre le altre parti del sistema, compresa l'esecuzione delle azioni nel bridge della versione finale, non fanno parte di questo lavoro.

In una fase preliminare è stata svolta un'analisi allo scopo di osservare il percorso di un comando nella versione precedente del sistema (Sezione 4.7.1). In quella versione il bridge aveva un ruolo diverso e inoltrava i piani all'esecutore basato su behavior tree e, pur disponendo di un client per l'action di navigazione, non lo utilizzava. L'analisi ha permesso di individuare questo punto di collegamento mancante nel passaggio del piano generato dal modello a Nav2 e in questo senso ha messo in evidenza i requisiti per la costruzione di un nodo dedicato alla navigazione.

In seguito alle osservazioni effettuate è stato possibile iniziare lo sviluppo del nodo di navigazione, passando per due diverse versioni, presentate nelle Sezioni 4.7.2 e 4.7.3. Il primo prototipo `simple_navigator` verifica l'invio di un goal a Nav2 attraverso l'interfaccia action di ROS 2 e ne attende l'esito in maniera bloccante. La versione successiva, `hima_navigator`, è un nodo che permette di inserire la navigazione all'interno del flusso dei piani d'azione. Il nodo segue l'avanzamento del goal senza bloccarsi, accetta destinazioni espresse per nome o come coordinate ed effettua un controllo sullo stato finale della navigazione riportato da Nav2, ma esegue soltanto la prima navigazione indicata nel piano. Nella versione finale i due nodi non sono utilizzati, tuttavia il confronto della Sezione 4.7.4 mostra che il controllo dello stato finale e le destinazioni in coordinate mancano nel bridge. Tali aspetti verranno ripresi nella Sezione 6.3.

La terza attività svolta riguarda la configurazione dell'ambiente ed alcune correzioni della versione precedente, che sono state necessarie per eseguire il sistema sulla macchina disponibile. Per prima cosa è stato necessario collegare il backend, che viene eseguito in WSL2, con LM Studio, che invece gira in locale su Windows. Tale collegamento è stato realizzato utilizzando l'indirizzo con cui l'host è visibile al sottosistema, non essendo l'indirizzo locale raggiungibile. Inoltre, in assenza di webcam e in generale di un robot che disponesse di una propria telecamera, sono state apportate delle modifiche al modulo di visione in modo tale che questo restituisse in maniera continua un'immagine nera. Il backend, che in origine inviava un'immagine vuota che il server rifiutava, ha avuto in sostituzione un'immagine nera di ripiego e questa soluzione è stata mantenuta anche nella configurazione in cui sono stati eseguiti i test (Sezione 5.2). Infine è stato verificato l'instradamento dei piani, che in quella versione dovevano raggiungere il bridge sulla porta 9091 e non direttamente il convertitore sulla porta 9090.

La quarta attività svolta riguarda l'impostazione e l'esecuzione delle prove per la validazione descritte nel Capitolo 5. Questa comprende la scelta della separazione in due blocchi operativi al fine di testare i due livelli in maniera distinta, la definizione della configurazione di prova e il confronto con quella in uso nella versione finale, la definizione degli scenari, le metriche e i vari criteri di successo, compreso il doppio criterio adottato per la navigazione, insieme infine alla definizione degli script di misura e all'analisi dei risultati.

6.3 Sviluppi futuri

In questa sezione sono disposte, sulla base dei risultati ottenuti nel Capitolo 5, una serie di proposte per sviluppi futuri del progetto, le quali sono ordinate per impegno crescente. Si inizia dagli interventi sul codice, che non richiedono nuovo hardware, successivamente quelli sulla macchina di esecuzione e sul VLM impiegato, l'estensione delle capacità del robot ed infine una modalità di validazione più ampia.

