



UNIVERSITÀ DEGLI STUDI DI PALERMO

Dipartimento di Ingegneria

CORSO DI Laurea Magistrale in Ingegneria Informatica
Indirizzo Intelligenza Artificiale

Taxi Autonomo

Simulazione Robotica con
Interazione Intelligente

STUDENTI

Davì Ernesto
Dioguardi Roberto
Ferrara Salvatore
Spezia Salvatore

DOCENTI

Prof. Antonio Chella
(*Robotica*)

Prof.ssa Valeria Seidita
(*Intelligenza Artificiale 2*)

Anno Accademico 2025/2026

Contents

1	Introduzione	4
1.1	Descrizione del Sistema	4
1.2	Architettura del Sistema	4
2	Agente Taxi Autonomo	6
2.1	Architettura Modulare	6
2.2	Flusso Informativo	7
2.3	Brain Layer	7
2.3.1	Perception Module	8
2.3.2	Navigation Module	9
2.3.3	Traffic Module	9
2.4	Sensor Layer	10
2.4.1	SimpleLidar	11
2.4.2	Modello cinematico di Ackermann	11
2.4.3	Classificazione dinamica degli ostacoli	11
2.4.4	Configurazione dual-LiDAR	11
2.5	Actuator Layer	12
2.5.1	CarMotor	12
2.5.2	CarBattery	13
2.6	Planning Layer	15
2.6.1	Waypoint Graph	15
2.6.2	Calcolo del percorso	15
2.6.3	Pathfinding dinamico pesato	15
2.7	Mission Layer	15
2.7.1	Macchina a Stati Finiti	16
2.7.2	Gestione delle prenotazioni	17
2.7.3	Integrazione con la Knowledge Base	17
3	Ambiente Dinamico della Simulazione	18
3.1	Traffico Veicolare	18
3.1.1	FSM	18
3.1.2	Navigazione su circuito	18
3.1.3	Sensore volumetrico	19
3.1.4	Frenata progressiva	19
3.1.5	Interazione con i semafori (V2I)	19
3.2	Pedoni	19
3.2.1	FSM	19
3.2.2	Sistema di personalità	20
3.2.3	Navigazione NavMesh	20
3.2.4	Logica di attraversamento	21
3.2.5	Interazione con il taxi	21
3.3	Semafori Intelligenti	21
3.3.1	Ciclo semaforico	21
3.3.2	Comunicazione V2I	22
3.3.3	Feedback visivo	22
3.3.4	Gestione guasti	22
3.4	Barriere Dinamiche	22

3.4.1	Ciclo di vita	23
3.4.2	Rilevamento dal taxi	23
3.4.3	Integrazione con il pathfinding	23
3.5	Sistema Meteorologico	24
3.5.1	Variazione delle condizioni meteo	24
3.5.2	Impatto sulla navigazione	24
3.6	Sistema temporale	24
3.6.1	Mappatura tempo reale–tempo simulato	24
3.6.2	Controllo del tempo	24
3.6.3	Integrazione con la knowledge base	25
3.7	Suddivisione in zone	25
3.7.1	Implementazione in Unity	25
3.7.2	Zone definite	25
4	Ontologia e Inferenza	27
4.1	Modellazione Ontologica	27
4.1.1	Gerarchia delle classi	27
4.1.2	Analisi delle relazioni	29
4.2	Implementazione della Base di Conoscenza	30
4.2.1	Schema Core: Ricerca e Personalizzazione	30
4.2.2	Schema Environment: Policy e Zone	32
5	L'algoritmo A*	34
5.1	Architettura del Grafo di Navigazione	34
5.2	Implementazione dell'algoritmo A*	34
5.2.1	Euristica utilizzata	34
5.2.2	Strutture dati	35
5.2.3	Pseudocodice	35
5.3	Fattori di Influenza sul Pathfinding	36
5.3.1	Policy di Guida	36
5.3.2	Condizioni meteorologiche	37
5.3.3	Eventi temporali	37
5.3.4	ZTL (Zona a Traffico Limitato)	38
5.4	Formula del costo dell'arco	38
5.5	Moltiplicatore combinato	38
6	Chat UI	40
6.1	Architettura di comunicazione	40
6.2	Booking (Pre-ride)	40
6.2.1	Selezione destinazione	40
6.2.2	Prenotazione corsa	41
6.2.3	Attesa in coda	42
6.2.4	Annulloamento prenotazione	43
6.3	Interazioni durante la corsa	44
6.3.1	Cambio destinazione	44
6.3.2	Fine corsa	44
6.4	Voice AI	45
6.4.1	Speech-to-Text (STT)	45
6.4.2	Text-to-Speech (TTS)	45

6.5	Music Streaming	46
6.5.1	Riproduzione musicale	46
6.5.2	Cambio di genere e controlli	46
6.6	Policy di guida	46
6.6.1	Selezione pre-ride	47
6.6.2	Modifica durante la corsa	47
6.6.3	Cambi automatici e explainability	47
6.7	Impostazioni LLM e Voce	47
7	LLM e Tooling Conversazionale	48
7.1	LLM: modelli e parametri	48
7.2	Strategia di Prompting	48
7.2.1	Filtri lessicali di pre-classificazione	48
7.2.2	Classificazione dei Tools	49
7.2.3	Classificazione dei Bisogni	50
7.2.4	Richiesta Fuori Contesto	51
7.2.5	Supporto alla selezione dell'opzione	52
7.3	Pipeline di classificazione delle richieste utente	52
7.4	POI Tools	53
7.4.1	Formula di ranking dei POI	53
7.4.2	Tool di ricerca del POI tramite il bisogno	54
7.4.3	Tool di ricerca dei POI tramite tag	54
7.4.4	Tool di ricerca POI per nome	55
7.5	Music Tools	56
7.6	Tool della modifica dello stile di guida	56
7.7	Gestione dei messaggi fuori contesto	57
7.7.1	Classificazione del bisogno	57
7.7.2	Risposta conversazionale	57
7.8	Explainability	57
7.8.1	Flusso di comunicazione dell'explainability	57
7.8.2	Messaggi dell'explainability	58

1 Introduzione

I veicoli autonomi rappresentano una delle principali evoluzioni dei sistemi di trasporto intelligenti, con l’obiettivo di migliorare sicurezza, efficienza e qualità dei servizi di mobilità. In questo contesto, l’interazione uomo-macchina riveste un ruolo fondamentale, soprattutto nei servizi rivolti direttamente all’utente finale, come i taxi autonomi, dove chiarezza e semplicità di utilizzo risultano essenziali. Il presente progetto ha come obiettivo la simulazione di un servizio di taxi autonomo dotato di un’interfaccia conversazionale intelligente, finalizzata a consentire un’interazione naturale tra utente e sistema all’interno di un ambiente simulato.

1.1 Descrizione del Sistema

Il sistema offre un’esperienza di mobilità completamente autonoma basata su un’interazione conversazionale naturale. Attraverso un’interfaccia di chat accessibile da smartphone, il passeggero può interagire con il sistema tramite input testuali o comandi vocali, formulando richieste in linguaggio naturale, selezionando la destinazione, modificando o terminando la corsa in qualsiasi momento e scegliendo lo stile di guida del veicolo. In caso di indisponibilità immediata del taxi, il sistema consente all’utente di mettersi in coda, gestendo in modo ordinato le richieste. Durante il viaggio, il sistema è in grado di fornire raccomandazioni personalizzate, di gestire funzionalità accessorie come il controllo della riproduzione musicale e di offrire spiegazioni trasparenti sulle decisioni adottate, quali il ricalcolo del percorso o i suggerimenti proposti. Il veicolo autonomo opera all’interno di una città simulata, navigando in modo autonomo nel rispetto delle regole del traffico e adattando dinamicamente il percorso e il comportamento di guida in base a condizioni variabili come traffico e meteo.

1.2 Architettura del Sistema

L’architettura del sistema è di tipo modulare e distribuito, ed è stata progettata per separare in modo chiaro le responsabilità tra le diverse componenti software. Al centro vi è un backend sviluppato in Python, che svolge il ruolo di nodo di coordinamento e di instradamento della comunicazione tra i vari moduli.

Il backend dialoga con l’ambiente di simulazione realizzato in Unity tramite WebSocket, abilitando uno scambio bidirezionale e in tempo reale di informazioni sullo stato del veicolo, della corsa, della prenotazione e sulle richieste provenienti dall’utente.

L’interazione con l’utente è gestita tramite una Chat UI implementata come applicazione web in HTML, anch’essa connessa al backend attraverso WebSocket. Questa scelta consente una comunicazione asincrona e immediata, permettendo all’utente di inviare comandi testuali o vocali e di ricevere le risposte del sistema in tempo reale.

Per la gestione della conversazione e l’interpretazione del linguaggio naturale, il backend si interfaccia con un LLM esposto tramite le API di OpenRouter. Il modello viene impiegato per comprendere le richieste dell’utente e classificarle, così da attivare in modo selettivo i tool più appropriati alla gestione della specifica richiesta.

Come base di conoscenza centralizzata il sistema adotta Neo4j, un database a grafo. In esso sono memorizzati i profili utente con preferenze e storico delle visite, i punti di interesse della città con relative categorie e tag, oltre ai moltiplicatori di costo associati alle diverse zone stradali in funzione delle policy di guida e delle condizioni meteo. La rap-

presentazione a grafo consente di eseguire query relazionali complesse, utili per produrre raccomandazioni personalizzate e per calcolare percorsi ottimali.

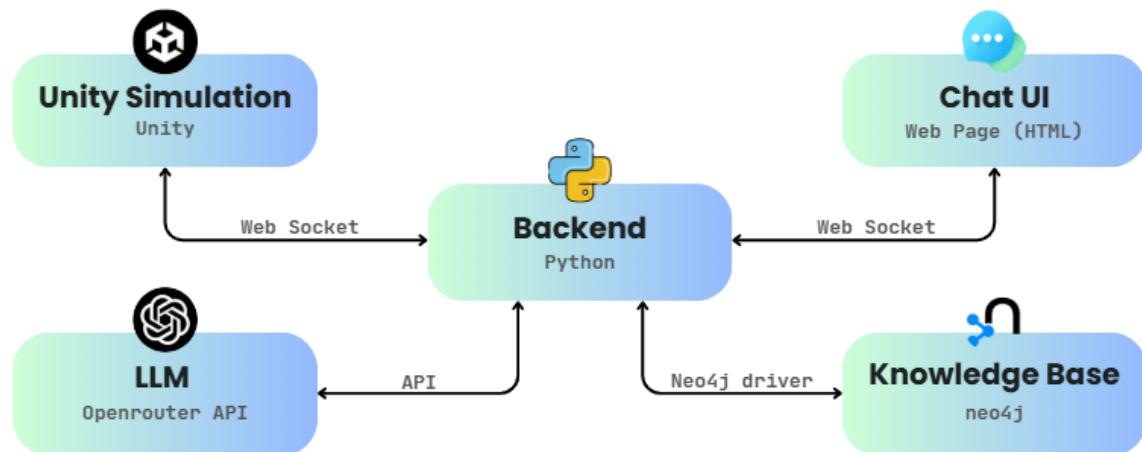


Figure 1: Architettura generale del sistema

2 Agente Taxi Autonomo

L'ambiente di simulazione Unity riproduce una città virtuale in cui opera il taxi autonomo. La scena include strade con semafori intelligenti, automobili¹ che generano traffico realistico, pedoni². Il taxi è modellato come un agente autonomo che percepisce l'ambiente attraverso sensori simulati, pianifica percorsi ottimali e reagisce in tempo reale a ostacoli e imprevisti.

2.1 Architettura Modulare

Il veicolo autonomo è implementato come un agente cognitivo multi-layer che integra percezione, pianificazione e attuazione secondo un'architettura gerarchica ispirata ai principi della robotica autonoma. Questa struttura modulare garantisce separazione delle responsabilità, manutenibilità del codice e possibilità di evoluzione indipendente dei singoli componenti. L'architettura si articola su cinque livelli funzionali distinti, ciascuno con responsabilità ben definite come mostrato nella figura 2.

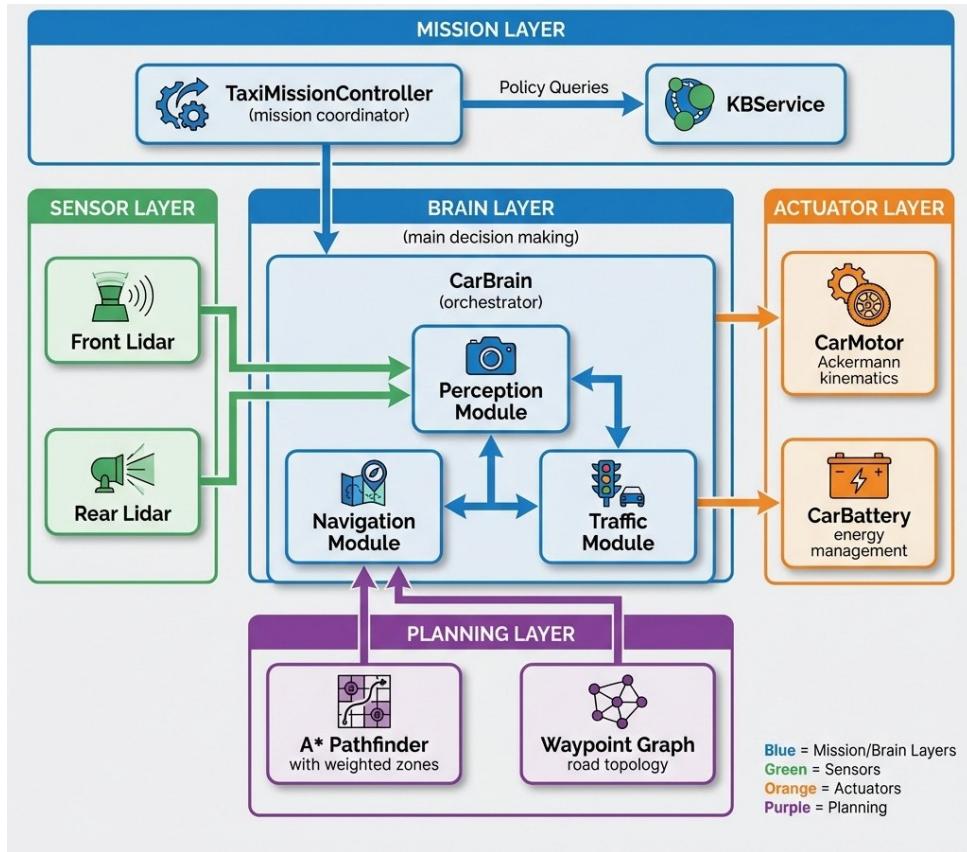


Figure 2: Architettura modulare Agente Taxi Autonomo

¹**TrafficCarAI**: è un agente reattivo che tramite una macchina a stati finiti simula il comportamento di un veicolo nel traffico urbano. Il veicolo naviga lungo un percorso predefinito (waypoint circuit) e reagisce dinamicamente a vincoli fisici (ostacoli, pedoni, automobili) e normativi (semafori).

²**PedestrianAI**: è un agente autonomo NavMesh-based dotato di personalità comportamentale. Il pedone naviga sui marciapiedi e decide se attraversare la strada basandosi sul proprio profilo di rischio e sulla percezione del traffico.

2.2 Flusso Informativo

L'architettura segue un pattern *Sense-Think-Act* tipico dei sistemi robotici:

1. **Sense (Percezione):** I sensori LiDAR acquisiscono informazioni sull'ambiente circostante, identificando ostacoli, pedoni e veicoli del traffico; i dati grezzi vengono strutturati in settori semantici (frontale, laterale, pedonale).
2. **Think (Elaborazione):** Il Brain Layer riceve i dati sensoriali e li elabora attraverso tre moduli specializzati:
 - il *Perception Module* interpreta gli stimoli e calcola vincoli di velocità;
 - il *Navigation Module* determina la traiettoria e l'angolo di sterzo;
 - il *Traffic Module* gestisce l'interazione V2I (*Vehicle-to-Infrastructure*) con i semafori intelligenti;
3. **Act (Attuazione):** I comandi calcolati vengono inviati agli attuatori, dove il *Car-Motor* applica la cinematica del veicolo e il *CarBattery* monitora il consumo energetico.

A livello superiore, il *Mission Layer* definisce gli obiettivi strategici (pickup passeggero, drop-off, ricarica) e delega l'esecuzione tattica al Brain Layer. Il *Planning Layer* fornisce i percorsi ottimali calcolati con l'algoritmo A*, considerando la topografia stradale e i pesi delle zone urbane.

2.3 Brain Layer

Il Brain Layer costituisce il nucleo decisionale del taxi autonomo. È composto da un orchestratore centrale (*CarBrain*) e tre moduli specializzati che gestiscono rispettivamente la percezione dell'ambiente, la navigazione e l'interazione con l'infrastruttura semaforica. Le sue responsabilità includono l'inizializzazione e il collegamento dei sottosistemi (moduli, sensori, attuatori), l'esecuzione della pipeline decisionale ad ogni frame della simulazione con una frequenza di circa 30 esecuzioni al secondo, la gestione della logica di emergenza (retromarcia anti-stallo) e il rilevamento delle zone urbane per il pathfinding pesato.

Pipeline decisionale

Ad ogni ciclo il *CarBrain* esegue una sequenza ordinata di operazioni:

1. **Aggiornamento percezione:** percezione dell'ambiente e decremento dei timer di cooldown per le rilevazioni sensoriali.
2. **Gestione energetica:** verifica dello stato di ricarica e del consumo batteria basato sulla distanza percorsa. Se la batteria è esaurita o in ricarica, il veicolo si arresta.
3. **Logica anti-stallo:** se il veicolo è fermo in maniera non intenzionale (per esempio se bloccato da un ostacolo) per un tempo superiore alla soglia configurata, viene attivata una manovra di retromarcia con controllo di sicurezza tramite LiDAR posteriore.

4. **Calcolo velocità target:** la velocità obiettivo viene determinata applicando tre vincoli concorrenti, ovvero il vincolo semaforico (limite imposto dal *Traffic Module*), il vincolo di curvatura (rallentamento predittivo e reattivo in curva) e il vincolo di percezione (limite basato su ostacoli e pedoni rilevati); la velocità finale è il minimo tra tutti i limiti, garantendo il rispetto del vincolo più restrittivo.
5. **Calcolo sterzo:** il *Navigation Module* determina l'angolo di sterzata considerando l'offset laterale (per manovre di sorpasso) e la direzione del prossimo waypoint.
6. **Attuazione:** i comandi vengono inviati al *CarMotor* per l'applicazione della cinematica.

Gestione delle zone urbane

Il *CarBrain* rileva automaticamente l'ingresso e l'uscita dalle zone urbane (*CityZone*) tramite dei trigger (collider). Ogni zona è caratterizzata da un nome identificativo, dalla tipologia (residenziale, industriale, ecc.) e da un moltiplicatore di peso per il pathfinding. Queste informazioni vengono utilizzate dall'algoritmo A* per calcolare percorsi che rispettino le policy di guida.

2.3.1 Perception Module

Il modulo di percezione (*CarBrain_Perception*) interpreta i dati grezzi del LiDAR provenienti dal Sensor Layer e li trasforma in vincoli operativi per la guida, ovvero una velocità desiderata e un offset laterale per i sorpassi, che vengono comunicati al *CarBrain*.

