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



TESI DI LAUREA

Progettazione e sviluppo di un assistente vocale basato su Large Language Model per la gestione operativa delle consegne in un sistema di gestione dei trasporti

Design and development of a Large Language Model-based voice assistant for delivery management in a transport management system

Relatore

Prof. Domenico Ursino

Correlatore

Ing. Gianluca Bonifazi

Candidato

Fabio Tempera

ANNO ACCADEMICO 2025-2026

Il computer è lo strumento più straordinario che abbiamo mai inventato.
È l'equivalente di una bicicletta per le nostre menti.

Steve Jobs

Sommario

Negli ultimi anni l'adozione di sistemi di Intelligenza Artificiale nei contesti gestionali operativi ha aperto scenari prima impraticabili, consentendo di ridurre le frizioni tra l'utente e il software in situazioni in cui l'interazione manuale risulta scomoda o impossibile. In questo lavoro viene progettato e realizzato un assistente vocale basato su Large Language Model da integrare nell'applicazione mobile MyDelivery, un sistema TMS sviluppato dall'azienda Infoservice s.r.l. di San Benedetto del Tronto. L'obiettivo principale consiste nel consentire agli autisti consegnatari di controllare l'intera applicazione tramite comandi vocali in linguaggio naturale, senza necessità di interagire manualmente con l'interfaccia touch. A tal fine viene progettata un'architettura a tre livelli completamente disaccoppiata, composta dall'applicazione mobile Flutter, da un server middleware Node.js e dal modello linguistico Claude Haiku di Anthropic. Il sistema riceve la trascrizione del comando vocale, la elabora tramite un prompt di sistema appositamente progettato con tecniche di *few-shot prompting* e *fuzzy matching* dinamico, e restituisce una struttura JSON che l'applicazione traduce in azioni operative dirette. Viene, inoltre, condotta una valutazione sperimentale comparativa tra il modello cloud adottato e un modello locale eseguito tramite Ollama, analizzandone accuratezza, latenza e scalabilità. I risultati della validazione mostrano un'accuratezza del 97% e latenze comprese tra 986 e 2929 millisecondi, compatibili con un uso operativo in tempo reale, con un risparmio fino al 98% sul consumo di token in ingresso nelle sequenze di comandi consecutivi grazie al meccanismo di prompt caching.

Parole chiave: Large Language Model, assistente vocale, prompt engineering, few-shot prompting, fuzzy matching, Flutter, Node.js, Transport Management System, Intelligenza Artificiale

Indice

Introduzione **1**

1 Introduzione ai Large Language Models **3**
1.1 Origini e traiettoria evolutiva . . . 3
1.2 L'architettura Transformer e il meccanismo di attenzione . . . 4
1.3 Dalla previsione del token successivo alla comprensione del linguaggio . . . 4
1.4 Le famiglie di modelli principali . . . 5
1.5 Il prompt engineering e il few-shot prompting . . . 5
1.6 Le API e l'integrazione applicativa . . . 6
1.7 Gli LLM come traduttori di intenti . . . 6

2 L'applicazione MyDelivery **8**
2.1 L'azienda Infoservice . . . 8
2.2 Il sistema MyDelivery: panoramica generale . . . 8
2.3 L'architettura tecnica . . . 9
2.3.1 Il backoffice web . . . 10
2.3.2 L'applicazione mobile . . . 11
2.3.3 Il gateway di autenticazione . . . 12
2.4 Le funzionalità dell'applicazione mobile . . . 13
2.4.1 Accesso e sincronizzazione iniziale . . . 13
2.4.2 La lista dei giri di consegna . . . 13
2.4.3 Il dettaglio del giro di consegna . . . 14
2.4.4 La gestione del singolo documento di consegna . . . 15
2.4.5 La stampa della ricevuta di consegna . . . 19
2.4.6 La cronologia e i report di fine giornata . . . 21
2.4.7 Le impostazioni e la firma dell'autista . . . 22
2.5 Il problema operativo e la necessità dell'assistente vocale . . . 22