6.3.1 Interventi sul codice

Un primo intervento potrebbe essere effettuato nel bridge, al fine di integrare nella navigazione un controllo dello stato finale restituito da Nav2, come già presente nel nodo `hima_navigator`, e prevedere una cancellazione del goal nel caso in cui si eccedano i tempi massimi, in modo che una navigazione interrotta non venga registrata come completata (Sezione 4.6.3). Dallo stesso nodo il bridge potrebbe riprendere anche la possibilità di accettare destinazioni espresse come coordinate (Sezione 4.7.4). Inoltre si osserva che lo stato riportato da Nav2 al termine dell'azione di navigazione risulta insufficiente per le destinazioni all'interno della zona di ingombro di un ostacolo, queste azioni vengono dichiarate riuscite con il robot a circa 0,3 m dal luogo richiesto. Sarebbe opportuno inserire un confronto tra la posizione raggiunta e quella richiesta, definendo una soglia limite (Sezione 5.7.1). Un'altra proposta riguarda la risalita dell'esito dell'azione nella catena di comando. Nella versione finale la connessione tra i due componenti si chiude appena il piano viene ricevuto (Sezione 4.6.1). Se si inserisse la possibilità per il bridge di inviare al backend un messaggio inerente all'esito di ogni azione, l'utente verrebbe informato anche quando Nav2 non è disponibile o il luogo richiesto è inesistente.

Un secondo intervento riguarda invece il backend. La conferma potrebbe essere generata a partire dal piano prodotto e senza effettuare una seconda chiamata al modello. Questa modifica eliminerebbe la contraddizione tra piano e conferma, oltre a permettere di risparmiare oltre un terzo del tempo di un comando d'azione (Sezione 5.7.2). L'inserimento di una verifica delle destinazioni, sulla base del file `locations.json` e con le stesse regole definite per il bridge, permetterebbe di avvisare subito l'utente quando un luogo non esiste. Alla base di questo problema c'è il disallineamento tra l'elenco dei luoghi dichiarato al modello e quello usato dal bridge (Sezione 4.2.4). Inserire la stessa modalità di ricavare l'elenco dal file dei luoghi anche nel backend potrebbe quindi eliminare le destinazioni proposte al modello, ma inesistenti.

6.3.2 Hardware ed esecuzione del modello

Dai dati misurati emerge che la latenza dipende prevalentemente dall'elaborazione dei prompt, più che dalla generazione delle risposte (Sezione 5.6). La tecnica più efficace, che permetterebbe di ottenere una riduzione netta della latenza, sarebbe l'impiego di una scheda grafica dedicata, che però è assente nella macchina utilizzata. Senza intervenire sull'hardware, la latenza potrebbe essere ridotta anche accorciando i prompt, e quindi il numero di token da elaborare a ogni comando (Sezione 5.7.3). Un hardware adeguato permetterebbe inoltre di testare modelli di dimensioni maggiori, come Qwen2.5-VL-7B (Capitolo 2), e quindi di distinguere i limiti dell'architettura da quelli del modello impiegato (Sezione 5.8).

6.3.3 Estensione delle capacità

Il TurtleBot3 utilizzato in simulazione è sprovvisto di una telecamera e di un braccio e rappresenta quindi una piattaforma di base con capacità limitate. La possibilità di utilizzare un robot dotato di

telecamera, sia in ambiente di simulazione che nel reale, renderebbe effettive le azioni di esplorazione, attualmente limitate ad una rotazione sul posto (Sezione 4.6.4). Si avrebbe la possibilità di collegare l'esplorazione all'analisi delle immagini e di comprendere l'immagine catturata dal robot nel materiale alla base della generazione del piano d'azione. Lo stesso ragionamento vale per le azioni di presa e di deposito, le quali sono presenti nei piani, ma non possono produrre effetti sul robot. Per testarle è necessario avere a disposizione una piattaforma mobile con un manipolatore integrato ed i servizi corrispondenti. Il candidato più prossimo è il robot Stretch, per il quale è stata avviata una preliminare fase di configurazione per la navigazione nel simulatore MuJoCo, non portata a termine. Il passo successivo sarebbe la prova su un robot reale.

Per l'esecuzione dei piani si potrebbe riattivare il pacchetto `hima_bt_runner`, presente nel progetto ma non in uso nella versione finale (Sezione 4.6.6). Il suo esecutore basato su behavior tree esegue i piani uno alla volta, interrompe la sequenza al primo fallimento e pubblica l'esito di ogni passo. Collegando questi eventi al backend, si potrebbero affrontare insieme i piani concorrenti (Sezione 4.6.5), i piani che proseguono dopo un errore (Sezione 4.6.2) e la risalita dell'esito nella catena di comando.