Logica di elaborazione

La logica di elaborazione realizza una pipeline di analisi strutturata in tre fasi:

- **Gestione dei pedoni:** i pedoni vengono trattati in modo speciale data la loro vulnerabilità. Viene applicato un filtro laterale che ignora i pedoni oltre una certa soglia (*pedestrianLateralMargin*), considerati sicuri sul marciapiede, e un filtro sulla sterzata che esclude, durante le curve, i pedoni all'esterno della traiettoria del veicolo. In prossimità del passeggero assegnato per il pickup viene inoltre applicata un'eccezione, riducendo la soglia di stop per consentire l'accostamento.
- **Gestione degli ostacoli frontal:** gli oggetti rilevati vengono classificati e trattati con soglie di sicurezza differenziate in base alla loro natura, che può essere statica o dinamica. Gli ostacoli generici usano la distanza *safetyDistance*, i pali o gli oggetti considerati immutabili nell'ambiente (come idranti, alberi, ecc) la *poleSafetyDistance*, i veicoli di traffico la *trafficCarSafetyDistance* e i pedoni la *pedestrianSafetyDistance*. Quando viene rilevato un *RoadBlock* (barriera stradale che impedisce al veicolo di percorrere una specifica strada), il modulo notifica il Mission Controller tramite l'evento *OnRoadBlockDetected*, innescando il ricalcolo del percorso.
- **Logica di sorpasso:** implementa un comportamento con memoria stabilizzata per evitare oscillazioni. Il sorpasso viene considerato solo se sono soddisfatte condizioni di sicurezza, quali spazio laterale sufficiente, assenza di traffico in senso opposto e semaforo non rosso, ed esclude oggetti non sorpassabili come *RoadBlock*, pedoni e veicoli del traffico. Una volta iniziata la manovra, un timer di memoria mantiene

l'offset laterale per un intervallo minimo anche se l'ostacolo non è più visibile, garantendo manovre più fluide e realistiche.

2.3.2 Navigation Module

Il modulo di navigazione (*CarBrain_Navigation*), a partire dalle informazioni ricevute dal *Perception Module* (lista di waypoint, offset target), gestisce il *path following* e calcola l'angolo di sterzo necessario per seguire la traiettoria.

Algoritmo di sterzo

Il calcolo dello sterzo avviene attraverso quattro fasi principali:

- **Gestione dell'offset con *SmoothDamp*:** il cambio di corsia per i sorpassi viene smussato utilizzando `Mathf.SmoothDamp` con un tempo dinamico; l'uscita dalla corsia (offset in aumento) utilizza un tempo breve per garantire reattività, mentre il rientro in corsia (offset in diminuzione) usa un tempo poco maggiore per ottenere una manovra più dolce.
- **Analisi della curvatura predittiva:** il sistema analizza la curvatura della strada nei prossimi 10 metri calcolando l'angolo tra i vettori direzionali del percorso (in base agli specifici waypoints da seguire), classificando come “strette” le curve con angolo superiore a 45°.
- **Look-ahead adattivo:** la distanza a cui il veicolo “guarda” viene calcolata dinamicamente come funzione di velocità e curvatura ($lookAhead = f(velocità, curvatura)$). A velocità alta e su strada dritta il look-ahead è lungo (fino a 15 m), mentre a velocità bassa o in curva è corto (minimo 3 m), permettendo di anticipare le curve senza sacrificare la precisione in manovra.
- **Calcolo dell'angolo target:** l'angolo di sterzo finale viene ottenuto determinando il punto target (waypoint più offset laterale), calcolando l'angolo tra la direzione corrente e quella target, applicando un *clamp* entro i limiti fisici delle ruote e interpolando in modo morbido verso l'angolo desiderato.

Rallentamento predittivo in curva

Il metodo `GetPredictiveCornerFactor` scansiona i prossimi waypoint alla ricerca di curve strette e, quando ne individua una entro la distanza di frenata, restituisce un fattore proporzionale alla distanza residua, consentendo al *CarBrain* di ridurre la velocità gradualmente prima della curva.

2.3.3 Traffic Module

Il modulo traffico (*CarBrain_Traffic*) gestisce la comunicazione V2I (Vehicle-to-Infrastructure) con i semafori intelligenti della città. I semafori sono pensati con una tecnologia 5G, tale per cui riescono a direzionare il segnale solo verso la corsia di appartenenza.

Comunicazione con i semafori

Il modulo implementa un pattern *Observer* per ricevere notifiche sui cambiamenti di stato:

- **Connessione:** quando il taxi entra nel trigger di un semaforo, si sottoscrive all'evento *OnStateChanged*.
- **Filtro direzionale:** i semafori alle spalle del veicolo vengono ignorati tramite controllo sul prodotto scalare (valori negativi indicano oggetti dietro al veicolo)
- **Ricezione stati:** gli stati *Red*, *Yellow*, *Green*, *Off* vengono tracciati internamente.
- **Disconnessione:** all'uscita dal trigger, la sottoscrizione viene rimossa.

Calcolo del limite di velocità

Il metodo `GetTrafficLightSpeedLimit` determina la velocità consentita in base alla posizione rispetto alla linea di stop. Quando il semaforo è verde non viene imposto alcun limite aggiuntivo, mentre in presenza di rosso o giallo e con il veicolo ancora lontano dalla linea viene applicato un rallentamento progressivo. Se il semaforo è rosso o giallo in prossimità della linea di stop, il veicolo effettua un arresto completo; qualora invece si trovi già oltre la linea, non viene applicato alcun ulteriore vincolo, così da permettere al taxi di liberare rapidamente l'incrocio.

Controllo geometrico

Un aspetto critico è il rilevamento del superamento della linea di stop tramite il metodo `IsStopLineBehind`. Dato il vettore dalla posizione del veicolo (`transform.position`) alla linea di stop

$$\text{dirToLine} = \text{stopLine.position} - \text{transform.position}$$

si calcola il prodotto scalare con la direzione di marcia:

$$\text{dot} = \text{Vector3.Dot}(\text{transform.forward}, \text{dirToLine})$$

La funzione restituisce `true` se

$$\text{dot} < 0$$

ossia se la linea di stop si trova geometricamente dietro al veicolo.

Questo controllo impedisce che il veicolo resti bloccato in mezzo all'incrocio o sulle strisce pedonali quando il semaforo diventa rosso, dopo che il taxi ha già superato la linea di stop.

2.4 Sensor Layer

Il *Sensor Layer* è responsabile dell'acquisizione delle informazioni ambientali necessarie al funzionamento del veicolo autonomo. Il componente principale è il *SimpleLidar*, un sensore simulato che replica il comportamento di un LiDAR a scansione radiale.

2.4.1 SimpleLidar

La classe *SimpleLidar* implementa un sensore di percezione che emette raggi in un arco frontale, rilevando ostacoli, pedoni e veicoli nel campo visivo del taxi. L'elaborazione dei raggi produce tre output semanticamente distinti, ciascuno rappresentato dalla struttura *ScanResult*: il *FrontSector* rileva ostacoli sulla traiettoria attuale per gestire frenate e sorpassi, il *LeftSector* verifica lo spazio libero a sinistra per valutare la fattibilità dei sorpassi e il *PedestrianSector* monitora i pedoni in un campo allargato per garantire la massima sicurezza pedonale. Ogni *ScanResult* incapsula la distanza dall'oggetto rilevato, il riferimento al tipo di ostacolo (*GameObject Unity*), la posizione in coordinate locali dell'ostacolo e un flag di validità.

2.4.2 Modello cinematico di Ackermann

Una caratteristica distintiva del *SimpleLidar* è la capacità di “vedere in curva”. Un LiDAR tradizionale rileva oggetti in linea retta, ma durante una curva il veicolo non proseguirà dritto, quindi un ostacolo apparentemente sulla traiettoria potrebbe in realtà trovarsi fuori dal percorso curvo. La soluzione implementata utilizza un'approssimazione del modello cinematico di Ackermann per predire la posizione futura del veicolo:

$$\text{curveOffset} = \frac{z^2 \cdot \tan(\delta)}{2L}$$

dove z è la distanza frontale dell'ostacolo, δ l'angolo di sterzo corrente (in radianti) e L il passo del veicolo (*wheelbase*). L'algoritmo calcola dove si troverà il veicolo quando avrà percorso la distanza z e “raddrizza” virtualmente la posizione dell'ostacolo:

$$\text{effectiveX} = \text{localHit.x} - \text{curveOffset}$$

Se *effectiveX* è vicino a zero, l'ostacolo si trova effettivamente sulla traiettoria curva e deve essere considerato; altrimenti viene ignorato perché il veicolo passerà a lato.

2.4.3 Classificazione dinamica degli ostacoli

Il sensore applica larghezze di rilevamento differenziate in base al tipo di oggetto. Per i pedoni viene utilizzato un campo allargato per massimizzare la sicurezza, mentre per i veicoli del traffico viene usato un campo esteso per anticipare potenziali collisioni agli incroci. Gli ostacoli generici vengono invece valutati sulla larghezza standard della corsia.

2.4.4 Configurazione dual-LiDAR

Il taxi è equipaggiato con due sensori LiDAR: un *Front LiDAR* in posizione frontale, dedicato alla navigazione, al rilevamento degli ostacoli e al supporto alle manovre di sorpasso, e un *Rear LiDAR* in posizione posteriore, utilizzato per la sicurezza durante la retromarcia anti-stallo. Il LiDAR posteriore viene consultato durante queste manovre e, se rileva un ostacolo entro la distanza di sicurezza, la retromarcia viene immediatamente interrotta.

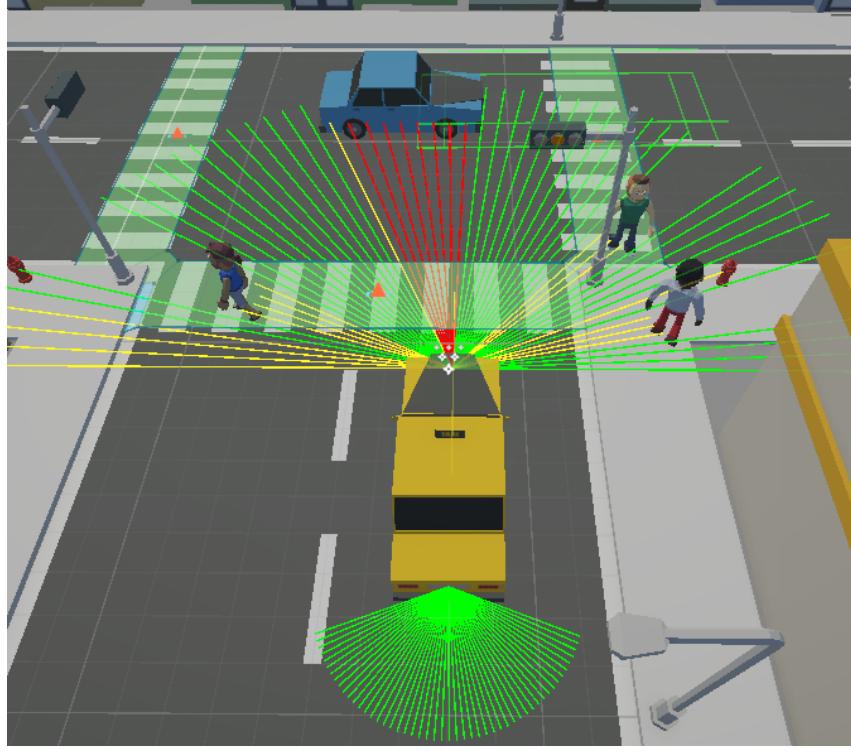


Figure 3: Rilevamento degli ostacoli tramite dual-LiDAR

2.5 Actuator Layer

L’*Actuator Layer* traduce i comandi decisionali del *Brain Layer* in azioni fisiche sul veicolo. È composto da due componenti: il *CarMotor* per la cinematica del movimento e il *CarBattery* per la gestione energetica.

2.5.1 CarMotor

La classe *CarMotor* implementa il modello cinematico del veicolo, traducendo i comandi di velocità e sterzo in movimento fisico il quale segue il modello Ackermann semplificato (*Bicycle Model*), in cui la velocità angolare è calcolata come

$$\omega = \frac{v}{L} \cdot \tan(\delta)$$

dove ω è la velocità angolare di rotazione, v la velocità lineare corrente, L il passo del veicolo e δ l’angolo di sterzo delle ruote. Questo modello cattura il comportamento di un veicolo a quattro ruote con sterzo anteriore, in cui il raggio di curvatura dipende dall’angolo di sterzo e dal passo.

Rotazione realistica

L’agente taxi, invece di ruotare attorno al centro, il *CarMotor* ruota attorno all’asse posteriore:

```
rotationPivot = transform.position - (transform.forward · rearAxeOffset)
```

e la trasformazione viene applicata tramite una rotazione attorno a questo pivot. Questo produce un comportamento visivamente realistico, in cui il muso del veicolo “spazza” lateralmente durante le curve, replicando la cinematica di un’automobile reale.

2.5.2 CarBattery

Il modulo **CarBattery** rappresenta il cuore della gestione energetica del taxi autonomo, simulando in modo realistico il consumo di batteria in base alla distanza percorsa e allo stile di guida selezionato.

Modello di consumo

Il sistema adotta un modello di consumo basato sulla distanza effettivamente percorsa dal veicolo. Ad ogni aggiornamento (con passo di $\Delta t = 1$ secondo) il consumo viene calcolato come:

$$\Delta \text{Consumed} = d_{\text{km}} \cdot 15.0 \cdot m_{\text{policy}}$$

dove d_{km} è la distanza percorsa nell'intervallo, 15.0 è il tasso di consumo base (15% della capacità totale per km) e m_{policy} è il moltiplicatore associato alla policy di guida attiva. Questo approccio permette di ottenere stime del consumo energetico indipendenti dalla velocità istantanea, concentrandosi sulla distanza totale e sullo stile di guida.

Verifica di fattibilità energetica

Per garantire la sicurezza delle missioni, il sistema implementa due livelli distinti di verifica della batteria a seconda del contesto.

1. Controllo Standard (Prenotazione e Monitoraggio)

Utilizzato all'atto della prenotazione iniziale e durante il monitoraggio continuo (es. riccalcoli per meteo, barriere o cambi policy). Si basa su una stima euristica che include un margine di sicurezza e una soglia minima di riserva. La condizione di accettabilità è:

$$B_{\text{attuale}} \geq C_{\text{viaggio}} + \underbrace{C_{\text{viaggio}} \cdot 0.05}_{\text{Margine 5\%}} + \underbrace{15.0}_{\text{Soglia Minima}}$$

dove la soglia minima del 15% serve implicitamente a garantire una riserva per raggiungere una stazione di ricarica a fine corsa.

2. Controllo Cambio Destinazione (Re-routing Utente)

Quando l'utente richiede di cambiare destinazione a corsa iniziata, il sistema esegue un controllo più rigoroso per garantire il *Safe Return*. In questo caso, viene calcolato esplicitamente il percorso di ritorno alla stazione di ricarica più vicina dalla *nuova* destinazione. La formula diventa:

$$B_{\text{attuale}} \geq C_{\text{viaggio}} + C_{\text{ritorno_eco}} + \text{Buffer (10.0)}$$

dove:

- C_{viaggio} : Consumo stimato per raggiungere la nuova destinazione con la policy attuale.
- $C_{\text{ritorno_eco}}$: Consumo stimato per andare dalla nuova destinazione alla stazione di ricarica più vicina, assumendo un profilo di guida **Eco** (moltiplicatore 0.7) per massimizzare l'autonomia di rientro.
- Buffer: Una riserva fissa del 10% della capacità totale.

Impatto delle policy di guida

Le tre policy di guida, diverse per accelerazione e consumi ma limitate alla stessa velocità massima per sicurezza, determinano il consumo energetico attraverso moltiplicatori dedicati salvati nella knowledge base Neo4j

Policy	Moltiplicatore Consumo	Accelerazione	Velocità massima
Eco	0.7	0.7	9.0
Comfort	1.0	0.9	9.0
Sport	1.2	1.2	9.0

Table 1: Parametri base delle policy di guida.

Meccanismi di sicurezza

Il sistema di guida autonoma implementa diversi meccanismi di sicurezza per garantire un comportamento affidabile in scenari complessi:

- **Eventi esterni critici:** in presenza di condizioni esterne non dipendenti dall'utente come *barriera* o *pioggia intensa*, il veicolo esegue un ricalcolo completo del percorso basandosi sulla policy attuale. Se la batteria risulta insufficiente per il nuovo tragitto, viene tentato il passaggio alla modalità *Eco* per verificare se sia possibile raggiungere la destinazione; se anche in questa modalità non vi si riesce, il sistema effettua uno *stop di emergenza*, cioè si ferma al punto più sicuro per lasciare il passeggero.
- **Cambio di policy richiesto dall'utente:** prima di applicare una nuova policy, il sistema esegue un controllo preventivo dell'autonomia residua. Se la batteria non è sufficiente per percorrere il percorso calcolato con la nuova policy, il cambio viene rifiutato e la policy corrente viene mantenuta.
- **Cambio di destinazione:** se la nuova meta non è raggiungibile con la policy attiva, ma lo è in *Eco*, il sistema effettua automaticamente la transizione a questa modalità aggiornando il percorso. Nel caso in cui nemmeno la modalità eco sia sufficiente, viene rifiutato il cambio destinazione.
- **Policy obbligatorie:** per profili che richiedono la policy "Comfort" come le donne in gravidanza, il taxi adotta un comportamento orientato alla safety del passeggero. Nei casi precedenti, nel caso in cui la batteria non sia sufficiente per mantenere la policy comfort fino a destinazione, non viene attivata la modalità *Eco*, ma il veicolo conclude la corsa nel punto sicuro più vicino nel caso di eventi esterni o rifiuta il cambio destinazione.
- **Logica Auto-Eco Fallback:** questa funzione esegue un ricalcolo completo del percorso partendo dalla posizione corrente utilizzando i pesi specifici della modalità *Eco*.

Gestione della ricarica

Il modulo supporta la ricarica automatica presso le stazioni dedicate presenti nella mappa. Quando il taxi si trova in stato di inattività o la batteria scende sotto la soglia minima,

il coordinatore di missione avvia automaticamente la procedura di ricarica, calcolando il tempo necessario per raggiungere il livello ottimale. Questa informazione viene utilizzata per stimare i tempi di attesa delle prenotazioni in coda, garantendo comunicazioni accurate verso gli utenti.

2.6 Planning Layer

Il *Planning Layer* è responsabile del calcolo dei percorsi ottimali che il taxi deve seguire per raggiungere le destinazioni. È composto da due elementi fondamentali: il *Waypoint Graph*, che rappresenta la topologia stradale, e l' A^* *Pathfinder*, che calcola i percorsi.

2.6.1 Waypoint Graph

La rete stradale della città è modellata come un grafo orientato pesato in cui i nodi (*WaypointNode*) rappresentano punti significativi della rete stradale, come incroci, curve, punti di interesse (P.O.I) e stazioni di ricarica. Gli archi corrispondono ai segmenti stradali percorribili, con una direzione che riflette il senso di marcia, mentre i pesi degli archi determinano il “costo” di attraversamento e influenzano la scelta del percorso. Ogni nodo contiene informazioni sulla propria tipologia e sui nodi raggiungibili, permettendo all’algoritmo di pathfinding di esplorare la rete in modo efficiente.

2.6.2 Calcolo del percorso

Quando il taxi deve raggiungere una destinazione, il processo di calcolo del percorso segue una sequenza di passi. Per prima cosa avviene l’identificazione dei nodi rilevanti, ovvero il nodo di partenza (posizione corrente o waypoint attuale) e il nodo di destinazione (P.O.I target). Successivamente vengono acquisiti i pesi della zona: i moltiplicatori di costo per le diverse zone urbane sono estratti dalla *Knowledge Base* tramite query al *KBService* e riflettono le policy di guida attive, ad esempio evitare zone trafficate o preferire percorsi più sicuri (per esempio in base al manto stradale). A questo punto l’algoritmo A^* calcola il percorso ottimale considerando sia la distanza euclidea sia i pesi delle zone; la sequenza di waypoints risultante viene quindi passata al *Navigation Module* per l’esecuzione.