3 Analisi dei requisiti **24**
3.1 Il contesto operativo e il profilo degli utenti . . . 24
3.2 I requisiti funzionali . . . 25
3.2.1 Ricezione e interpretazione dei comandi vocali . . . 25
3.2.2 Correzione degli errori di pronuncia tramite fuzzy matching . . . 25
3.2.3 Produzione di output strutturati . . . 26
3.2.4 Copertura delle funzionalità dell'applicazione . . . 26

3.2.5 Supporto al pagamento multiplo . . . 26
3.2.6 Conversione di espressioni numeriche e temporali . . . 26
3.3 I requisiti non funzionali . . . 27
3.3.1 Latenza di risposta . . . 27
3.3.2 Affidabilità dell'output . . . 27
3.3.3 Sicurezza delle credenziali . . . 27
3.3.4 Scalabilità economica . . . 28
3.3.5 Indipendenza dall'infrastruttura esistente . . . 28
3.4 I vincoli progettuali . . . 28
3.5 Il contratto di comunicazione tra i livelli . . . 29
3.6 Gestione del contesto dinamico . . . 29

4 Progettazione **31**
4.1 Il modello a tre livelli . . . 31
4.1.1 Livello 1: l'applicazione mobile (Flutter) . . . 32
4.1.2 Livello 2: il server middleware (Node.js) . . . 32
4.1.3 Livello 3: il motore di Intelligenza Artificiale (Claude) . . . 32
4.1.4 Il flusso operativo end-to-end . . . 33
4.2 La scelta del modello linguistico . . . 33
4.2.1 I criteri di selezione . . . 33
4.2.2 La famiglia Claude di Anthropic . . . 33
4.3 La progettazione del prompt di sistema . . . 34
4.3.1 La struttura del prompt . . . 34
4.3.2 Il fuzzy matching dinamico . . . 34
4.3.3 La memoria operativa e la gestione dello stato . . . 35
4.3.4 La gestione degli errori . . . 35
4.4 La mappatura delle azioni operative . . . 36
4.4.1 La struttura del JSON di risposta . . . 36
4.4.2 Le azioni di navigazione e visualizzazione . . . 37
4.4.3 Le azioni di gestione operativa . . . 37
4.4.4 Le azioni di gestione amministrativa . . . 37
4.4.5 Le azioni di reportistica e impostazioni . . . 39
4.4.6 L'azione di gestione degli errori . . . 39
4.5 Le liste di riferimento e il contesto dinamico . . . 39
4.6 La progettazione del pacchetto Flutter . . . 40
4.6.1 L'architettura interna del pacchetto . . . 40
4.6.2 La macchina a stati del widget . . . 41
4.6.3 Il contratto di integrazione . . . 41
4.7 La sicurezza della comunicazione . . . 41
4.8 L'alternativa locale: SLM e Ollama . . . 42
4.8.1 La motivazione e il contesto . . . 42
4.8.2 Il modello scelto e i criteri di selezione . . . 43

5 Implementazione **44**
5.1 L'infrastruttura server . . . 44
5.1.1 Parametri di accesso . . . 44
5.1.2 Primo accesso e ricognizione del sistema . . . 45
5.2 Configurazione dell'ambiente . . . 45
5.2.1 Aggiornamento del sistema operativo . . . 46
5.2.2 Installazione di Node.js . . . 46
5.2.3 Installazione di PM2 . . . 46