6.3.4 Validazione

Il prossimo passo per ottenere una validazione del sistema più completa sarebbe la verifica della catena di comando completa, dalla dashboard al robot con tutti i componenti attivi, la quale non è inclusa nella presente trattazione. Al fine di ottenere un'indagine statistica sui dati presentati, che permetta di osservare i risultati ottenuti in maniera più precisa, risulta necessario effettuare un maggior numero di ripetizioni per i test. I comandi potrebbero comprendere una formulazione più articolata, in modo da osservare le capacità del modello quando si presentano richieste strutturate in maniera meno semplice. Nelle prove di navigazione sarebbe opportuno utilizzare la posa reale del robot, fornita dal simulatore, e mettere alla prova la localizzazione in scenari con una posa iniziale errata o una mappa non aggiornata. Infine un ultimo aspetto da includere nella validazione sono le prove con la presenza di persone, in modo da valutare l'interazione del sistema all'interno di un contesto assistivo, scopo per il quale è stato progettato. La valutazione, all'interno di un contesto assistivo, è importante per osservare quanto le risposte del modello risultano chiare ed i tempi di risposta accettabili

6.4 Considerazioni conclusive

I risultati mostrano che i due livelli di HIMA funzionano in simulazione anche su una macchina priva di scheda grafica. Un modello di piccole dimensioni eseguito in locale genera piani d'azione corretti per i comandi previsti, anche se con tempi di risposta elevati, e ROS 2 e Nav2 eseguono in modo affidabile le navigazioni richieste. I limiti individuati non riguardano il funzionamento dei singoli componenti, che nelle prove dedicate hanno dimostrato entrambi un buon livello di affidabilità, ma la loro integrazione e i tempi di risposta. Essendo andati a buon fine, in quasi tutti gli scenari previsti, nessuno dei limiti osservati impedisce ai due livelli di svolgere il proprio compito. Resta da verificare con prove sistematiche il funzionamento della catena completa, dalla dashboard al robot. Affinché HIMA possa essere utilizzabile in un contesto assistivo, gli interventi devono essere mirati ad ottimizzare la trasparenza della catena di comando del sistema nei confronti dell'utente, e prevedere che sia riportato l'esito delle azioni all'utente in modo coerente con ciò che il robot esegue. Il secondo aspetto da curare riguarda la latenza ed i tempi di attesa, che devono scendere a valori compatibili con un'interazione di supporto per l'utente.

Bibliografia

N. Riferimento

[1] Istat, «Indicatori demografici – Anno 2025», Statistica report, Roma, 31 mar. 2026. [Online]. Disponibile: https://www.istat.it/wp-content/uploads/2026/03/Report_Indicatori-demografici_Anno-2025.pdf (consultato il 5 ott. 2026).

[2] Istat, «Previsioni della popolazione residente, delle famiglie e delle forze di lavoro – Base 1/1/2025», comunicato stampa, Roma, 31 ago. 2026. [Online]. Disponibile: https://www.istat.it/comunicato-stampa/previsioni-della-popolazione-residente-delle-famiglie-e-delle-forze-di-lavoro-base-1-1-2025/ (consultato il 5 ott. 2026).

[3] Istat, «Le condizioni di salute della popolazione anziana in Italia – Anno 2019», Statistica report, Roma, 14 lug. 2021. [Online]. Disponibile: https://www.istat.it/it/files/2021/07/Report-anziani-2019.pdf (consultato il 5 ott. 2026).

[4] OECD, *Who Cares? Attracting and Retaining Care Workers for the Elderly* (OECD Health Policy Studies). Parigi, Francia: OECD Publishing, 2020, doi: 10.1787/92c0ef68-en.

[5] D. Feil-Seifer e M. J. Matarić, «Defining socially assistive robotics», in *Proc. 9th Int. Conf. Rehabil. Robot. (ICORR)*, Chicago, IL, USA, 2005, pp. 465–468, doi: 10.1109/ICORR.2005.1501143.

[6] P. Asgharian, A. M. Panchea e F. Ferland, «A review on the use of mobile service robots in elderly care», *Robotics*, vol. 11, n. 6, art. 127, 2022, doi: 10.3390/robotics11060127.

[7] Istat, «Cittadini e ICT – Anno 2025», comunicato stampa, Roma, 22 apr. 2026. [Online]. Disponibile: https://www.istat.it/comunicato-stampa/cittadini-e-ict-anno-2025/ (consultato il 5 ott. 2026).

[8] R. Shao *et al.*, «Large VLM-based vision-language-action models for robotic manipulation: A survey», 2025, arXiv:2508.13073, doi: 10.48550/arXiv.2508.13073.

[9] B. Ichter *et al.*, «Do as I can, not as I say: Grounding language in robotic affordances», in *Proc. 6th Conf. Robot Learn. (CoRL)*, in Proc. Mach. Learn. Res., vol. 205, 2023, pp. 287–318.