2.6.3 Pathfinding dinamico pesato

Una caratteristica distintiva del sistema è il pathfinding *context-aware*: i pesi degli archi non sono statici, ma vengono modulati dinamicamente in base al contesto. Le policy di guida consentono all’utente di richiedere percorsi più veloci, più sicuri o più economici; le condizioni meteo, ad esempio in caso di pioggia, possono rendere alcune zone più costose; lo stato del traffico assegna pesi maggiori alle aree congestionate. I moltiplicatori zonali vengono recuperati dalla *Knowledge Base* tramite query che tengono conto della policy attiva e delle condizioni ambientali correnti, permettendo al taxi di adattare i propri percorsi in tempo reale.

2.7 Mission Layer

Il Mission Layer rappresenta il livello più alto dell’architettura, responsabile della gestione strategica delle missioni del taxi. Il componente principale è il *TaxiMissionController*, che implementa una Macchina a Stati Finiti (FSM) per orchestrare l’intero ciclo di vita di una corsa.

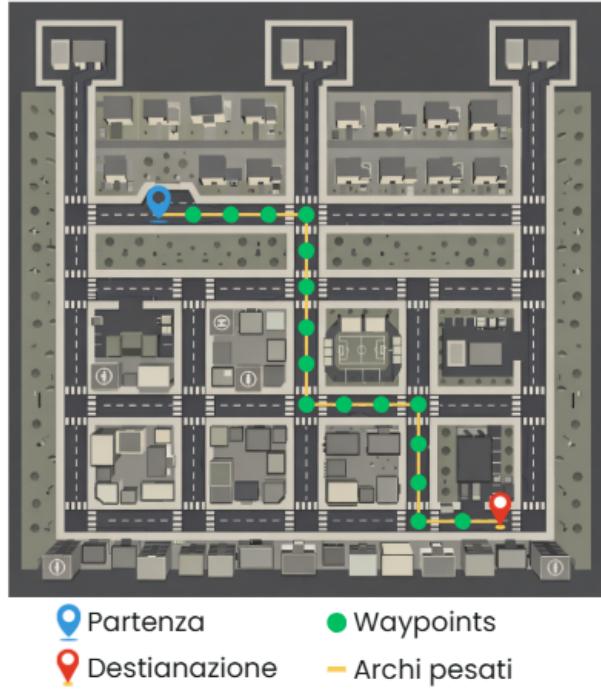


Figure 4: Esempio calcolo del percorso

2.7.1 Macchina a Stati Finiti

La FSM del Mission Layer coordina le operazioni del taxi attraverso una serie di stati interconnessi, come illustrato nella figura seguente:

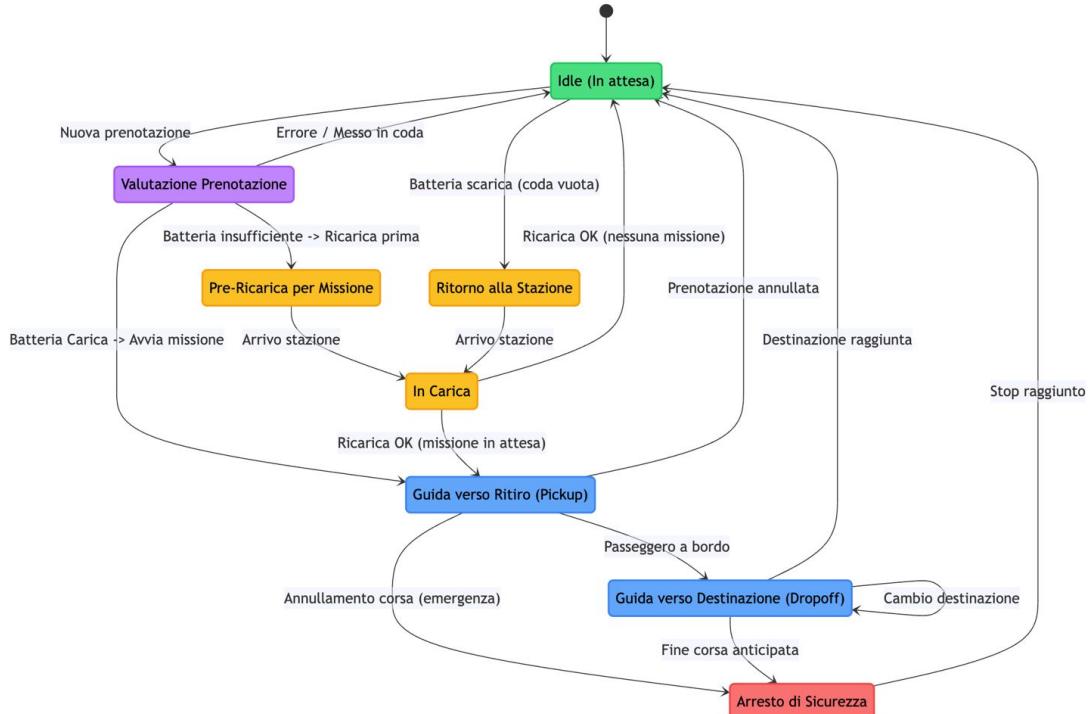


Figure 5: Schema della gestione delle missioni, con rappresentazione degli stati della FSM e dei controlli esterni.

Il *Taxi Mission Controller* gestisce quattro aspetti fondamentali del servizio taxi:

- **Coda di prenotazioni:** gestione FIFO delle richieste di prenotazione in arrivo.
- **Gestione della batteria:** stima dell'autonomia e decisione di ricarica preventiva.
- **Pickup & Dropoff:** caricamento/scaricamento del cliente e cambio destinazione.
- **Arresto:** annullamento corsa e fine corsa anticipata.

2.7.2 Gestione delle prenotazioni

Il sistema implementa una coda FIFO per le prenotazioni: quando arriva una nuova richiesta mentre il taxi è occupato, questa viene accodata e, al termine della corsa corrente, il taxi verifica la coda e avvia automaticamente la missione successiva. Se durante la valutazione si verifica un errore o la richiesta non è fattibile, la prenotazione viene messa in coda o rifiutata e il taxi ritorna allo stato *Idle*.

2.7.3 Integrazione con la Knowledge Base

Il *Mission Layer* interagisce con il *KBService* per eseguire diverse operazioni: le *policy queries*, che recuperano i moltiplicatori delle zone in base alla policy di guida attiva; il cambio di policy, che comporta il ricalcolo del percorso quando l'utente modifica la policy durante la corsa; e la gestione del meteo, che adatta i pesi in caso di condizioni atmosferiche avverse. Queste query permettono al taxi di prendere decisioni informate sulla scelta del percorso, integrando la conoscenza del dominio con le preferenze dell'utente.

3 Ambiente Dinamico della Simulazione

L'ambiente della simulazione è progettato per essere non deterministico e reattivo: include elementi dinamici che interagiscono con l'agente taxi autonomo, creando scenari realistici che sfidano il sistema di guida. Questa sezione analizza l'implementazione dei componenti ambientali.

3.1 Traffico Veicolare

Il traffico veicolare è simulato attraverso veicoli *TrafficCarIA*, ovvero un agente reattivo a stati finiti che simula il comportamento di un veicolo nel traffico urbano. Il veicolo naviga lungo un percorso predefinito (waypoint circuit) e reagisce dinamicamente a vincoli fisici (ostacoli e pedoni) e normativi (semafori).

3.1.1 FSM

Il comportamento di ogni veicolo del traffico è governato da una FSM a quattro stati, descritti di seguito e illustrati in figura 6.

- **Driving:** stato principale in cui il veicolo segue il percorso, rileva ostacoli e rispetta i semafori.
- **Blocked:** il veicolo è fermo a causa di un ostacolo; attende la rimozione e avvia il timer anti-stallo, alla fine del quale va in stato di *Reversing*.
- **Reversing:** stato in cui viene eseguita una manovra di retromarcia per tentare di sbloccarsi.
- **Waiting:** fase di attesa successiva alla retromarcia, prima di riprendere la guida normale.

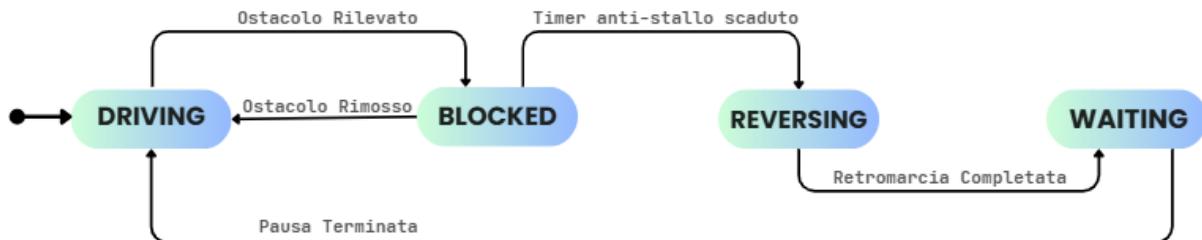


Figure 6: FSM per TrafficCarAI

3.1.2 Navigazione su circuito

I veicoli del traffico seguono percorsi definiti tramite *WaypointCircuit*, una struttura che contiene una sequenza ordinata di waypoint. La **navigazione** avviene attraverso tre passi principali: in fase di inizializzazione il veicolo identifica il waypoint più vicino sulla propria tratta, durante la **progressione** passa al waypoint successivo quando la distanza da quello corrente scende sotto la soglia *reachThreshold* e, al termine del circuito, ricomincia dal primo waypoint realizzando un **loop** continuo.

3.1.3 Sensore volumetrico

Il sistema di rilevamento ostacoli utilizza un sensore volumetrico con distanza dinamica proporzionale alla velocità, definita come

$$\text{distanzaRilevamento} = \text{detectionDistance} + (\text{velocità} \times \text{speedDetectionFactor})$$

in modo che un veicolo più veloce inizi a frenare molto prima rispetto a uno lento, evitando collisioni brusche. Il sensore impiega una `Physics.BoxCast` orientata verso la direzione del prossimo waypoint (non necessariamente dritta), così da “vedere” anche nelle curve.

3.1.4 Frenata progressiva

Quando viene rilevato un ostacolo, la velocità viene ridotta secondo una curva quadratica:

$$\text{velocità} = \text{maxSpeed} \times (\text{fattoreDistanza})^2$$

$$\text{fattoreDistanza} = \frac{\text{distanzaOstacolo} - \text{stopDistance}}{\text{distanzaRilevamento} - \text{stopDistance}}$$

che produce un rallentamento graduale a distanza e una frenata più aggressiva in prossimità dell’ostacolo. Se la distanza scende sotto la soglia `stopDistance`, il veicolo si arresta completamente.

3.1.5 Interazione con i semafori (V2I)

I veicoli del traffico comunicano con i semafori intelligenti tramite un pattern V2I. All’ingresso nel trigger del semaforo il veicolo si sottoscrive all’evento `OnStateChanged`, ricevendo e tracciando internamente gli stati *Red*, *Yellow* e *Green*; la velocità viene quindi modulata in base alla distanza dalla linea di stop: rallentamento progressivo quando è lontano, arresto completo in prossimità della linea e via libera oltre la linea per liberare rapidamente l’incrocio. All’uscita dal trigger la sottoscrizione viene rimossa, mentre il controllo `IsStopLineBehind()` impedisce che il veicolo rimanga bloccato in mezzo all’incrocio quando il semaforo diventa rosso dopo che ha già superato la linea di stop.

3.2 Pedoni

I pedoni sono simulati attraverso *PedestrianAI*, ovvero un agente autonomo NavMesh-based dotato di personalità comportamentale. Il pedone naviga sui marciapiedi e decide se attraversare la strada basandosi sul proprio profilo di rischio e sulla percezione del traffico.

3.2.1 FSM

Il comportamento dei pedoni è governato da una FSM, i cui stati sono descritti di seguito e illustrati in figura 7.

- **Wandering:** esplorazione casuale del marciapiede con pause periodiche.
- **WaitingToCross:** attesa al bordo strada, durante la quale il pedone valuta se attraversare in base al rischio percepito.

- **Crossing:** attraversamento attivo della strada.
- **Idle:** fermo in attesa (ad esempio dopo la salita o la discesa dal taxi).
- **HailingTaxi:** il pedone sta chiamando il taxi con un'animazione di saluto.
- **MovingToPickup:** il pedone si sta spostando verso un punto di pickup valido.

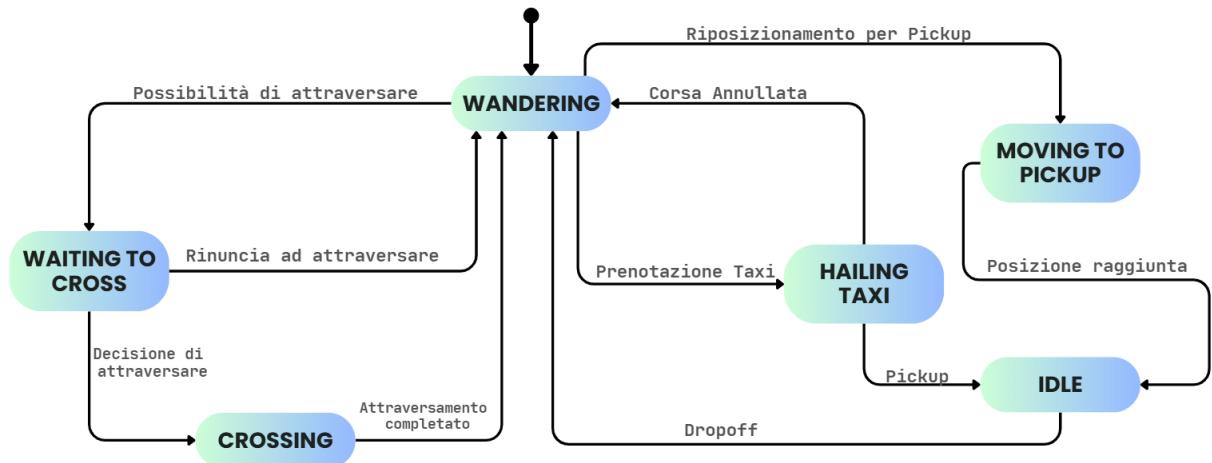


Figure 7: FSM per PedestrianAI

3.2.2 Sistema di personalità

Ogni pedone ha una personalità che influenza radicalmente il suo comportamento. Sono attualmente modellate due personalità principali:

- **Calmo:** velocità pari a 1.8 m/s (cammina); valuta il rischio e tende ad aspettare se stanno arrivando auto.
- **Impulsivo:** velocità pari a 5.0 m/s (corre); attraversa immediatamente senza controllare il traffico.

La personalità influisce anche sulla *riskTolerance*, ovvero la distanza entro cui il pedone percepisce le auto come pericolose. Un pedone calmo con alta tolleranza attraverserà solo se l'auto è abbastanza lontana (minimo 5m), mentre valori più bassi di tolleranza portano a comportamenti meno prudenti.

3.2.3 Navigazione NavMesh

I pedoni utilizzano il sistema di navigazione *NavMesh* di Unity, una mesh pre-calcolata che definisce le superfici percorribili. La *NavMesh* della simulazione include due tipologie di aree:

- **Sidewalk:** marciapiede, zona sicura con costo base basso.
- **Dangerous:** carreggiata stradale con costo diverso in base al tipo di attraversamento (su strisce pedonali oppure no).

La NavMesh non viene generata dinamicamente, ma è pre-calcolata (*baked*) in fase di sviluppo e include già sia i marciapiedi sia le strade. A runtime varia invece la *maschera di navigazione* (*areaMask*) del singolo pedone, che funge da filtro sulle aree accessibili:

- **Wandering**: `areaMask = sidewalkMask` → solo marciapiede.
- **Crossing**: `areaMask = allAreasMask` → marciapiede + strada.
- **Post-crossing**: `areaMask = sidewalkMask` → ritorno al solo marciapiede.

Il ripristino della maschera dopo l'attraversamento avviene con un piccolo *delay* (0.1s) per evitare che il *NavMeshAgent* “teletrasporti” il pedone sul marciapiede più vicino prima che abbia completato il movimento.

3.2.4 Logica di attraversamento

Quando un pedone raggiunge un punto di attraversamento viene invocata `RequestCrossing`, che provoca la transizione allo stato *WaitingToCross* e la memorizzazione della destinazione. La decisione effettiva di attraversare è gestita da `DecisionLogic`, eseguita periodicamente ogni *decisionInterval*: un pedone *Impulsive* attraversa immediatamente, mentre un pedone *Calm* controlla il risultato di `IsCarApproaching(riskTolerance)` e, se la situazione è sicura, ha una probabilità *crossingProbability* di iniziare l'attraversamento, altrimenti continua ad aspettare. Quando la decisione è positiva, `StartCrossing` sblocca la NavMesh, imposta la nuova destinazione e provoca la transizione allo stato *Crossing*; il completamento avviene quando *remainingDistance* è minore di 0.1 m, momento in cui il pedone ritorna allo stato *Wandering*. Un timer anti-stallo di 5 secondi nello stato *WaitingToCross* impedisce che un pedone rimanga bloccato indefinitamente.

3.2.5 Interazione con il taxi

Alcuni pedoni sono associati a utenti, identificati tramite un *userId* configurato, e possono diventare passeggeri. In questo caso il pedone entra nello stato *HailTaxi*, si ferma, si gira verso il veicolo e attiva l'animazione di saluto (*wave*); successivamente passa alla fase *ApproachVehicle*, durante la quale si avvicina alla portiera destra del taxi, lato marciapiede. Raggiunta la posizione di imbarco, entra in stato *Idle*, pronto per la salita, che viene gestita esternamente dal *MissionCoordinator*; dopo la discesa dal veicolo il pedone transita nuovamente a *ReturnToWandering* e riprende l'esplorazione.

3.3 Semafori Intelligenti

I semafori della simulazione sono implementati come **Smart Traffic Light**, dispositivi che simulano una comunicazione **V2I (Vehicle-to-Infrastructure)** ispirata alle tecnologie 5G. A differenza dei semafori tradizionali che richiedono riconoscimento visivo, questi semafori comunicano attivamente il proprio stato ai veicoli nel raggio d'azione.

3.3.1 Ciclo semaforico

Ogni semaforo esegue un ciclo temporizzato gestito tramite coroutines di Unity, alternando quattro stati principali:

- **Red** (durata predefinita 5 s): i veicoli devono fermarsi alla linea di stop.

- **Green** (10 s): via libera al transito attraverso l’incrocio.
- **Yellow** (2 s): fase di transizione in cui i veicoli rallentano e si preparano allo stop.
- **Off**: semaforo spento, in cui il controllo del traffico non è più regolato dal segnale luminoso.

Le durate dei singoli stati sono configurabili per ogni semaforo tramite i parametri *redDuration*, *greenDuration* e *yellowDuration*. Il parametro *startDelay* consente di sfasare l’avvio dei cicli tra semafori diversi, permettendo di creare “onde verdi” lungo un asse stradale, e coordinare il comportamento corretto in caso di incroci ad X.

3.3.2 Comunicazione V2I

La comunicazione tra semaforo e veicoli avviene tramite un *trigger collider* che definisce la zona di copertura del segnale. Quando un veicolo entra nel trigger, il semaforo richiama `ConnectToTrafficLight()` sul veicolo, che a sua volta si sottoscrive all’evento `OnStateChanged` del semaforo. Ad ogni cambio di stato, il semaforo invia notifiche in tempo reale a tutti i veicoli connessi, mentre al termine dell’attraversamento `DisconnectFromTrafficLight()` rimuove la sottoscrizione. Questo pattern di tipo *Observer* garantisce che i veicoli ricevano immediatamente le informazioni sui cambi di stato, evitando la necessità di interrogare continuamente il semaforo in polling.