5.2.4 Creazione della cartella di progetto . . . 47
5.3 Il file di configurazione .env . . . 47
5.4 Il codice sorgente del server . . . 48
5.4.1 Inizializzazione e configurazione . . . 48
5.4.2 Il template del prompt di sistema . . . 49
5.4.3 Funzioni di utilità . . . 50
5.4.4 La struttura della richiesta al modello e il caching . . . 52
5.4.5 L'endpoint principale . . . 53
5.4.6 L'endpoint di health check e l'avvio del server . . . 56
5.5 Avvio del server e verifica del funzionamento . . . 57
5.5.1 Avvio con PM2 per l'esecuzione persistente . . . 57
5.5.2 Configurazione del firewall . . . 58
5.6 Test di funzionamento sulla macchina server . . . 58
5.6.1 Test dell'endpoint di health check . . . 58
5.6.2 Primo test del comando vocale e risoluzione dell'errore sul nome del modello . . . 58
5.6.3 Test dopo la correzione del nome del modello . . . 59
5.6.4 Test dell'incasso con pagamento multiplo . . . 60
5.6.5 Test del fuzzy matching su nome cliente impreciso . . . 60
5.6.6 Lo script di test interattivo . . . 61
5.7 Test di funzionamento su dispositivo mobile . . . 65
5.7.1 Il problema della raggiungibilità del server . . . 65
5.7.2 La soluzione tramite tunneling con ngrok . . . 66
5.7.3 Preparazione del progetto Flutter di esempio . . . 66
5.7.4 Configurazione dei permessi Android . . . 67
5.7.5 Test riuscito con ngrok . . . 67
5.8 Il pacchetto Flutter per l'acquisizione vocale . . . 67
5.8.1 La struttura del pacchetto . . . 68
5.8.2 Il flusso interno del widget . . . 68
5.8.3 Il contratto di integrazione . . . 69
5.9 Considerazioni sull'ambiente di test . . . 69
5.10 L'implementazione del server locale con Ollama . . . 70
5.10.1 Installazione di Ollama e download del modello . . . 70
5.10.2 La struttura del server gemello . . . 70
5.10.3 Le iterazioni di ottimizzazione del prompt . . . 72
5.10.4 L'avvio parallelo e la configurazione . . . 75

6 Validazione **76**
6.1 Metodologia di test . . . 76
6.2 Risultati per categoria operativa . . . 77
6.2.1 Navigazione e visualizzazione . . . 77
6.2.2 Gestione operativa delle consegne . . . 77
6.2.3 Gestione amministrativa . . . 78
6.2.4 Reportistica e impostazioni . . . 78
6.2.5 Gestione degli errori . . . 78
6.3 Risultati dei test su dispositivo mobile . . . 78
6.4 Analisi delle prestazioni . . . 79
6.4.1 Tempi di risposta . . . 79
6.4.2 Consumo di token . . . 80
6.5 L'esperimento con il modello locale . . . 81
6.5.1 I risultati dei test e le iterazioni di ottimizzazione . . . 81

6.5.2 Il confronto con Claude Haiku 4.5 84
6.5.3 Valutazione e prospettive 85
6.6 Criticità emerse e limitazioni 86
6.6.1 I limiti dell'approccio con modello locale 87

Conclusioni **88**

Bibliografia **90**

Sitografia **92**

Ringraziamenti **93**

Elenco delle figure

2.1 Schermata di login del pannello di configurazione web di MyDelivery 10
2.2 Sezione di gestione degli utenti nel pannello di configurazione 11
2.3 Sezione di gestione dei parametri nel pannello di configurazione 11
2.4 Sezione di configurazione dei metodi di pagamento nel backoffice 12
2.5 Schermate di login e di sincronizzazione iniziale dell'applicazione mobile . . 13
2.6 Schermata principale dell'applicazione con la lista dei giri di consegna e i filtri di stato 14
2.7 Dettaglio di un giro di consegna 15
2.8 Pagina di gestione del singolo documento di consegna con le azioni disponibili (sinistra) e il riepilogo (destra) 16
2.9 Pannello di gestione dell'esito della consegna con i campi per referente e annotazioni 17
2.10 Interfaccia di gestione degli incassi con la ripartizione per metodo di pagamento 18
2.11 Schermata di visualizzazione delle partite aperte del cliente con possibilità di incasso 19
2.12 Schermata di gestione delle cauzioni e dei resi vuoti con il saldo per articolo . 20
2.13 Dettaglio degli articoli contenuti in un documento di consegna 20
2.14 Esempio di ricevuta di consegna generata da MyDelivery con il riepilogo degli incassi e delle cauzioni 21
2.15 Schermata della cronologia con la selezione del periodo e le opzioni di generazione dei report 22

3.1 Struttura di una richiesta HTTP POST inviata dall'applicazione Flutter al middleware, con la trascrizione vocale e le liste di riferimento della sessione corrente. 30

4.1 Schema dell'architettura a tre livelli: l'applicazione Flutter invia la trascrizione al middleware Node.js, che la inoltra alle API di Claude e restituisce il JSON strutturato. 31
4.2 Macchina a stati del widget `VoiceAssistantButton`; sono evidenziati i cinque stati e le transizioni tra essi in risposta agli eventi del ciclo di elaborazione. 41