[10] Z. Yu *et al.*, «A survey on efficient vision-language-action models», 2025, arXiv:2510.24795, doi: 10.48550/arXiv.2510.24795.

[11] F. Bordes *et al.*, «An introduction to vision-language modeling», 2024, arXiv:2405.17247, doi: 10.48550/arXiv.2405.17247.

[12] A. Vaswani *et al.*, «Attention is all you need», in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, vol. 30, Long Beach, CA, USA, 2017, pp. 5998–6008.

[13] W. X. Zhao *et al.*, «A survey of large language models», 2023, arXiv:2303.18223, doi: 10.48550/arXiv.2303.18223.

[14] Google DeepMind, «Gemma 4 model card», Google AI for Developers. [Online]. Disponibile: https://ai.google.dev/gemma/docs/core/model_card_4 (consultato il 18 set. 2026).

[15] A. Radford *et al.*, «Learning transferable visual models from natural language supervision», in *Proc. 38th Int. Conf. Mach. Learn. (ICML)*, in Proc. Mach. Learn. Res., vol. 139, 2021, pp. 8748–8763.

[16] S. Bai *et al.*, «Qwen2.5-VL technical report», 2025, arXiv:2502.13923, doi: 10.48550/arXiv.2502.13923.

[17] H. Liu, C. Li, Q. Wu e Y. J. Lee, «Visual instruction tuning», in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, vol. 36, New Orleans, LA, USA, 2023.

[18] E. J. Hu *et al.*, «LoRA: Low-rank adaptation of large language models», in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2022.

[19] Qwen Team, «Qwen2.5-VL-7B-Instruct», Hugging Face. [Online]. Disponibile: https://huggingface.co/Qwen/Qwen2.5-VL-7B-Instruct (consultato il 18 set. 2026).

[20] Gemma Team, «Gemma 4 technical report», 2026, arXiv:2607.02770, doi: 10.48550/arXiv.2607.02770.

[21] LM Studio, «Developer docs». [Online]. Disponibile: https://lmstudio.ai/docs/developer (consultato il 21 set. 2026).

[22] S. Macenski, T. Foote, B. Gerkey, C. Lalancette e W. Woodall, «Robot Operating System 2: Design, architecture, and uses in the wild», *Sci. Robot.*, vol. 7, n. 66, art. eabm6074, 2022, doi: 10.1126/scirobotics.abm6074.

[23] M. Quigley *et al.*, «ROS: An open-source Robot Operating System», in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Giappone, 2009.

[24] ROS 2 Documentation (Humble), «Different ROS 2 middleware vendors». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Different-Middleware-Vendors.html (consultato il 21 set. 2026).

[25] ROS 2 Documentation (Humble), «Client libraries». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Basic/About-Client-Libraries.html (consultato il 21 set. 2026).

[26] ROS 2 Documentation (Humble), «Quality of Service settings». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html (consultato il 21 set. 2026).

[27] ROS 2 Documentation, «Distributions». [Online]. Disponibile: https://docs.ros.org/en/rolling/Releases.html (consultato il 21 set. 2026).

[28] ROS 2 Documentation (Humble), «Nodes». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Basic/About-Nodes.html (consultato il 21 set. 2026).

[29] ROS 2 Documentation (Humble), «Topics». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html (consultato il 21 set. 2026).

[30] ROS 2, «common_interfaces» (ramo humble), GitHub. [Online]. Disponibile: https://github.com/ros2/common_interfaces (consultato il 21 set. 2026).

[31] T. Foote, «tf: The transform library», in *Proc. IEEE Int. Conf. Technol. Pract. Robot Appl. (TePRA)*, Woburn, MA, USA, 2013, pp. 1–6, doi: 10.1109/TePRA.2013.6556373.

[32] ROS 2 Documentation (Humble), «Tf2». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Tf2.html (consultato il 21 set. 2026).

[33] W. Meeussen, «REP 105: Coordinate frames for mobile platforms», ROS Enhancement Proposals, 2010. [Online]. Disponibile: https://www.ros.org/reps/rep-0105.html (consultato il 21 set. 2026).

[34] ROS 2 Documentation (Humble), «Services». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html (consultato il 21 set. 2026).

[35] ROS 2 Documentation (Humble), «Actions». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html (consultato il 21 set. 2026).