3.3.3 Feedback visivo

Il semaforo aggiorna i propri materiali per simulare l’accensione delle luci, fornendo un feedback visivo coerente con lo stato logico. Ogni stato dispone di un materiale illuminato dedicato (*lightOnRedMat*, *lightOnYellowMat*, *lightOnGreenMat*), mentre le luci inattive utilizzano un materiale spento comune (*lightOffMat*). Lo switch tra materiali avviene nel metodo `UpdateVisuals()` ad ogni cambio di stato, sincronizzando l’aspetto grafico con la logica del ciclo semaforico.

3.3.4 Gestione guasti

Il sistema supporta la simulazione di guasti dinamici dei semafori. Il metodo `SetWorkingStatus(false)` spegne il semaforo e interrompe il ciclo normale, portandolo nello stato *Off* in cui tutte le luci sono spente e i veicoli trattano l’incrocio come una situazione di “dare precedenza”. Chiamando `SetWorkingStatus(true)` il ciclo viene riavviato e il semaforo torna al comportamento ordinario. Questa funzionalità consente di modellare scenari di emergenza o manutenzione all’interno dell’ambiente simulato.

3.4 Barriere Dinamiche

Le barriere dinamiche (*RoadBlock*) sono ostacoli temporanei che appaiono durante la simulazione per bloccare determinate strade. Rappresentano eventi imprevisti come lavori stradali, incidenti o emergenze, costringendo il taxi a ricalcolare il proprio percorso.

3.4.1 Ciclo di vita

Le barriere seguono un ciclo di vita semplice. In fase di **attivazione** la barriera viene resa attiva (`SetActive(true)`) da un sistema esterno, ad esempio uno *spawn manager*. Successivamente avviene il **rilevamento**: il taxi individua la barriera tramite la combinazione tra scansione LiDAR e controllo del tag *RoadBlock*. Una volta rilevata, viene chiamato `TriggerDisappearance()`, che avvia il **timer** di sparizione; trascorsi *vanishDelay* secondi la barriera si disattiva automaticamente. In caso di riattivazione, lo stato interno viene **resetto** nel metodo `OnEnable()`. Questo timer di sparizione simula uno scenario realistico in cui, dopo un certo tempo, l'ostacolo viene rimosso perché i lavori sono terminati o l'incidente è stato risolto.

3.4.2 Rilevamento dal taxi

Il modulo di percezione del taxi (*CarBrain_Perception*) identifica le barriere attraverso una catena di passi: innanzitutto la **scansione LiDAR**, durante la quale uno dei raggi colpisce l'oggetto; quindi il **controllo del tag**, che verifica se l'oggetto colpito è marcato con il tag *RoadBlock*. In caso positivo viene emessa una **notifica** tramite l'evento `OnRoadBlockDetected`, che viene ricevuto dal *TaxiMissionController* e innesca il **ricalcolo del percorso**, attivando il pathfinding alternativo.

3.4.3 Integrazione con il pathfinding

Quando una barriera viene rilevata, il sistema di pianificazione del percorso integra l'informazione nel grafo stradale. Per prima cosa viene identificato il waypoint o il segmento di rete bloccato dalla barriera, che viene temporaneamente marcato come non attraversabile. Successivamente il percorso A* viene ricalcolato escludendo il nodo interessato, così da trovare una rottura alternativa. Se non esiste alcun percorso valido che aggiri l'ostacolo, il taxi si pone in attesa o segnala un errore.

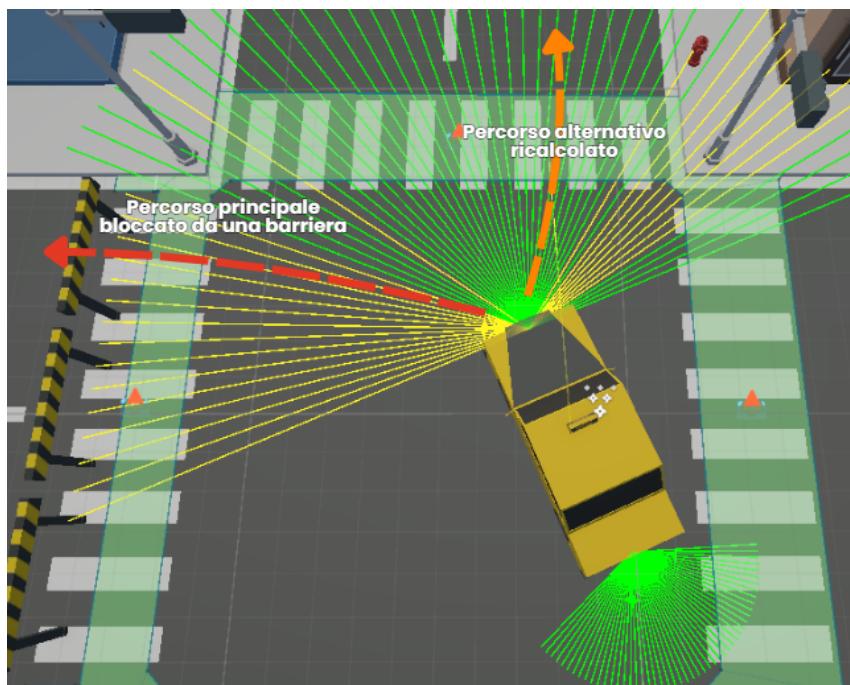


Figure 8: Ricalcolo del percorso

3.5 Sistema Meteorologico

La simulazione integra un sistema meteorologico dinamico, gestito dal componente `WeatherManager`, che introduce variabilità ambientale durante l'esecuzione della missione.

3.5.1 Variazione delle condizioni meteo

Ogni 30 secondi il sistema esegue un controllo probabilistico (10%) per attivare un evento di pioggia. Quando la pioggia inizia, persiste per un intervallo casuale compreso tra 30 secondi e 2 minuti, quindi termina e avvia un periodo di *cooldown* di 60 secondi durante il quale il meteo rimane sereno e non possono verificarsi nuovi eventi. Oltre alla generazione automatica, l'operatore può attivare o disattivare manualmente la pioggia dall'interfaccia Unity, bypassando sia la probabilità sia il cooldown, funzionalità utile per scopi dimostrativi e di testing. L'effetto visivo è realizzato mediante un sistema particolare ancorato al veicolo, che simula la pioggia solo nell'area circostante il taxi, riducendo il carico di rendering rispetto a una simulazione estesa all'intera mappa.

3.5.2 Impatto sulla navigazione

Il sistema meteorologico infuisce direttamente sulla pianificazione del percorso. In caso di pioggia, l'algoritmo di pathfinding penalizza le zone con superfici stradali più rischiose (ad es. sanpietrini o manto dissestato), preferendo strade considerate più sicure in condizioni di bagnato. Il meteo viene considerato solo quando il passeggero è a bordo: durante l'avvicinamento al punto di *pickup* il taxi ignora le penalità meteo per privilegiare la rapidità. Ogni ricalcolo dovuto al meteo viene comunicato all'utente tramite il sistema di explainability, spiegando la motivazione del cambio di percorso.

3.6 Sistema temporale

La simulazione implementa un sistema di gestione del tempo attraverso il componente `TimeManager`, che consente di comprimere lo scorrere di un'intera giornata in un intervallo di tempo ridotto.

3.6.1 Mappatura tempo reale–tempo simulato

Il sistema utilizza un fattore di scala temporale per accelerare il tempo simulato. La conversione è definita da:

$$\text{Fattore di scala} = \frac{24 \times 60 \times 60}{\text{Minuti reali per giornata} \times 60}$$

Con la configurazione di default, una giornata completa di 24 ore simulate corrisponde a 120 minuti di tempo reale. Ne deriva un fattore di scala pari a 12×: ogni secondo reale equivale a 12 secondi simulati. La simulazione parte quindi dalle ore 08:00 del tempo simulato e prosegue ciclicamente, tornando alle 8:00 al termine delle 24 ore.

3.6.2 Controllo del tempo

Dall'interfaccia Unity sono disponibili diverse funzionalità di controllo:

- **Pausa:** permette di sospendere lo scorrimento del tempo simulato;

- **Impostazione orario:** consente di impostare manualmente un orario specifico per testare scenari mirati;
- **Velocità variabile:** la durata del giorno può essere modificata per accelerare o rallentare la simulazione.

3.6.3 Integrazione con la knowledge base

L'ora simulata viene inviata al backend nelle richieste di calcolo del percorso. La knowledge base Neo4j contiene nodi di tipo **Context** che rappresentano fasce orarie con caratteristiche particolari (es. ingresso/uscita scolastica, eventi, turni lavorativi). Quando l'ora corrente rientra in una di queste fasce, il contesto corrispondente viene considerato attivo e influenza i moltiplicatori di costo delle zone interessate.

3.7 Suddivisione in zone

La mappa della simulazione è suddivisa in zone geografiche distinte, ciascuna con caratteristiche specifiche che influenzano il comportamento del pathfinding e le decisioni di navigazione del taxi.

3.7.1 Implementazione in Unity

Le zone sono implementate tramite il componente **CityZone**, uno script associato a **GameObject** dotati di **BoxCollider** configurato come *trigger*. All'avvio della simulazione il sistema associa automaticamente a ciascuna zona tutti i waypoint che ricadono al suo interno, creando una mappatura tra nodi del grafo e aree geografiche. Ogni zona è identificata da un **kbZoneId**, che consente la corrispondenza con i nodi **Zone** memorizzati nella knowledge base Neo4j. Questo collegamento è fondamentale per applicare i moltiplicatori di costo calcolati dal backend durante la pianificazione del percorso.

3.7.2 Zone definite

La mappa è suddivisa in quattro zone, come mostrato nella figura 9

- **Centro Storico** – superficie: sanpietrino (*cobblestone*); area turistica e commerciale con pavimentazione irregolare tipica dei centri storici italiani, spesso soggetta a regolamentazione ZTL.
- **Zona Residenziale** – superficie: dossi rallentatori; quartiere abitativo con presenza di dossi e limiti di velocità ridotti per la sicurezza dei residenti.
- **Zona Suburbana** – superficie: asfalto liscio; area periferica con strade moderne ben asfaltate, che garantiscono una percorrenza fluida e confortevole.
- **Zona Industriale** – superficie: strada dissestata; area produttiva con manto stradale deteriorato dal passaggio frequente di mezzi pesanti.

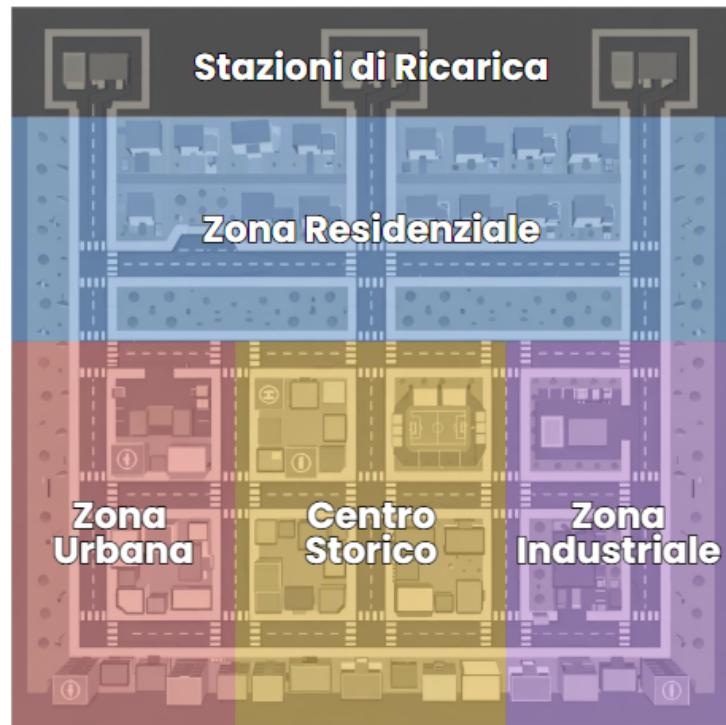


Figure 9: Suddivisione della mappa in zone

4 Ontologia e Inferenza

Questa sezione presenta la modellazione semantica del sistema. L'approccio adottato parte dalla definizione formale dell'ontologia utilizzata, per poi analizzare la sua concreta implementazione all'interno di Neo4j.

4.1 Modellazione Ontologica

L'ontologia proposta formalizza il dominio del sistema dell'agente taxi autonomo che opera in un ambiente dinamico attraverso una struttura gerarchica rigorosa, progettata per supportare il ragionamento semantico e la navigazione *context-aware*.

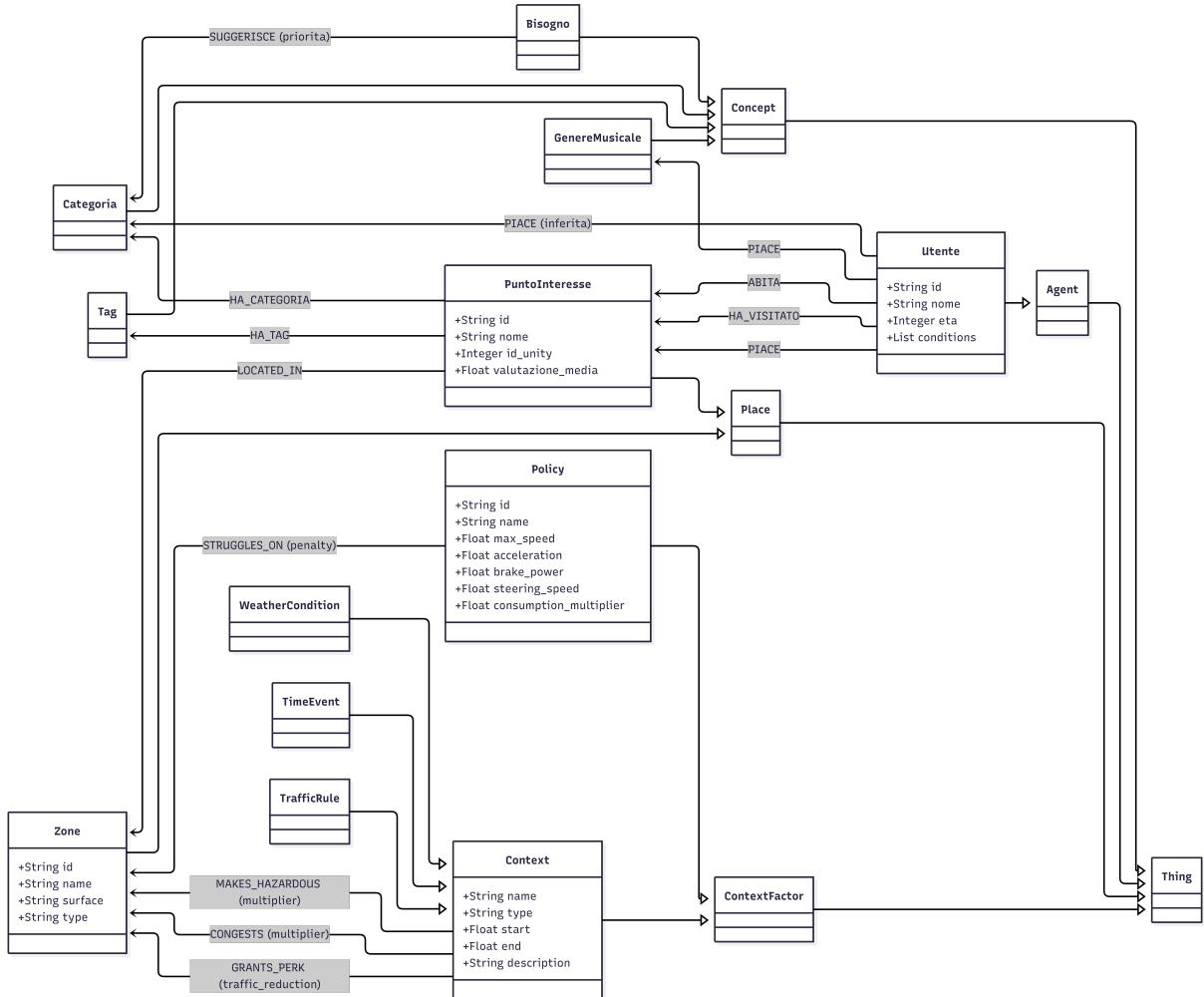


Figure 10: Schema dell'Ontologia

4.1.1 Gerarchia delle classi

La tassonomia dell'ontologia si radica nella classe astratta **Thing**, introdotta come livello più generale per garantire uniformità alla modellazione OWL e favorire l'estendibilità del modello. A partire da essa, la gerarchia si articola in cinque concetti fondamentali: **Agent**, **Place**, **Concept**, **Context** e **ContextEffect**, ciascuno responsabile della rappresentazione di un aspetto semantico distinto del dominio.

Agent

La classe **Agent** modella le entità attive dotate di capacità decisionale all'interno del sistema.

Aspetto	Neo4j	Ontologia
Classe generale	Nodo :Agent	Classe Agent
Utente umano	Nodo :Utente	Sottoclasse User di Agent
Identificativo	Proprietà del nodo id	Proprietà dati hasId
Età	Proprietà del nodo age	Proprietà dati hasAge
Condizione fisica	Proprietà del nodo physicalCondition	Proprietà dati hasPhysicalCondition

Place

La macro-classe **Place** modella le entità spaziali rilevanti per la navigazione e distingue semanticamente tra destinazioni puntuali dell'utente (**PointOfInterest**) e aree geografiche su cui si propagano effetti contestuali e vincoli di navigazione (**Zone**).

Sottoclasse	Neo4j Label	Proprietà chiave
PointOfInterest	:PuntoInteresse	hasUnityId, hasRating
Zone	:Zone	hasZoneId, hasSurfaceType

Concept

La classe astratta **Concept** raggruppa le astrazioni logiche utilizzate per il ragionamento ad alto livello e costituisce il vocabolario semantico del sistema di raccomandazione. I suoi sottotipi possono essere distinti in concetti descrittivi, impiegati per rappresentare preferenze e caratteristiche (**Tag**, **MusicGenre**, **Category**), e concetti decisionali, direttamente coinvolti nei processi di inferenza e selezione (**Need**, **Policy**).

Sottoclasse	Neo4j Label	Descrizione
Need	:Bisogno	Bisogni utente (fame, sete, ecc.).
Category	:Categoria	Categorie dei Point of Interest.
Tag	:Tag	Etichette descrittive.
MusicGenre	:GenereMusicale	Generi musicali.
Policy	:Policy	Profili di guida (Comfort, Sport, Eco).

Context

La classe **Context** modella le condizioni dinamiche che influenzano il comportamento dell'agente e il processo di navigazione. Nell'ontologia formale, la distinzione tra differenti tipologie di contesto è espressa tramite sottoclassi dedicate, mentre nell'implementazione Neo4j la stessa informazione è rappresentata mediante una proprietà **type** sui nodi :Context, al fine di semplificare la modellazione e le interrogazioni. Le proprietà dati **hasStartTime** e **hasEndTime** definiscono l'intervallo temporale di validità del contesto.

Sottoclasse	Neo4j type	Descrizione
WeatherCondition	"Weather"	Condizioni meteorologiche.
TimeEvent	"Time"	Fasce orarie ed eventi temporali.
TrafficRule	"ZTL_Rule"	Regole di traffico contestuali.