Elenco delle tabelle

3.1 Scenari operativi identificati per il sistema di assistente vocale, raggruppati per categoria funzionale. 27

4.1 Riepilogo delle quattordici azioni operative mappate nel prompt di sistema. . 36

6.1 Tempi di risposta misurati durante i test, espressi in millisecondi. 79
6.2 Confronto del consumo di token in ingresso con e senza prompt caching. . . . 81
6.3 Risultati delle quattro iterazioni di ottimizzazione del prompt per Qwen2.5-1.5B. 83
6.4 Confronto tra Claude Haiku 4.5 e Qwen2.5-1.5B locale sui test condotti. 84

Introduzione

La digitalizzazione dei processi logistici, nelle piccole e medie imprese, ha compiuto negli ultimi anni passi significativi; eppure alcune frizioni operative fondamentali restano irrisolte. Una di queste riguarda l'interazione tra gli autisti consegnatari e le applicazioni mobile di cui si servono durante la giornata lavorativa: strumenti progettati per semplificare il lavoro che, nelle condizioni reali del campo, richiedono all'operatore di liberare le mani, guardare lo schermo e digitare con precisione, proprio nei momenti in cui farlo è più scomodo o pericoloso. Mani occupate dai colli, guanti da lavoro, temperatura rigida, guida del veicolo: sono situazioni ordinarie per un consegnatario, non eccezioni. Quando l'interfaccia touch impone una pausa in questi momenti, le operazioni vengono rinviate, dimenticate o registrate in modo impreciso, generando disallineamenti contabili e operativi che il personale amministrativo deve poi correggere manualmente.

È in questo contesto che si colloca il presente lavoro, svolto nell'ambito di un tirocinio presso Infoservice s.r.l., azienda di San Benedetto del Tronto che sviluppa soluzioni gestionali per un'ampia gamma di settori. L'esigenza emersa dall'azienda è chiara: consentire agli autisti di controllare l'intera applicazione MyDelivery usando la sola voce, senza modificare le abitudini operative e senza imporre formati di comando rigidi o parole chiave da memorizzare. La risposta tecnologica a questa esigenza è individuata nei Large Language Model, sistemi di Intelligenza Artificiale capaci di comprendere il linguaggio naturale nelle sue forme più variabili e colloquiali e di tradurlo in strutture dati che un sistema software tradizionale può elaborare direttamente.

Il lavoro si articola nella progettazione e nello sviluppo di un assistente vocale basato sul modello Claude Haiku di Anthropic, integrato nell'ecosistema MyDelivery attraverso un'architettura a tre livelli. Il primo livello è l'applicazione mobile Flutter esistente, arricchita da un pacchetto vocale sviluppato nell'ambito di questo tirocinio che gestisce l'acquisizione audio, la trascrizione tramite il motore nativo del dispositivo e la comunicazione con il livello successivo. Il secondo livello è un server middleware Node.js, che custodisce le credenziali API, assembla il prompt di sistema e funge da intermediario sicuro tra il dispositivo mobile e il modello linguistico. Il terzo livello è il motore di Intelligenza Artificiale, che riceve il prompt completo, interpreta il comando vocale dell'utente, risolve le ambiguità sui nomi dei clienti e dei giri tramite un meccanismo di *fuzzy matching* dinamico e restituisce una struttura JSON che l'applicazione traduce in azioni operative dirette. Il sistema copre quattordici scenari operativi distinti, dalla registrazione degli incassi alla gestione delle cauzioni, dalla navigazione tra le schermate alla stampa delle ricevute. Viene, inoltre, condotta una valutazione sperimentale comparativa tra il modello cloud adottato e un'alternativa locale basata su Ollama e sul modello Qwen2.5-1.5B, al fine di verificarne la reale praticabilità in produzione.