[36] ROS 2 Documentation (Humble), «Executors». [Online]. Disponibile: https://docs.ros.org/en/humble/Concepts/Intermediate/About-Executors.html (consultato il 21 set. 2026).

[37] ROS 2 Documentation (Humble), «Writing an action server and client (Python)». [Online]. Disponibile: https://docs.ros.org/en/humble/Tutorials/Intermediate/Writing-an-Action-Server-Client/Py.html (consultato il 21 set. 2026).

[38] ROS 2 Documentation (Humble), «Using callback groups». [Online]. Disponibile: https://docs.ros.org/en/humble/How-To-Guides/Using-callback-groups.html (consultato il 21 set. 2026).

[39] S. Macenski, T. Moore, D. V. Lu, A. Merzlyakov e M. Ferguson, «From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the Robot Operating System 2», *Robot. Auton. Syst.*, vol. 168, art. 104493, 2023.

[40] S. Macenski e I. Jambrecic, «SLAM Toolbox: SLAM for the dynamic world», *J. Open Source Softw.*, vol. 6, n. 61, art. 2783, 2021, doi: 10.21105/joss.02783.

[41] S. Macenski, F. Martín, R. White e J. Ginés Clavero, «The Marathon 2: A navigation system», in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Las Vegas, NV, USA, 2020, pp. 2718–2725, doi: 10.1109/IROS45743.2020.9341207.

[42] S. Macenski, S. Singh, F. Martín e J. Ginés, «Regulated pure pursuit for robot path tracking», *Auton. Robots*, vol. 47, n. 6, pp. 685–694, 2023, doi: 10.1007/s10514-023-10097-6.

[43] Nav2, «Nav2 Documentation». [Online]. Disponibile: https://docs.nav2.org (consultato il 23 set. 2026).

[44] N. Koenig e A. Howard, «Design and use paradigms for Gazebo, an open-source multi-robot simulator», in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sendai, Giappone, 2004, pp. 2149–2154, doi: 10.1109/IROS.2004.1389727.

[45] E. Todorov, T. Erez e Y. Tassa, «MuJoCo: A physics engine for model-based control», in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Vilamoura, Portogallo, 2012, pp. 5026–5033, doi: 10.1109/IROS.2012.6386109.

[46] ROBOTIS, «TurtleBot3 e-Manual: Features». [Online]. Disponibile: https://emanual.robotis.com/docs/en/platform/turtlebot3/features (consultato il 23 set. 2026).

[47] C. C. Kemp, A. Edsinger, H. M. Clever e B. Matulevich, «The design of Stretch: A compact, lightweight mobile manipulator for indoor human environments», in *Proc. IEEE Int. Conf.*

Robot. Autom. (ICRA), Filadelfia, PA, USA, 2022, pp. 3150–3157, doi: 10.1109/ICRA46639.2022.9811922.

[48] FastAPI, «FastAPI documentation». [Online]. Disponibile: https://fastapi.tiangolo.com (consultato il 5 ott. 2026).

[49] React, «React documentation». [Online]. Disponibile: https://react.dev (consultato il 5 ott. 2026).

[50] Microsoft, «What is Windows Subsystem for Linux?», Microsoft Learn. [Online]. Disponibile: https://learn.microsoft.com/windows/wsl/about (consultato il 5 ott. 2026).

[51] OpenCV, «OpenCV documentation». [Online]. Disponibile: https://docs.opencv.org (consultato il 5 ott. 2026).

[52] pyttsx3, «pyttsx3 documentation». [Online]. Disponibile: https://pyttsx3.readthedocs.io (consultato il 5 ott. 2026).

[53] Vite, «Vite documentation». [Online]. Disponibile: https://vite.dev (consultato il 5 ott. 2026).

[54] W. Hess, D. Kohler, H. Rapp e D. Andor, «Real-time loop closure in 2D LIDAR SLAM», in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, Stoccolma, Svezia, 2016, pp. 1271–1278, doi: 10.1109/ICRA.2016.7487258.

[55] S. Thrun, W. Burgard e D. Fox, *Probabilistic Robotics* (Intelligent Robotics and Autonomous Agents). Cambridge, MA, USA: MIT Press, 2005.

[56] M. Colledanchise e P. Ögren, *Behavior Trees in Robotics and AI: An Introduction* (Chapman & Hall/CRC Artificial Intelligence and Robotics Series). Boca Raton, FL, USA: CRC Press, 2018, doi: 10.1201/9780429489105.