ContextEffect

Una differenza chiave tra l'ontologia formale e l'implementazione Neo4j riguarda la gestione dei *pesi* (ad esempio *penalty*, *multiplier*, *reduction*) che influenzano i costi di attraversamento delle zone. Neo4j, in quanto *Labeled Property Graph*, consente di associare tali valori direttamente alle relazioni; ad esempio:

```
(Rain)-[:MAKES_HAZARDOUS {multiplier: 3.0}]->(Cobblestone)
```

Nell'ontologia, invece, i pesi vengono reificati tramite la classe `ContextEffect`, che modella in modo esplicito gli effetti numerici che contesti e policy esercitano sulle zone. La Tabella ?? illustra la corrispondenza tra la rappresentazione implementativa nel grafo Neo4j e il modello ontologico formale basato su `ContextEffect`.

Aspetto	Neo4j	Ontologia
Nodo sorgente	Nodo :Context (es. Rain)	Istanza di WeatherCondition
Relazione	Arco diretto :MAKES_HAZARDOUS	Proprietà oggetto hasContext
Peso dell'effetto	Proprietà dell'arco <code>multiplier = 3.0</code>	Proprietà dati <code>hasEffectValue = 3.0</code>
Tipo di effetto	Implicito nel tipo di relazione	Proprietà dati <code>hasEffectType = "Hazard"</code>
Zona bersaglio	Nodo :Zone	Istanza di Zone, collegata tramite <code>affectsZone</code>

Dal punto di vista semantico, il cammino ontologico



è equivalente alla relazione diretta pesata del grafo Neo4j tra contesto e zona. Questa dualità è intenzionale: l'ontologia fornisce una descrizione formale e rigorosa del dominio, mentre Neo4j implementa una rappresentazione ottimizzata per l'elaborazione dei costi e per gli algoritmi di pathfinding.

4.1.2 Analisi delle relazioni

Le relazioni principali tra le classi sono modellate come *object properties* e hanno un corrispettivo diretto nelle relazioni del grafo Neo4j, come riassunto nelle tabelle seguenti.

Relazioni User–POI

Le relazioni tra utente e POI collegano lo storico di interazione e le preferenze esplicite dell'utente ai luoghi fisici.

Ontologia	Neo4j	Significato
livesIn	:ABITA	Associa ogni User al POI di residenza univoco.
hasVisited	:HA_VISITATO	Rappresenta lo storico delle visite dell'utente ai POI.
likesPlace	:PIACE	Codifica preferenze esplicite o inferite dell'utente verso i POI.

Relazioni semantiche

Le relazioni semantiche collegano bisogni astratti a categorie di POI, abilitando una raccomandazione dichiarativa orientata all'obiettivo.

Ontologia	Neo4j	Significato
hasSatisfactionSuggestion	:SUGGERISCE	Collega un Need alle Category idonee a soddisfarlo, permettendo di selezionare i POI coerenti con i bisogni correnti.

Relazioni topologiche

Le relazioni topologiche descrivono l'ancoraggio spaziale dei POI alle zone, supportando sia il reasoning contestuale sia la propagazione degli effetti.

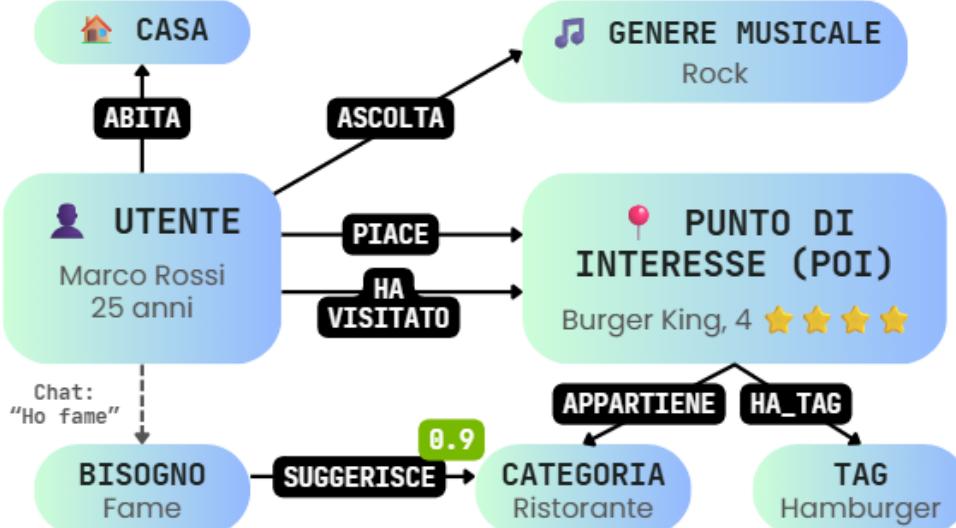
Ontologia	Neo4j	Significato
locatedIn	:LOCATED_IN	Collega ogni PointOfInterest alla Zone in cui è situato.
contains	Relazione inversa di :LOCATED_IN	Permette di recuperare tutti i POI contenuti in una zona e di propagarle gli effetti contestuali.

4.2 Implementazione della Base di Conoscenza

L'ontologia definita è stata mappata in un Property Graph su Neo4j per gestire le query di raccomandazione e il calcolo dinamico dei pesi per la navigazione.

4.2.1 Schema Core: Ricerca e Personalizzazione

Lo schema di ricerca e personalizzazione è composto dai seguenti nodi e proprietà, come mostrato nella figura 11



- + La relazione **“Piace”** viene inserita dopo **3 visite**
- ⚠ La relazione **“Suggerisce”** ha un attributo **“Priorità”** che esprime il grado di **compatibilità**

Figure 11: Schema core: Ricerca e Personalizzazione

- **Utente:** agente umano che interagisce con il sistema, identificato da *Nome* ed *Età*.

Relazione	Destinazione	Significato
ABITA	Casa	Collega l'utente alla sua abitazione principale.
ASCOLTA	Genere musicale	Preferenza musicale esplicita.
PIACE	POI	Preferenza esplicita verso un luogo.
HA_VISITATO	POI	Storico delle visite effettuate.

- **Punto di Interesse:** destinazione fisica nel mondo virtuale, raggiungibile dal taxi, identificata da *Nome* e *Rating*.

Relazione	Destinazione	Significato
APPARTIENE	Categoria	Classificazione tassonomica.
HA_TAG	Tag	Parole chiave descrittive.

- **Bisogno:** stato interno o necessità espressa dall'utente tramite chat.

Relazione	Destinazione	Significato
SUGGERISCE	Categoria	Relazione dotata dell'attributo <i>Priorità</i> , che indica quanto una categoria è idonea a soddisfare il bisogno (grado di compatibilità).

- **Categoria:** raggruppamento logico dei POI che organizza i luoghi in insiemi coerenti. Funge da ponte semantico tra il linguaggio naturale dell'utente (**Bisogno**) e i luoghi fisici (**POI**).

- **Tag:** descrittore di dettaglio intrinseco al POI, utilizzato per arricchirne la rappresentazione semantica. Abilita ricerche più granulari e filtri “orizzontali” che attraversano diverse categorie.

4.2.2 Schema Environment: Policy e Zone

Lo schema di navigazione e di contesto è composto dai seguenti nodi e interazioni, raffigurati in Figura 12. La formula per il calcolo del peso dell’arco mostrata in figura è approfondita nella Sezione 5.5.



Figure 12: Schema Environment: Policy e Zone

- **Zona:** nodo centrale del livello ambiente che rappresenta un’area geografica. È identificata da un *Nome*, che funge da identificativo dell’area, e da una *Superficie*, che descrive la caratteristica fisica della strada.
- **Policy:** stile di guida attivo che interagisce con le caratteristiche delle zone.

Relazione	Destinazione	Significato
ODIA	Zona	Definisce una penalità statica se una policy non è adatta alla superficie di una zona.

- **Meteo:** condizione ambientale corrente che influenza la percorribilità delle zone.

Relazione	Destinazione	Significato
RENDE PERICOLOSA	Zona	La pioggia rende alcune superfici molto più costose da attraversare in sicurezza.

- **Traffico:** eventi temporali legati a luoghi specifici che impattano il flusso veicolare.

Relazione	Destinazione	Significato
GENERA TRAFFICO	Zona	Aumenta il costo di attraversamento di una zona in intervalli orari specifici.

- **ZTL:** regole di restrizione del traffico attive in determinate fasce orarie.

Relazione	Destinazione	Significato
RIDUCE TRAFFICO	Zona	Quando la ZTL è attiva il traffico privato diminuisce, rendendo la zona preferibile per il taxi.

5 L'algoritmo A*

L'algoritmo A* pesato è una variante dell'A* standard che modifica i costi degli archi in base a fattori contestuali. Nel sistema dell'agente taxi autonomo viene utilizzato per calcolare percorsi ottimali tenendo conto di:

- Policy di guida selezionata;
- Condizioni meteorologiche;
- Orario ed eventi temporali;
- Zone della città e tipo di superficie stradale.

In particolare, il costo di attraversamento di una zona problematica viene moltiplicato per un fattore di penalizzazione, così da scoraggiare l'ingresso in aree non adatte alla policy corrente e rendere l'algoritmo *consapevole del contesto*.

5.1 Architettura del Grafo di Navigazione

Ogni nodo del grafo di navigazione è rappresentato nella simulazione Unity da un oggetto `Waypoint`, istanza della classe `WaypointNode`, che presenta le seguenti proprietà:

Proprietà	Descrizione
<code>nodeType</code>	Road, POI, ChargingStation.
<code>isBlocked</code>	Nodo temporaneamente bloccato (barriere dinamiche).
<code>isStoppingAllowed</code>	Permette la fermata per operazioni di pickup.
<code>movementCost</code>	Costo base dell'arco (default: 1.0).
<code>outgoingConnections</code>	Archi uscenti verso i nodi connessi.
<code>parentZone</code>	Zona di appartenenza del nodo.

Table 2: Proprietà principali della classe WaypointNode.

5.2 Implementazione dell'algoritmo A*

L'algoritmo A* combina la ricerca a costo uniforme (Dijkstra) con un'euristica ammissibile che guida l'esplorazione verso la destinazione.

5.2.1 Euristica utilizzata

L'euristica $h(n)$ stima il costo rimanente dal nodo n alla destinazione. Nel sistema viene impiegata la distanza euclidea 3D:

$$h(n) = \sqrt{(x_{\text{target}} - x_n)^2 + (y_{\text{target}} - y_n)^2 + (z_{\text{target}} - z_n)^2}$$

Questa euristica è ammissibile (non sovrastima mai il costo reale) e consistente, garantendo che A* trovi sempre il percorso ottimale.

5.2.2 Strutture dati

Struttura	Scopo
<code>openSet</code>	Insieme dei nodi di frontiera ancora da esplorare.
<code>gScore[n]</code>	Costo effettivo dal nodo <code>start</code> al nodo <code>n</code> .
<code>fScore[n]</code>	Stima del costo totale: $g(n) + h(n)$.
<code>cameFrom[n]</code>	Predecessore di <code>n</code> utilizzato per ricostruire il percorso.

Table 3: Strutture dati principali utilizzate dall'algoritmo A*.

5.2.3 Pseudocodice

Algorithm 1 A* pesato per il pathfinding contestuale

```

1: Input: startNode, targetNode
2: Output: percorso ottimale oppure null
3:
4: inizializza openSet con {startNode}
5: imposta gScore[startNode] ← 0
6: imposta fScore[startNode] ← h(startNode, targetNode)
7: while openSet non è vuoto do
8:   current ← nodo in openSet con fScore minimo
9:   if current == targetNode then
10:    return RicostruiscePercorso(cameFrom, current)
11:   end if
12:   rimuovi current da openSet
13:   for all neighbor in current.outgoingConnections do
14:     salta se neighbor.isBlocked o (neighbor è POI e neighbor ≠ targetNode)
15:     baseCost ← Distanza(current, neighbor)
16:     zoneMultiplier ← GetZoneMultiplier(neighbor)
17:     weightedCost ← baseCost × zoneMultiplier
18:     tentativeG ← gScore[current] + weightedCost
19:     if tentativeG < gScore[neighbor] then
20:       cameFrom[neighbor] ← current
21:       gScore[neighbor] ← tentativeG
22:       fScore[neighbor] ← tentativeG + h(neighbor, targetNode)
23:       if neighbor non è in openSet then
24:         aggiungi neighbor a openSet
25:       end if
26:     end if
27:   end for
28: end while
29: return null                                ▷ percorso non trovato

```

5.3 Fattori di Influenza sul Pathfinding

5.3.1 Policy di Guida

Le policy sono definite nel Knowledge Graph Neo4j e determinano le preferenze di percorso. Ogni policy "soffre" (STRUGGLES_ON) su determinati tipi di superficie stradale, che vengono penalizzati nel calcolo A*.

Policy Comfort

Zona	Superficie	Moltiplicatore	Motivazione
Centro Storico	Sanpietrino (pavé)	2.5x	Le pavimentazioni in pietra causano vibrazioni continue.
Zona Industriale	Strada dissestata	2.5x	Buche e irregolarità provocano scossoni.
Zona Residenziale	Dossi rallentatori	2.5x	I dossi rallentatori creano sobbalzi.
Zona Suburbana	Asfalto liscio	1.0x	Strade moderne ben asfaltate, nessuna penalità.

Table 4: Penalizzazioni di superficie per la policy di guida *Comfort*.

Policy Sport

Zona	Superficie	Moltiplicatore	Motivazione
Zona Residenziale	Dossi rallentatori	2.0x	I dossi obbligano a rallentare bruscamente.
Zona Industriale	Strada dissestata	2.0x	Le buche limitano la velocità massima.
Centro Storico	Sanpietrino (pavé)	1.0x	Le vibrazioni non penalizzano la dinamica di guida.
Zona Suburbana	Asfalto liscio	1.0x	Strade scorrevoli, nessuna penalità.

Table 5: Penalizzazioni di superficie per la policy di guida *Sport*.

Policy Eco

Zona	Superficie	Moltiplicatore	Motivazione
Centro Storico	Sanpietrino (pavé)	1.0x	Nessuna penalità basata sulla superficie.
Zona Industriale	Strada dissestata	1.0x	Nessuna penalità basata sulla superficie.
Zona Residenziale	Dossi rallentatori	1.0x	Nessuna penalità basata sulla superficie.
Zona Suburbana	Asfalto liscio	1.0x	Nessuna penalità basata sulla superficie.

Table 6: Penalizzazioni di superficie per la policy di guida *Eco*.

5.3.2 Condizioni meteorologiche

Le condizioni meteo modificano i moltiplicatori tramite la relazione **MAKES_HAZARDOUS**, con i seguenti effetti sulle diverse zone:

Zona	Superficie	Moltiplicatore	Motivazione
Centro Storico	Sanpietrino	3.0x	Il sanpietrino bagnato diventa estremamente scivoloso, aumentando il rischio di sbandamento.
Zona Industriale	Strada dissestata	2.0x	Le buche si riempiono d'acqua diventando fangose e meno visibili.
Zona Residenziale	Dossi rallentatori	1.5x	I dossi bagnati riducono l'aderenza in frenata.
Zona Suburbana	Asfalto liscio	1.2x	L'asfalto liscio ha buon drenaggio, con impatto minimo sulle condizioni di guida.

Table 7: Effetti della pioggia sui moltiplicatori di costo per le diverse zone.

5.3.3 Eventi temporali

I contesti temporali generano congestione tramite la relazione **CONGESTS**, applicando moltiplicatori di costo in specifici intervalli orari:

Evento	Orario	Zona impattata	Moltiplicatore
SchoolEntry	7:30–8:30	Centro Storico	10.0x
SchoolExit	13:00–14:00	Centro Storico	10.0x
MatchNight	20:00–23:00	Centro Storico	20.0x
FactoryShift_Morning	7:00–8:00	Zona Industriale	5.0x
FactoryShift_Evening	17:00–18:00	Zona Industriale	5.0x

Table 8: Eventi temporali e relativi moltiplicatori di congestione.

5.3.4 ZTL (Zona a Traffico Limitato)

La ZTL rappresenta un *bonus* per il taxi autonomo, modellato tramite la relazione GRANTS_PERK. A differenza dei fattori precedenti, questo *riduce* il costo di attraversamento:

Parametro	Valore
Zona	Centro Storico
Orario attivo	8:00–20:00
Moltiplicatore	0.6x

Table 9: Parametri della ZTL applicati al Centro Storico.

Il moltiplicatore 0.6x indica che il traffico nella ZTL è ridotto del 40% rispetto al normale grazie alle restrizioni di accesso per i veicoli privati.

5.4 Formula del costo dell’arco

$$\text{EdgeCost}(u \rightarrow v) = \text{Distance}(u, v) \times M_{\text{zone}}(v)$$

dove

$$\text{Distance}(u, v) = \sqrt{(x_v - x_u)^2 + (y_v - y_u)^2 + (z_v - z_u)^2}$$

e $M_{\text{zone}}(v)$ è il moltiplicatore combinato associato alla zona del nodo v .

5.5 Moltiplicatore combinato

La funzione `get_zone_multipliers_with_context()` calcola il moltiplicatore combinato per ciascuna zona, tenendo conto congiuntamente di tutti i fattori che influenzano il costo di attraversamento. La query riportata di seguito viene eseguita sulla base di conoscenza Neo4j e applica la seguente formula:

$$M_{\text{zone}}(v) = M_{\text{policy}}(v) \times M_{\text{weather}}(v) \times M_{\text{time}}(v) \times M_{\text{perk}}(v)$$

```

MATCH (z:Zone)

// 1. Policy Base
OPTIONAL MATCH (p:Policy {name: $policy})-[r1:STRUGGLES_ON]->(z)
WITH z, COALESCE(r1.penalty, 1.0) AS PolicyPenalty

// 2. Weather Malus
OPTIONAL MATCH (c:Context {name: $weather})-[r2:MAKES_HAZARDOUS]->(z)
WITH z, PolicyPenalty, COALESCE(r2.risk_factor, 1.0) AS WeatherPenalty

// 3. Time/Event Congestion
// Cerca contesti attivi ORA (start <= hour <= end)
OPTIONAL MATCH (t:Context)-[r3:CONGESTS]->(z)
WHERE t.start <= $hour AND $hour <= t.end

// Raccogliamo i nomi degli eventi attivi
WITH z, PolicyPenalty, WeatherPenalty,
    collect(t.name) AS TimeEvents,
    MAX(COALESCE(r3.multiplier, 1.0)) AS TimePenalty

// 4. ZTL Perk (Time Dependent)
// Controlla se c'è un benefit ZTL attivo ORA
OPTIONAL MATCH (ztl:Context)-[r4:GRANTS_PERK]->(z)
WHERE ztl.start <= $hour AND $hour <= ztl.end
WITH z, PolicyPenalty, WeatherPenalty, TimePenalty, TimeEvents,
    COALESCE(r4.traffic_reduction, 1.0) AS PerkFactor

// Calcolo Finale
// Base * Weather * Time * Perk
RETURN z.id AS zone_id,
       z.name AS zone_name,
       PolicyPenalty,
       WeatherPenalty,
       TimePenalty,
       TimeEvents,
       PerkFactor,
       (PolicyPenalty * WeatherPenalty * TimePenalty * PerkFactor) AS final_multiplier

```

6 Chat UI

L’interfaccia utente costituisce il canale principale di interazione tra passeggero e taxi autonomo. La UI web si collega al backend FastAPI tramite WebSocket e usa HTTP per alcune funzioni di supporto (ricerca POI, streaming musicale, impostazioni LLM). Le risposte in chat seguono la pipeline di interpretazione descritta nella Sezione 7, “LLM e Tooling Conversazionale”.

6.1 Architettura di comunicazione

La UI si connette al WebSocket `/ws` e invia messaggi strutturati con `session_id` (e, durante la corsa, `ride_id`). Le risposte dal backend sono in formato `assistant_response` con testo, opzioni UI e comandi. In parallelo, la UI usa endpoint HTTP per: `/api/pois/search` (autocomplete POI), `/music/{genre}` (audio) e `/llm/models`, `/llm/set-model` (impostazioni LLM).

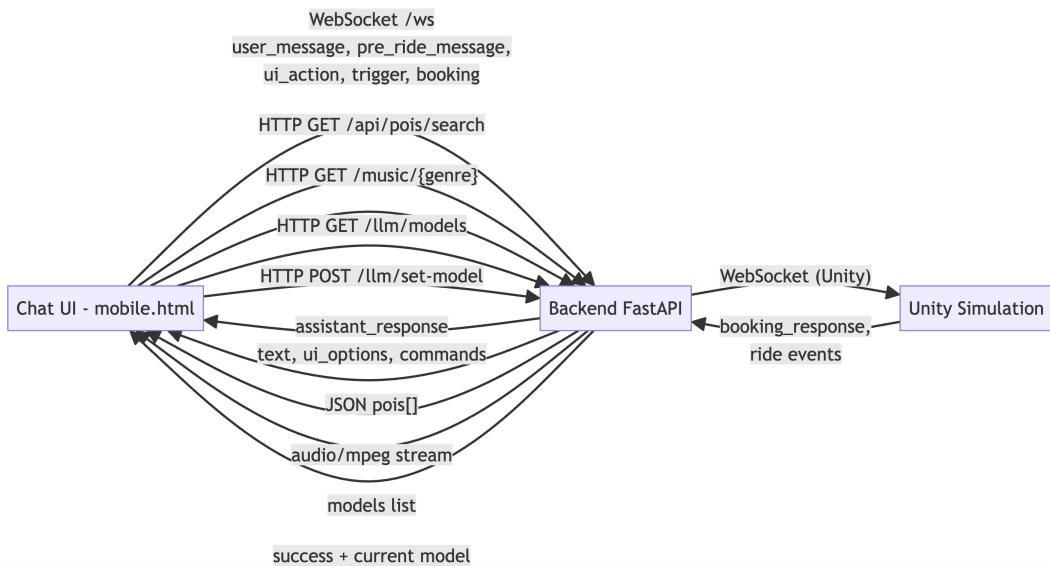


Figure 13: Schema dei canali di comunicazione tra Chat UI, backend FastAPI e simulazione Unity.

6.2 Booking (Pre-ride)

6.2.1 Selezione destinazione

La destinazione può essere scelta in due modi: *Cerca* (autocomplete POI tramite ricerca testuale) oppure *Chiedi* (mini-chat con STT integrato). Nel secondo caso la UI invia messaggi `pre_ride_message` e riceve risposte con opzioni selezionabili; quando un POI viene scelto, la destinazione è mostrata in basso e si abilita la prenotazione.

6.2.2 Prenotazione corsa

Alla conferma l'utente invia una `richiesta_prenotazione` contenente destinazione, `user_id` e policy di guida selezionata. Il backend inoltra la richiesta a Unity e ritorna un acknowledgement in attesa della risposta. In caso di condizioni utente particolari (es. gravidanza), la policy richiesta può essere forzata a *Comfort*, con messaggio esplicativo.

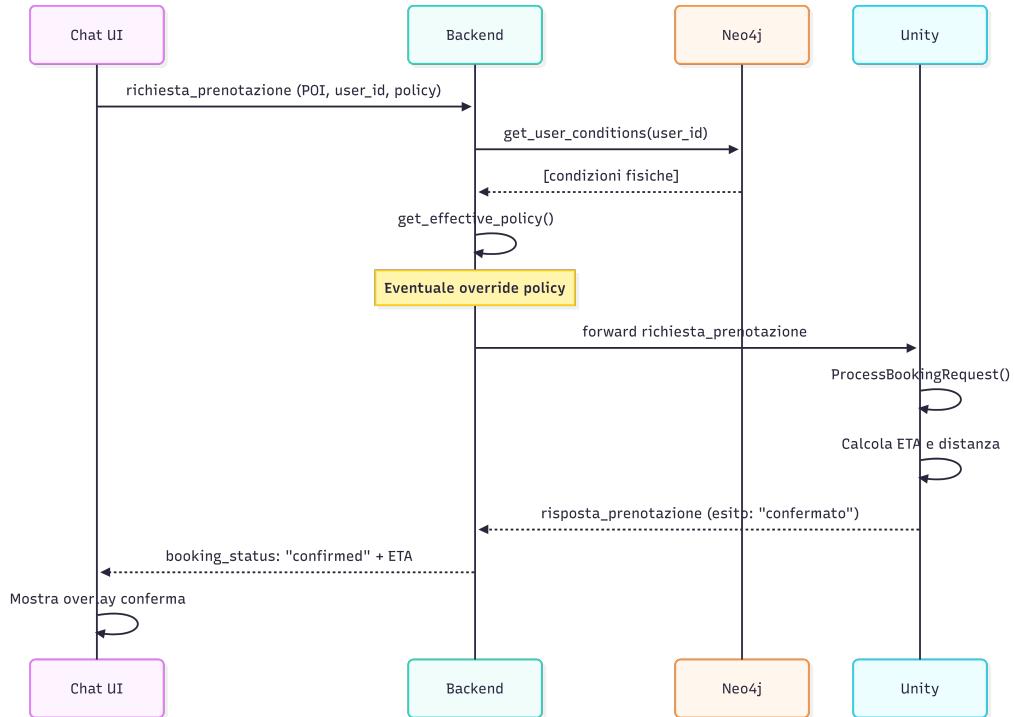


Figure 14: Sequenza della richiesta di prenotazione.

6.2.3 Attesa in coda

Se il taxi è occupato, Unity restituisce lo stato di coda e l'interfaccia mostra una richiesta di conferma. In caso di accettazione, la UI visualizza posizione e tempo stimato, aggiornati con messaggi `queue_update`.

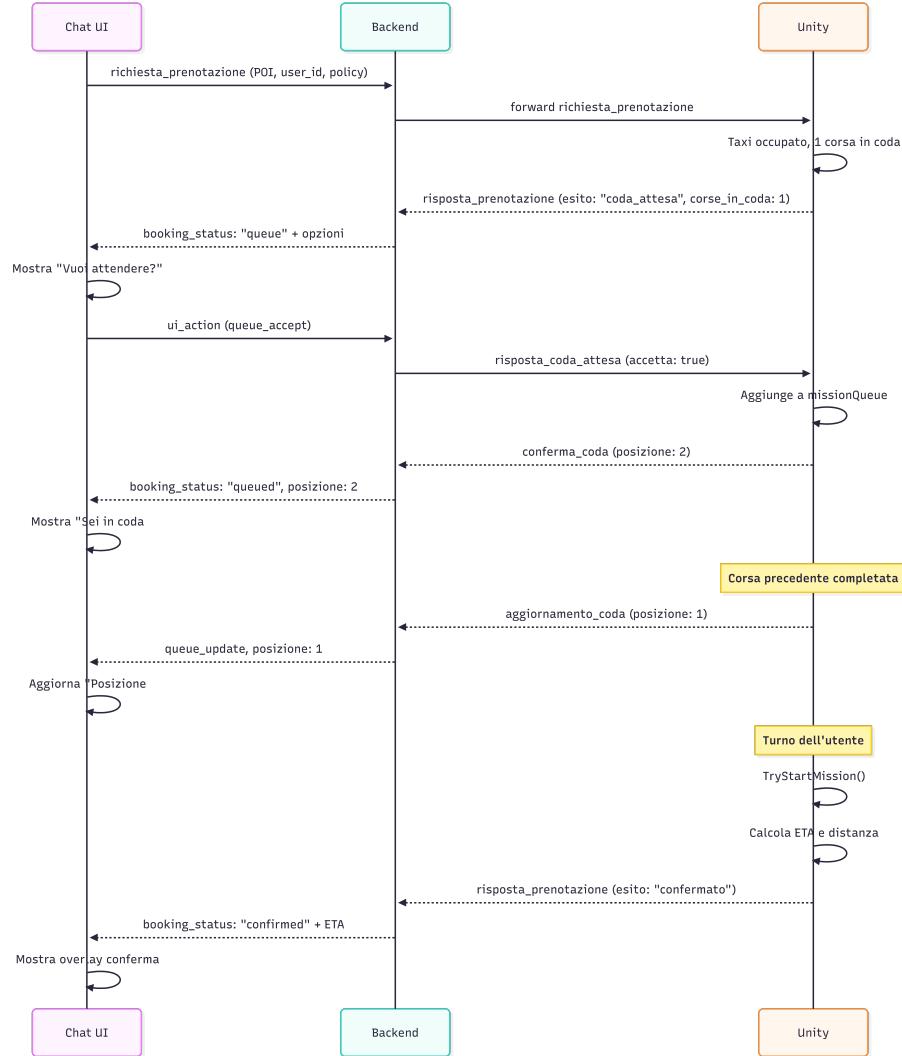


Figure 15: Sequenza dell'attesa in coda.

6.2.4 Annullamento prenotazione

Durante l'attesa o la coda è disponibile il pulsante *Annulla prenotazione*, che invia `annulla_prenotazione`. Rifiutare la coda produce lo stesso effetto. Unity conferma l'annullamento e la UI ritorna allo stato iniziale.

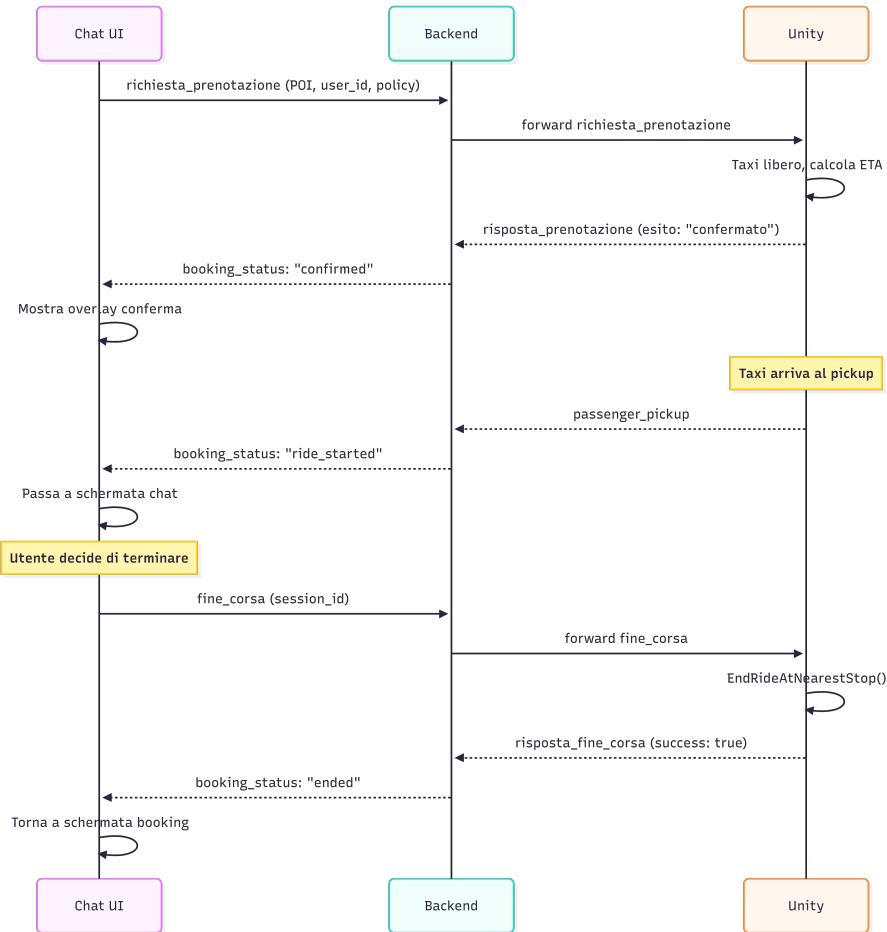


Figure 16: Sequenza dell'annullamento della prenotazione.

6.3 Interazioni durante la corsa

6.3.1 Cambio destinazione

Durante la corsa l'utente può richiedere un cambio destinazione via chat testuale o vocale. Il backend interpreta la richiesta con tool LLM, genera un comando **REROUTE_TO** e inoltra a Unity una richiesta **cambio_destinazione**. La UI aggiorna il banner destinazione con il nuovo POI.

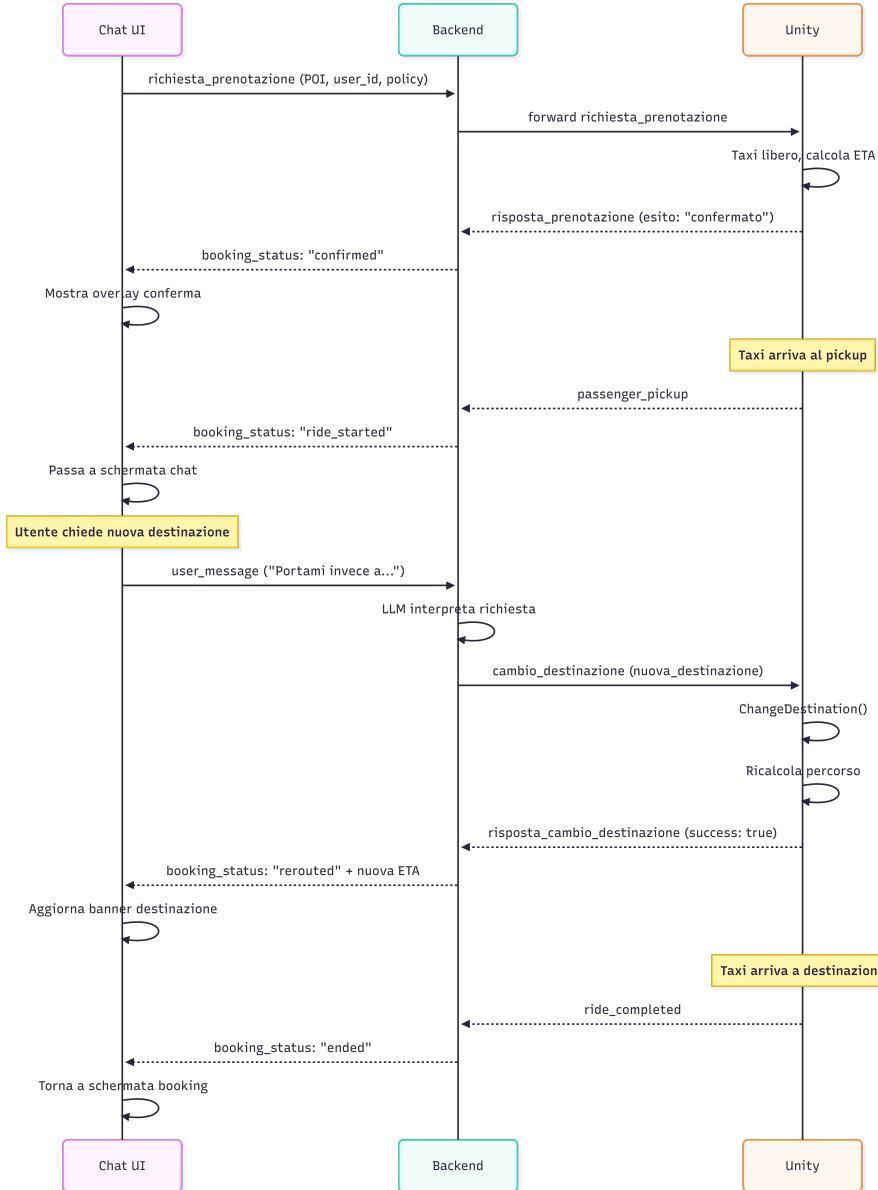


Figure 17: Sequenza del cambio destinazione.

6.3.2 Fine corsa

L'utente può terminare la corsa con il pulsante *Fine Corsa*, che invia il trigger **RIDE_END**. La UI attende la conferma di Unity; in caso positivo, riceve il comando **redirect_to_main** e torna alla schermata di prenotazione.

6.4 Voice AI

6.4.1 Speech-to-Text (STT)

La UI usa l'API `SpeechRecognition` del browser, configurata in `it-IT`. I risultati intermedi sono mostrati in tempo reale e, al termine, il testo viene inviato automaticamente al backend.

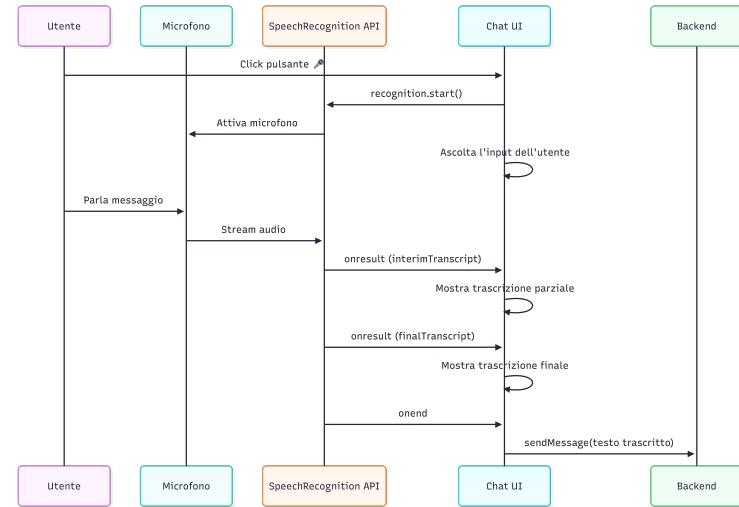


Figure 18: Sequenza della funzionalità STT.

6.4.2 Text-to-Speech (TTS)

I messaggi del taxi possono essere letti ad alta voce tramite `SpeechSynthesis`. La gestione è delegata a un `ttsManager` con coda massima di 30 segmenti; per evitare sovrapposizioni tra tab, un lock distribuito è gestito via `localStorage`. L'utente può attivare/disattivare la lettura vocale nelle impostazioni.

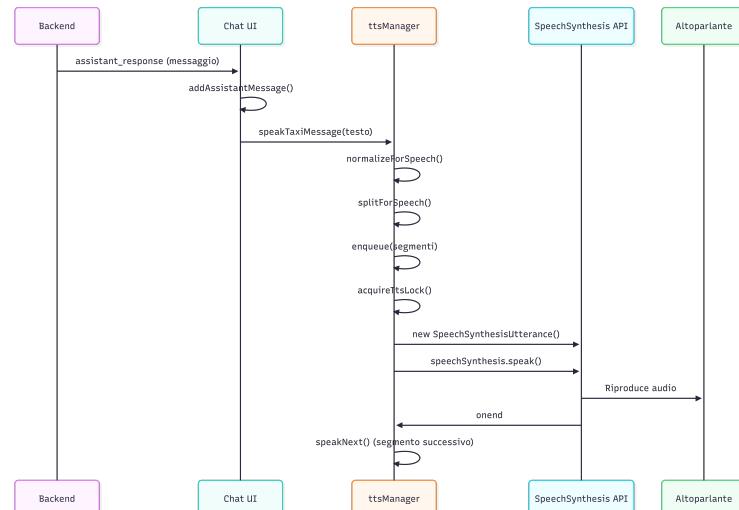


Figure 19: Sequenza della funzionalità TTS.

6.5 Music Streaming

6.5.1 Riproduzione musicale

All'avvio della corsa il sistema propone la musica. Se esiste una preferenza salvata, la riproduzione parte automaticamente; altrimenti viene richiesta la selezione del genere. La UI riceve il comando `PLAY_MUSIC` e riproduce lo stream da `/music/{genre}`.

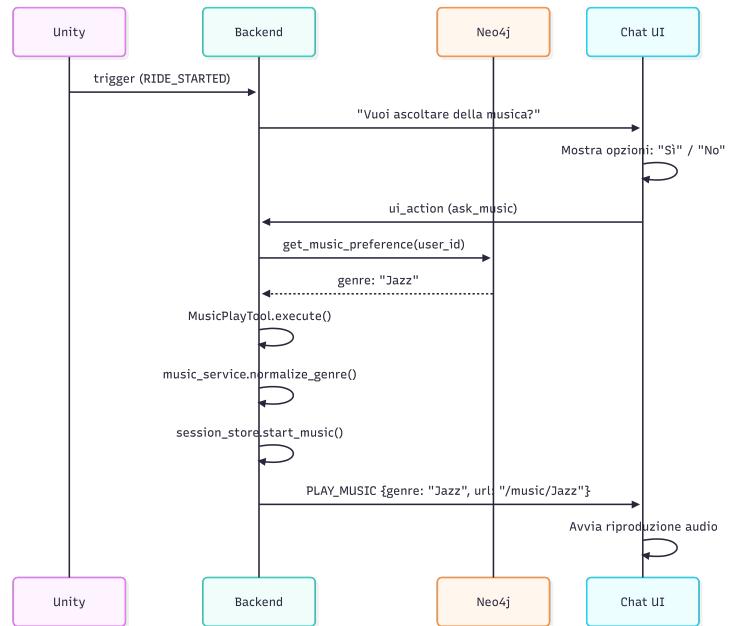


Figure 20: Sequenza della riproduzione musicale.

6.5.2 Cambio di genere e controlli

Il cambio di genere avviene tramite comandi o pulsanti, gestiti dal backend e tradotti in `PLAY_MUSIC` con il nuovo genere. La UI supporta anche pausa, stop e volume con `PAUSE_MUSIC`, `STOP_MUSIC`, `SET_VOLUME`.

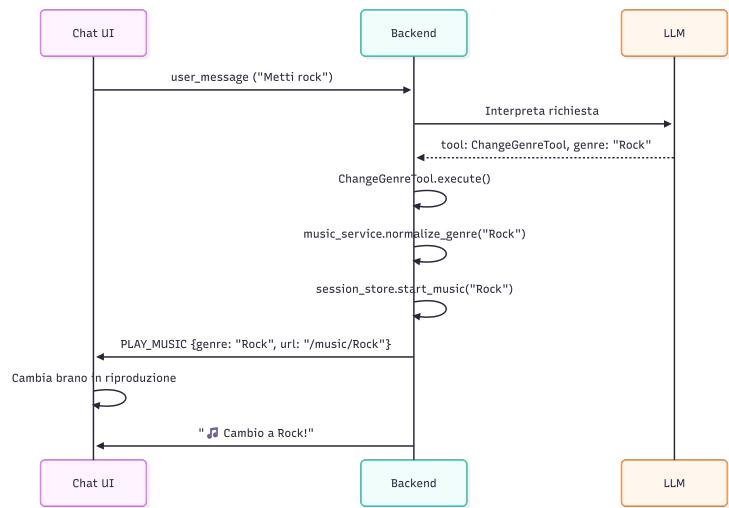


Figure 21: Sequenza del cambio di genere musicale.

6.6 Policy di guida

Le policy disponibili sono *Comfort*, *Sport* ed *Eco*. L'utente può scegliere in fase di prenotazione o durante la corsa. Inoltre in chat vengono mandati tutti i messaggi di explainability che riguardano il ricalcolo del percorso quando c'è un cambio di policy.

6.6.1 Selezione pre-ride

Durante il booking l'utente sceglie la policy tramite i pulsanti dedicati. Se il profilo utente presenta condizioni che richiedono maggiore sicurezza, il backend forza *Comfort* e notifica l'override.

6.6.2 Modifica durante la corsa

L'utente può chiedere un cambio policy in chat testuale o vocale; il backend interpreta la richiesta, verifica eventuali vincoli di sicurezza e invia a Unity il comando `cambio_policy`.

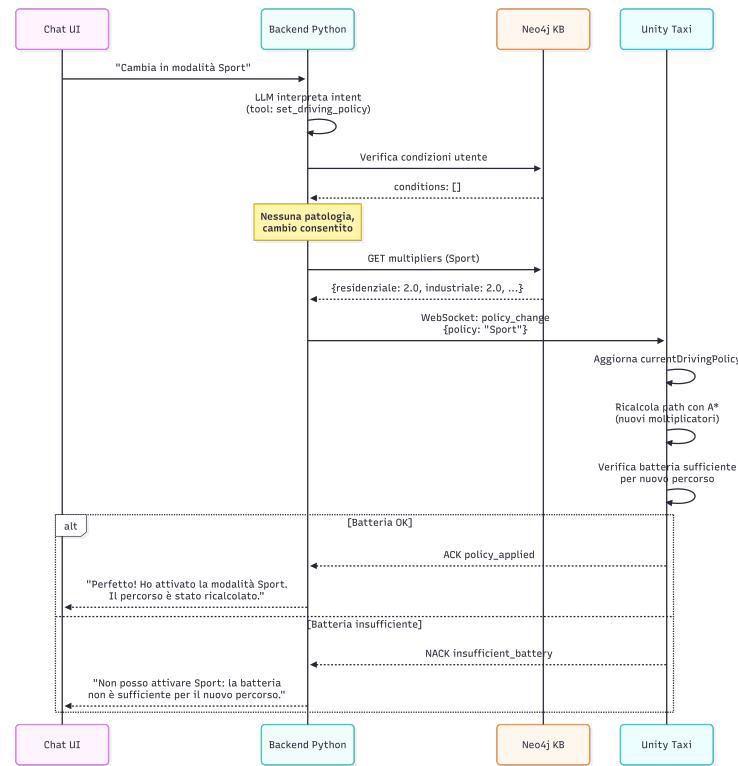


Figure 22: Sequenza del cambio di policy durante la corsa.

6.6.3 Cambi automatici e explainability

In condizioni operative critiche (es. energia insufficiente), Unity può forzare la policy e inviare un messaggio di explainability. La UI mostra questi messaggi come notifiche di sistema in chat.

6.7 Impostazioni LLM e Voce

La UI integra una schermata di impostazioni per selezionare provider e modello LLM, salvati in `localStorage` e applicati al backend tramite `/llm/set-model`. Nella stessa schermata è presente il toggle per abilitare/disabilitare la lettura vocale (TTS).

7 LLM e Tooling Conversazionale

7.1 LLM: modelli e parametri

Il sistema è basato su un’architettura multi-provider che supporta due fornitori di modelli linguistici, Ollama e OpenRouter. Il provider viene scelto a runtime, mantenendo invariata l’interfaccia applicativa.

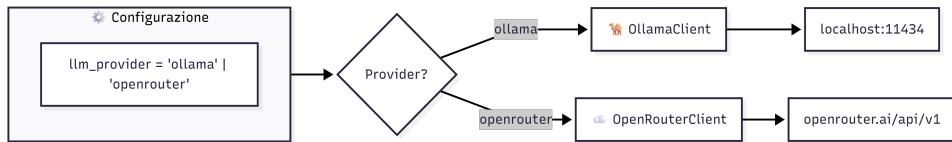


Figure 23: Architettura multi-provider per l’integrazione di più LLM.

7.2 Strategia di Prompting

Il backend interpreta le richieste dell’utente adottando una strategia di **multi-stage fallback** basato su una pipeline di filtri lessicali e più stadi decisionali basata su modelli linguistici.

7.2.1 Filtri lessicali di pre-classificazione

Prima di invocare l’LLM, il sistema applica un layer di filtri lessicali deterministici con il compito di riconoscere pattern ricorrenti e, quando attivati, gestiscono la richiesta in modo diretto, senza chiamare il modello, migliorando accuratezza e latenza.

- **Negation Filter:** utilizza espressioni regolari per individuare la presenza di negazioni all’interno di messaggi articolati, per prevenire l’esecuzione di azioni non desiderate e richiedere eventuali chiarimenti prima di procedere. Il filtro si basa sui seguenti pattern:

```
NEGATION_PATTERNS = [
    r"\bnon\s+",           # "non voglio", "non ho"
    r"\bniente\s+",        # "niente musica"
    r"\bno\b(?!s+grazie)", # "no" ma NON "no grazie"
    r"\bmai\b",            # "mai"
    r"\bsenza\s+",          # "senza musica"
    r"\bnon\b"              # "non" anche senza spazio
]
```

- **Fast Interaction Filter:** rileva, all’interno di risposte brevi, conferme e rifiuti espressi dall’utente mediante il controllo dei confini di parola (*word boundary*). Il riconoscimento avviene sulla base delle seguenti liste:

```

# Conferme (max 3 parole, nessuna negazione)
SIMPLE_CONFIRMATIONS = [
    "ok", "okay", "sì", "si", "va bene", "perfetto", "grazie", "d'accordo", "capito",
    "certo", "esatto", "giusto", "benissimo", "ottimo", "fantastico", "eccellente"
]

# Rifiuti (max 5 parole)
SIMPLE_REJECTIONS = [
    "no", "no grazie", "niente", "lascia stare", "annulla", "non importa", "non serve",
    "lascia perdere", "basta così"
]

```

- **Help & Discovery Filter:** mediante sub-string matching, è in grado di identificare richieste di aiuto e di presentare all'utente le funzionalità disponibili, sulla base del seguente elenco di parole chiave:

```

help_patterns = [
    "aiuto", "help", "cosa puoi fare", "cosa sai fare",
    "come funziona", "che puoi fare", "cosa fai",
    "quali sono le opzioni", "menu", "comandi"
]

```

- **Greeting Filter:** tramite *sub-string matching*, rileva messaggi di saluto brevi e risponde con un messaggio di benvenuto e opzioni rapide. Il riconoscimento avviene sulla base delle seguenti parole chiave:

```

greetings = [
    "ciao", "salve", "buongiorno", "buonasera", "buonanotte",
    "hey", "ehi", "hello", "hi"
]

```

7.2.2 Classificazione dei Tools

Rappresenta il primo stadio dell'architettura decisionale, attivato dalla funzione `classify_with_tools()` quando i filtri lessicali non vengono intercettati nella fase precedente. Consiste nella classificazione del messaggio dell'utente in uno dei tool disponibili del sistema, integrando regole per il riconoscimento di negazioni esplicite e mantenendo il contesto necessario a produrre risposte coerenti.

```

TOOL_CLASSIFICATION_PROMPT = """Sei l'assistente di un taxi autonomo.  

Analizza il messaggio e scegli il tool corretto.

CONTESTO CORRENTE: {context}

TOOL DISPONIBILI:  

{tools}

MESSAGGIO UTENTE: "{message}"

==== REGOLE CRITICHE ===

1. NEGAZIONI: Se il messaggio contiene "non", "no", "niente",  

rispondi con tool_id "none"  

- "non ho fame" → {"tool_id": "none"}

2. MUSICA = ASCOLTARE durante il viaggio  

- "metti musica", "voglio ascoltare" → music_play  

- "ferma/spegni la musica" → music_stop

3. BISOGNI (poi_need): Mappa alle seguenti categorie:  

- CIBO: "fame", "mangiare", "pizzeria", "ristorante" → need="Fame"  

- BERE: "sete", "bere", "bar", "caffè", "drink" → need="Sete"  

- SPESA: "spesa", "supermercato", "latte", "pane" → need="Spesa"  

- SHOPPING: "regalo", "vestiti", "negozi", "comprare" → need="Shopping"  

- SALUTE: "sto male", "farmacia", "medicina", "ospedale"  

→ need="Salute" (o "Malessere" se farmacia)  

- CINEMA: "vedere un film", "cinema", "andare al cinema" → need="Cinema"  

- FITNESS: "palestra", "allenarmi", "fitness", "sport" → need="Fitness"  

- SOLDI: "prelevare", "banca", "bancomat", "soldi", "contanti"  

→ need="Denaro"  

- DORMIRE: "hotel", "dormire", "alloggio", "stanza" → need="Alloggio"  

- RELAX: "parco", "giardino", "passeggiata", "mare" → need="Relax"  

- CULTURA: "museo", "mostra", "arte", "libreria" → need="Cultura"  

- DIVERTIMENTO: "annoio", "divertirmi", "svago" → need="Divertimento"  

- LAVORO: "lavoro", "ufficio", "fabbrica", "andare a lavorare"  

→ need="Lavoro"  

- MECCANICO: "meccanico", "auto guasta", "officina", "riparazione"  

→ need="Meccanico"

4. RISPOSTE SPECIFICHE (poi_tag):  

- "voglio una pizza" → poi_tag con tag="pizza"  

- "voglio un hamburger" → poi_tag con tag="hamburger"

5. NAVIGAZIONE DIRETTA (poi_direct):  

- "portami al/alla [nome]" → poi_direct con poi_name

==== FORMATO RISPOSTA ===
Rispondi SOLO con JSON valido:  

{"tool_id": "...", "params": {"...": "..."}, "confidence": 0.0-1.0}

ESEMPI:  

- "ho fame" → {"tool_id": "poi_need", "params": {"need": "Fame"}, "confidence": 0.95}  

- "devo prelevare soldi" → {"tool_id": "poi_need", "params": {"need": "Denaro"}, "confidence": 0.95}  

- "voglio allenarmi" → {"tool_id": "poi_need", "params": {"need": "Fitness"}, "confidence": 0.95}  

- "andiamo al cinema" → {"tool_id": "poi_need", "params": {"need": "Cinema"}, "confidence": 0.95}  

- "cerco un hotel" → {"tool_id": "poi_need", "params": {"need": "Alloggio"}, "confidence": 0.95}  

- "voglio bere qualcosa" → {"tool_id": "poi_need", "params": {"need": "Sete"}, "confidence": 0.95}  

- "portami a casa" → {"tool_id": "poi_direct", "params": {"poi_name": "casa"}, "confidence": 0.95}
"""

```

7.2.3 Classificazione dei Bisogni

Rappresenta il secondo livello della classificazione, attivato dalla funzione `classify_need()`. Nel caso in cui la classificazione di primo livello non abbia avuto esito positivo o presenti un livello di confidenza inferiore a 0.7, questo livello tenta di individuare il bisogno dell'utente tra *Fame*, *Sete*, *Malessere*, *Divertimento* e *Shopping*, al fine di suggerire POI compatibili.

```

NEED_CLASSIFICATION_PROMPT = """Sei un classificatore di intenti.
L'utente è un passeggero in un taxi autonomo.
Analizza il messaggio e determina se esprime un bisogno.

BISOGNI DISPONIBILI:
- Fame: voglia di mangiare (pranzo, cena, colazione, spuntino, ristorante, pizza, panino, torta, dolce)
- Sete: voglia di bere (bar, caffè, bevande, rinfrescarsi, caldo, aperitivo)
- Malessere: problemi di salute (farmacia, medicina, mal di testa, nausea, stare male)
- Divertimento: svago, intrattenimento (cinema, teatro, disco, stadio, giocare, noia)
- Shopping: acquisti, negozi (comprare, regalo, vestiti, scarpe, spesa, supermercato, ingredienti)

REGOLE:
- Se il messaggio esprime chiaramente un bisogno, restituisci il bisogno con alta confidence.
- Se il messaggio è ambiguo ma potrebbe indicare un bisogno, restituisci con bassa confidence.
- Se è un saluto, domanda generica o off-topic, restituisci need=null.
- Per "Fame", identifica anche la subcategory: "colazione" (mattina), "pranzo" (mezzogiorno), "cena" (sera), "spuntino" (leggero).

ESEMPI COMPLESSI:
- "oggi è perfetto per una torta"
-> need="Fame", subcategory="spuntino"
(o Shopping se cerca pasticceria)
- "domani la mia ragazza fa il compleanno"
-> need="Shopping" (regalo)
- "mi annoio a morte"
-> need="Divertimento"
- "ho il frigo vuoto"
-> need="Shopping" (spesa)

Rispondi SOLO con JSON valido, senza altro testo:
{"need": "Fame|Sete|Malessere|Divertimento|Shopping|null",
 "confidence": 0.0-1.0,
 "subcategory": "colazione|pranzo|cena|spuntino|null"}}

Messaggio utente: "{message}"
"""

```

7.2.4 Richiesta Fuori Contesto

Questo rappresenta l'ultimo livello della classificazione, attivato dalla funzione `get_conversational_response()`. Nel caso in cui anche la classificazione dei bisogni non abbia avuto esito positivo o presenti un livello di confidenza inferiore a 0.65, l'LLM ha il compito di rispondere in modo cortese all'utente e di reindirizzarlo verso le funzionalità fornite dal sistema, come la riproduzione di musica o la ricerca di punti di interesse.

```
CONVERSATIONAL_PROMPT = """Sei l'assistente vocale di un taxi autonomo.  
L'utente ti ha scritto qualcosa che NON rientra nei tuoi compiti.
```

NON SEI IN GRADO DI RISPONDERE A:

- Domande di cultura generale (capitali, storia, scienza, ecc.)
- Richieste di intrattenimento (barzellette, storie, giochi)
- Consigli personali o sentimentali
- Qualsiasi cosa non riguardi il viaggio in taxi

PUOI AIUTARE CON:

- Musica durante il viaggio
- Trovare posti dove mangiare o bere
- Destinazioni e punti di interesse

ISTRUZIONI:

1. Riconosci gentilmente che l'utente ha chiesto qualcosa fuori dalle tue competenze.
2. NON rispondere alla domanda.
3. Ricorda cosa PUOI fare per lui durante il viaggio.
4. Sii simpatico ma fermo.

Messaggio utente: "{message}"

```
Rispondi in italiano, max 2 frasi. Usa emoji per essere amichevole.  
"""
```

7.2.5 Supporto alla selezione dell'opzione

Questo componente viene attivato dalla funzione `match_option()` esclusivamente quando l'utente si trova in uno stato di scelta attiva ed è utilizzato per agevolarlo nell'inserimento della propria selezione.

```
OPTION_MATCHING_PROMPT = """Sei un assistente che aiuta a capire  
quale opzione l'utente ha scelto.  
L'utente deve scegliere tra queste opzioni:
```

{options_list}

Analizza il messaggio dell'utente e determina:

1. Se ha scelto un'opzione specifica (restituisci l'ID esatto)
2. Se vuole annullare/rifiutare (parole come "no", "niente", "annulla", "non voglio", "lascia stare")
3. Se non hai capito cosa vuole

REGOLE:

- Cerca corrispondenze anche parziali nei nomi (es. "la terrazza" → "Ristorante La Terrazza")
- Se l'utente dice "il primo", "il secondo" ecc., mappa alla posizione nella lista
- Se l'utente esprime un nuovo bisogno diverso, considera come "unclear"

```
Rispondi SOLO con JSON valido, senza altro testo:  
{"selected_id": "ID_OPZIONE|cancel|null",  
 "action": "select|cancel|unclear"}
```

Messaggio utente: "{message}"

"""

7.3 Pipeline di classificazione delle richieste utente

L'intera pipeline di classificazione mediante la cascata di prompt descritta nella sezione precedente viene riassunta nella figura 24

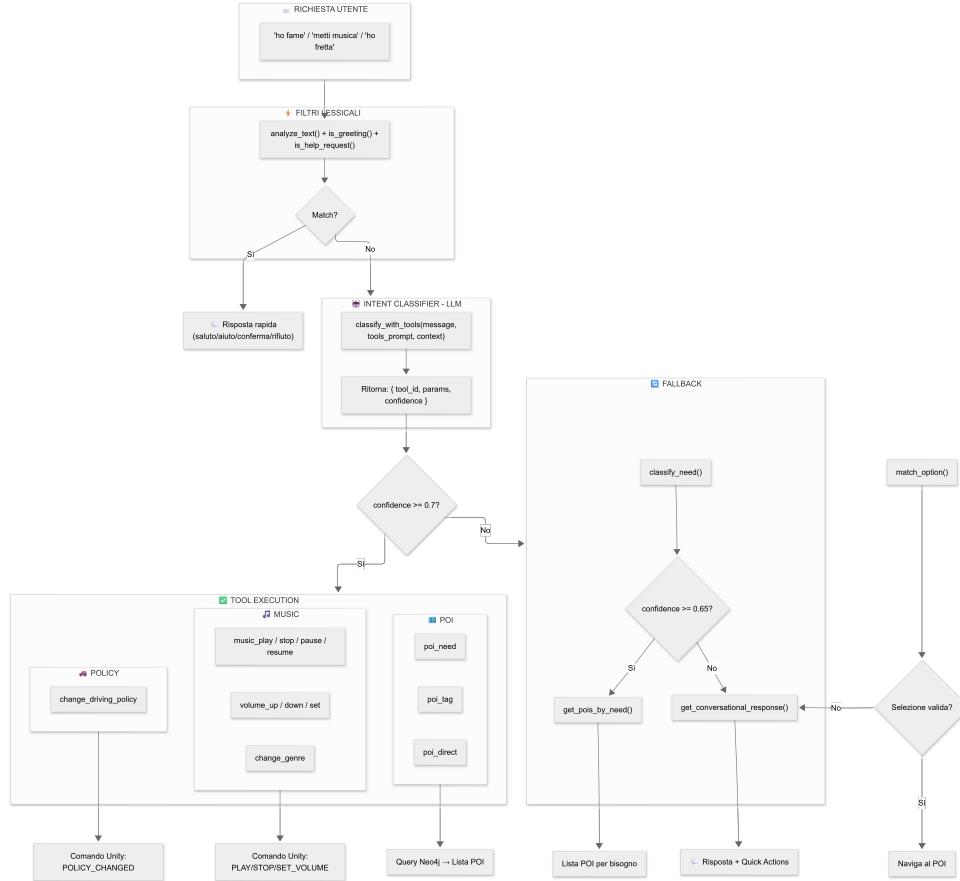


Figure 24: Schema Generale: Richiesta → Classificazione → Risposta

7.4 POI Tools

I POI Tools sono strumenti che gestiscono la ricerca e la selezione di Punti di Interesse (POI) all'interno del sistema di navigazione dell'agente taxi autonomo.

7.4.1 Formula di ranking dei POI

Prima di mostrare i POI all'utente, il sistema li ordina utilizzando uno *score* personalizzato.

$$\text{score} = \text{priorita} \cdot \text{like_boost} \cdot \text{visit_boost} \cdot \text{rating}$$

Spiegazione dei fattori

- **Priorità**: peso della relazione (Bisogno) - [:SUGGERISCE {priorita}] -> (Categoria), utilizzato solo per il calcolo dello score nel caso in cui sia stata effettuata una ricerca dei POI in base ad un bisogno dell'utente, con un valore compreso tra 1 e 10.
- **Like_boost** : assume valore 2.0 se sull'utente e sul POI esiste una relazione PIACE (inserita automaticamente dal sistema quando l'utente ha visitato quel POI almeno tre volte), altrimenti 1.0.
- **Visit_boost** : assume valore 1.5 se l'utente ha una relazione HA_VISITATO sul POI, altrimenti 1.0.

- Rating : valutazione media del POI; se non disponibile viene utilizzato un valore di default pari a 3.0.

7.4.2 Tool di ricerca del POI tramite il bisogno

Il *poi_need* gestisce le richieste basate sui bisogni generici dell'utente (ad esempio fame, sete, malessere). Il tool viene attivato quando la classificazione primaria identifica uno dei bisogni gestiti dal sistema. A questo punto la funzione `get_pois_by_need()` esegue su Neo4j la seguente query, che recupera e riordina i POI in base allo score personalizzato.

```
// Trova categorie suggerite per il bisogno
MATCH (b:Bisogno {nome: $need})-[:SUGGERISCE]-(cat:Categoria)

// Trova POI di quelle categorie
MATCH (poi:PuntoInteresse)-[:HA_CATEGORIA]-(cat)

// Casa dell'utente (serve per filtrare Residenziale in Alloggio)
OPTIONAL MATCH (home_user:Utente {id: $user_id})-[:ABITA]->(home:PuntoInteresse)

// Controlla preferenze PIACE (solo su POI!)
OPTIONAL MATCH (u:Utente {id: $user_id})-[:PIACE]->(poi)

// Controlla visite precedenti
OPTIONAL MATCH (u2:Utente {id: $user_id})-[v:HA_VISITATO]->(poi)

WITH poi, cat, s.priorita AS priorita, home,
CASE WHEN u IS NOT NULL THEN true ELSE false END AS liked,
CASE WHEN v IS NOT NULL THEN true ELSE false END AS visited,
CASE WHEN u IS NOT NULL THEN 2.0 ELSE 1.0 END AS like_boost,
CASE WHEN v IS NOT NULL THEN 1.5 ELSE 1.0 END AS visit_boost

// Per Alloggio: mostra solo la casa dell'utente + hotel (escludi altre case)
WHERE $need <> "Alloggio"
OR cat.nome <> "Residenziale"
OR poi = home

// Calcola score: priorità × boost_piace × boost_visitato × rating
WITH poi, cat.nome AS categoria, priorita, liked, visited,
priorita * like_boost * visit_boost * COALESCE(poi.valutazione_media, 3.0) AS score

RETURN poi.id AS id,
poi.nome AS name,
poi.id_unity AS id_unity,
categoria AS category,
poi.valutazione_media AS rating,
liked,
visited,
score
ORDER BY score DESC
LIMIT $limit
```

7.4.3 Tool di ricerca dei POI tramite tag

Il *poi_tag* gestisce le richieste in cui l'utente esprime un bisogno generico o un desiderio legato a un prodotto/servizio. In pratica, il tool cerca POI associati a un tag specifico, a partire dall'intento riconosciuto dalla classificazione primaria. A questo punto la funzione `get_pois_by_tag()` esegue su Neo4j la seguente query, che recupera e riordina i POI più rilevanti per il bisogno individuato:

```

// Trova POI con il tag
MATCH (poi:PuntoInteresse)-[:HA_TAG]->(t:Tag)
WHERE toLower(t.nome) = toLower($tag)

// Controlla preferenze
OPTIONAL MATCH (u:Utente {id: $user_id})-[:PIACE]->(poi)
OPTIONAL MATCH (u2:Utente {id: $user_id})-[v:HA_VISITATO]->(poi)

WITH poi, t,
CASE WHEN u IS NOT NULL THEN true ELSE false END AS liked,
CASE WHEN v IS NOT NULL THEN true ELSE false END AS visited,
CASE WHEN u IS NOT NULL THEN 2.0 ELSE 1.0 END AS like_boost,
CASE WHEN v IS NOT NULL THEN 1.5 ELSE 1.0 END AS visit_boost

// Ottieni categoria
OPTIONAL MATCH (poi)-[:HA_CATEGORIA]->(cat:Categoria)

WITH poi, cat.nome AS category, liked, visited,
like_boost * visit_boost * COALESCE(poi.valutazione_media, 3.0) AS score

RETURN poi.id AS id,
poi.nome AS name,
poi.id_unity AS id_unity,
category,
poi.valutazione_media AS rating,
liked,
visited,
score
ORDER BY score DESC
LIMIT $limit

```

7.4.4 Tool di ricerca POI per nome

Il *poi_direct* porta l'utente direttamente a un singolo POI specifico, identificato per nome, oppure alla propria abitazione. Per la ricerca di un POI per nome viene utilizzata la funzione `find_poi_by_name()`, che esegue la seguente query Neo4j:

```

MATCH (poi:PuntoInteresse)
WHERE toLower(poi.nome) CONTAINS $token_0
AND toLower(poi.nome) CONTAINS $token_1
-- ... (un CONTAINS per ogni token estratto dal nome)

OPTIONAL MATCH (poi)-[:HA_CATEGORIA]->(cat:Categoria)

RETURN poi.id AS id,
poi.nome AS name,
poi.id_unity AS id_unity,
cat.nome AS category,
poi.valutazione_media AS rating
ORDER BY size(poi.nome)
LIMIT 1

```

Nel caso in cui l'utente chieda di essere portato “a casa”, viene invocata la funzione `get_user_home()`, che esegue:

```

MATCH (u:Utente {id: $user_id})-[:ABITA]->(casa:PuntoInteresse)
OPTIONAL MATCH (casa)-[:HA_CATEGORIA]->(c:Categoria)
RETURN casa.id AS id,
casa.nome AS name,
casa.id_unity AS id_unity,
c.nome AS category,
casa.valutazione_media AS rating

```

7.5 Music Tools

I *music_tools* gestiscono la riproduzione musicale durante la corsa in taxi. Permettono all’utente di avviare, fermare, mettere in pausa la musica e regolare il volume, come mostrato nella tabella 10.

Tool	Trigger	Comando Unity
music_play	“metti musica”, “voglio ascoltare”	PLAY_MUSIC
music_stop	“ferma la musica”, “stop”	STOP_MUSIC
music_pause	“metti in pausa”	PAUSE_MUSIC
music_resume	“riprendi la musica”	RESUME_MUSIC
volume_up	“alza il volume”, “più forte”	SET_VOLUME (+2)
volume_down	“abbassa il volume”, “più piano”	SET_VOLUME (-2)
volume_set	“volume a 5”, “imposta volume 50%”	SET_VOLUME (valore)
change_genre	“cambia genere”, “metti jazz”	PLAY_MUSIC (nuovo genere)

Table 10: Mappatura tra tool, frasi di attivazione e comandi Unity.

Tra i tool elencati nella tabella, non tutti richiedono una query a Neo4j, ma si limitano semplicemente a inviare comandi che verranno eseguiti nell’ambiente Unity. Gli unici due tool che richiedono un’inferenza sono:

- **Riproduzione musicale**

Viene attivato il tool `music_play`. Se l’utente specifica un genere musicale (presente nel campo `params`), questo viene riprodotto direttamente. In caso contrario, viene effettuata una query per determinare il genere preferito dell’utente.

```
MATCH (u:Utente {id: $user_id})-[:PIACE]->(g:GenereMusicale)
RETURN g.nome AS genre
```

Se l’utente ha più generi preferiti, ne viene selezionato uno casualmente. Se non esistono preferenze salvate, vengono mostrate le opzioni disponibili nell’interfaccia.

- **Cambio del genere musicale**

Se l’utente specifica il nuovo genere nel messaggio, questo viene riprodotto immediatamente senza query. In caso contrario, viene mostrato un set di opzioni nell’interfaccia. Una volta selezionato il genere dall’interfaccia, questo viene riprodotto e la preferenza viene salvata nella base di conoscenza.

La query seguente viene utilizzata per aggiornare le preferenze musicali dell’utente dopo che le ha selezionate dall’interfaccia UI:

```
MATCH (u:Utente {id: $user_id})
MERGE (g:GenereMusicale {nome: $genre})
MERGE (u)-[:PIACE]->(g)
RETURN u.id AS id
```

7.6 Tool della modifica dello stile di guida

Il *change_driving_policy_tool* permette all’utente di modificare la modalità di guida, tra quelle indicate nella Tabella 11, del taxi autonomo durante la corsa.

Modalità	Descrizione	Keyword di attivazione
Sport	Guida dinamica e reattiva, orientata alla velocità	<i>fretta, urgente, veloce, sbrigati, corri, rapido, presto, accelera</i>
Comfort	Guida fluida e morbida, orientata al benessere del passeggero	<i>tranquillo, calma, relax, comodo, piano, senza fretta, lentamente</i>
Eco	Guida ottimizzata per il risparmio energetico	<i>ecologico, eco, risparmio, verde, ambiente, green, economico</i>

Table 11: Modalità di guida e relative keyword di attivazione.

Il cambio della policy di guida non è incondizionato: per gli utenti con particolari condizioni fisiche (ad esempio gravidanza), il sistema forza la modalità Comfort per la sicurezza del passeggero.

Nel caso in cui il sistema non riesca a inferire la policy desiderata, vengono mostrate le opzioni disponibili nell’interfaccia. La selezione dell’utente viene poi interpretata tramite la funzione `match_option()`.

7.7 Gestione dei messaggi fuori contesto

Il flusso di gestione dei messaggi fuori contesto viene attivato richiamando la funzione `classify_need()` quando la classificazione primaria basata sui tool non riconosce il messaggio dell’utente, oppure quando, in presenza di opzioni attive, la funzione `match_option()` non riesce a identificare la scelta dell’utente.

7.7.1 Classificazione del bisogno

Se il messaggio non è associabile a un tool specifico, il sistema esegue una seconda analisi chiedendo all’LLM di individuare un eventuale *bisogno implicito* tra categorie predefinite. Se il valore di `confidence` è ≥ 0.65 , il bisogno viene considerato valido e il sistema ricerca i POI pertinenti in Neo4j richiamando la funzione `get_pois_by_need()`.

7.7.2 Risposta conversazionale

Quando anche `classify_need()` non individua un bisogno con confidenza sufficiente, il sistema utilizza la funzione `get_conversational_response()` per generare una risposta puramente conversazionale, nella quale vengono proposte all’utente delle opzioni rapide tramite l’interfaccia, così da permettergli di rientrare nel flusso operativo.

7.8 Explainability

L’explainability nel sistema dell’agente taxi autonomo permette di comunicare all’utente le motivazioni delle decisioni prese dal veicolo. I messaggi vengono generati nell’ambiente Unity e inviati al backend, che li inoltra alla Chat UI.

7.8.1 Flusso di comunicazione dell’explainability

Il sistema di explainability segue un flusso a quattro componenti che permette al taxi autonomo di spiegare all’utente le motivazioni delle proprie decisioni di navigazione.

1. Rilevamento dell'evento

Il processo inizia quando il `TaxiMissionController` rileva un evento che richiede il ricalcolo del percorso. Una volta completato il ricalcolo, il controller emette l'evento `OnRouteRecalculated` passando come parametro un codice identificativo della causa.

2. Gestione e traduzione

Il `MissionCoordinator` riceve la notifica e invoca il metodo `HandleRouteRecalculated()` con il compito di tradurre il codice tecnico in un messaggio comprensibile per l'utente. Il messaggio viene quindi inviato al backend tramite WebSocket utilizzando la funzione `SendExplainabilityMessage()`.

3. Inoltro (backend)

Il backend FastAPI riceve il messaggio tramite la connessione WebSocket dedicata a Unity. La funzione `handle_unity_message()` identifica l'azione come "explainability" e inoltra il contenuto alla Chat UI del passeggero corrispondente, aggiungendo il tag `message_type: "explainability"`.

4. Visualizzazione (Chat UI)

La Chat UI riceve il messaggio e lo presenta all'utente.

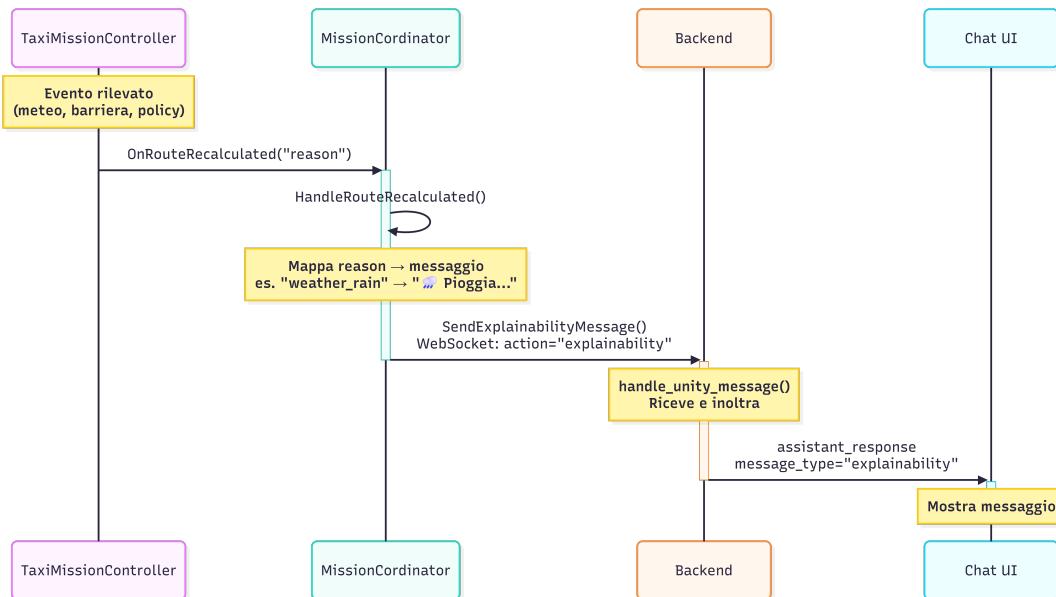


Figure 25: Flusso di comunicazione dell'explainability.

7.8.2 Messaggi dell'explainability

Nella tabella seguente sono riportati i messaggi di explainability e i relativi trigger che ne causano l'attivazione. Per ciascun messaggio, il backend aggiunge automaticamente il tempo di arrivo stimato aggiornato, nel formato “[OROLOGIO] Nuovo tempo stimato: X minuti”.

Categoria	Trigger	Messaggio
Meteo	Inizio pioggia	[PIOGGIA] Pioggia in arrivo! Navigazione aggiornata per condizioni scivolose.
Meteo	Fine pioggia	[SOLE] Meteo migliorato! Navigazione aggiornata alle condizioni ottimali.
Ostacoli	Barriera rilevata	[LAVORI] Ho ricalcolato il percorso per evitare un blocco stradale.
Policy	Cambio policy confermato	[CHECK] Modalità [X] attivata con successo. Il percorso è stato ricalcolato sulla base della policy scelta e potrebbe aver subito variazioni.
Policy	Policy già attiva	Modalità [X] già attiva.
Policy	Switch forzato a ECO (ricalcolo)	[ATTENZIONE] Batteria insufficiente con la policy attuale! Ho attivato la modalità ECO per garantire l'arrivo.
Batteria	Bassa pre-missione	Batteria bassa: mi dirigo alla stazione di ricarica.
Batteria	Insufficiente per cambio policy	Impossibile usare [X]: batteria insufficiente per completare la corsa e tornare alla colonnina.
Batteria	Switch automatico ECO	Batteria insufficiente per la policy attuale. Passo a ECO per completare la corsa.
Batteria	Insufficiente + policy obbligatoria	[ATTENZIONE] Batteria non sufficiente per mantenere la modalità [X] fino a destinazione. Per la tua sicurezza mi fermo alla prossima fermata sicura.
Batteria	Critica + policy obbligatoria	[STOP] Batteria non sufficiente per mantenere la modalità [X] fino a destinazione. Per la tua sicurezza mi fermo alla prossima fermata sicura.
Batteria	Cambio destinazione impossibile	[STOP] Mi dispiace, la batteria non è sufficiente per raggiungere la nuova destinazione. Posso riportarti alla destinazione precedente oppure puoi terminare la corsa qui.
Batteria	Critica (emergenza)	[STOP] Mi scuso per il disagio! A causa del nuovo percorso, la batteria non è sufficiente per completare la corsa. Ti lascerò al punto sicuro più vicino.
Fine corsa	Richiesta anticipata	Richiesta fine corsa: mi fermo al punto sicuro più vicino.
Utente	Override gravidanza	Per la tua sicurezza, utilizziamo la modalità Comfort [DIVANO].

Table 12: Messaggi di sistema in base agli eventi rilevati.