Questa tesi è strutturata come di seguito specificato:

- Il *Capitolo 1* introduce il paradigma dei Large Language Model, ripercorrendo l'evoluzione dai primi modelli statistici alle architetture Transformer, e descrive le tecniche di addestramento, il *prompt engineering*, il *few-shot prompting* e le modalità di integrazione applicativa tramite API.
- Il *Capitolo 2* descrive il contesto aziendale e il sistema MyDelivery nella sua interezza, dall'architettura tecnica composta da backoffice web e applicazione mobile alla mappatura completa delle schermate e delle funzionalità offerte agli autisti, fino all'identificazione del problema operativo che ha motivato il progetto.
- Il *Capitolo 3* presenta l'analisi dei requisiti del sistema vocale, definendo il profilo degli utenti, i requisiti funzionali e non funzionali, i vincoli progettuali imposti dall'azienda e il contratto di comunicazione tra i livelli dell'architettura.
- Il *Capitolo 4* illustra la progettazione dell'architettura a tre livelli, le ragioni della scelta del modello Claude Haiku, il processo iterativo di sviluppo del prompt di sistema e la mappatura completa delle quattordici azioni operative in strutture JSON.
- Il *Capitolo 5* descrive tutte le fasi operative che hanno portato dal progetto a un sistema funzionante e testabile, dalla configurazione della macchina server all'implementazione del middleware, dai test su dispositivo mobile fisico allo sviluppo del pacchetto Flutter, fino all'implementazione del server gemello per la valutazione sperimentale dell'alternativa locale basata su Ollama.
- Il *Capitolo 6* presenta la validazione del sistema, analizzando i risultati per categoria operativa, le prestazioni misurate in termini di latenza e consumo di token, l'impatto del prompt caching e le criticità emerse durante i test su macchina server e su dispositivo mobile Android fisico. Una sezione dedicata presenta, inoltre, i risultati dell'esperimento con il modello locale basato su Ollama: le iterazioni di ottimizzazione del prompt, il confronto diretto con Claude Haiku 4.5 e la valutazione delle prospettive evolutive.

CAPITOLO 1

Introduzione ai Large Language Models

In questo capitolo viene introdotto il paradigma dei Large Language Model, ripercorrendo l'evoluzione che ha portato dai primi modelli linguistici statistici agli attuali sistemi basati su architetture Transformer. Si analizzano i meccanismi fondamentali che consentono a questi modelli di comprendere e generare linguaggio naturale, con particolare attenzione alle tecniche di addestramento, al concetto di prompt engineering e alle modalità di integrazione applicativa tramite API. Infine, si inquadra la categoria di utilizzo rilevante per il presente lavoro, ovvero l'impiego degli LLM come traduttori di intenti in contesti operativi vincolati.

1.1 Origini e traiettoria evolutiva

Il problema della comprensione automatica del linguaggio naturale accompagna la ricerca in Informatica fin dai suoi albori. I primi approcci, sviluppati a partire dagli anni Cinquanta del Novecento, si basavano su regole grammaticali codificate manualmente e su dizionari di associazioni simboliche. Sistemi come ELIZA, presentato da Joseph Weizenbaum nel 1966, dimostravano che una macchina poteva sostenere conversazioni apparentemente coerenti attraverso semplici meccanismi di corrispondenza di pattern, senza alcuna comprensione semantica reale [19].

Negli anni Ottanta e Novanta, l'approccio statistico prese il sopravvento: invece di codificare regole esplicite, i ricercatori iniziarono ad addestrare modelli sulle frequenze di co-occorrenza delle parole in grandi corpus testuali. I modelli *n-gram* e, successivamente, i modelli a variabili latenti come la *Latent Semantic Analysis* (LSA), permisero di catturare relazioni semantiche distribuite su interi documenti [3]. Tuttavia, questi sistemi erano fondamentalmente incapaci di modellare dipendenze a lungo raggio all'interno di una frase: la probabilità di una parola veniva calcolata tenendo conto al massimo delle due o tre parole precedenti, rendendo impossibile la comprensione di strutture sintattiche complesse.

La svolta arrivò con l'introduzione delle reti neurali ricorrenti (RNN) e, in particolare, delle architetture *Long Short-Term Memory* (LSTM) proposte da Hochreiter e Schmidhuber nel 1997 [6]. Queste reti erano in grado di mantenere uno stato interno che propagava informazioni attraverso sequenze di lunghezza arbitraria, superando il limite fondamentale dei modelli statistici tradizionali. L'applicazione delle LSTM alla traduzione automatica, culminata nel paradigma *sequence-to-sequence* introdotto da Sutskever ed altri nel 2014 [15], segnò un salto qualitativo nelle prestazioni dei sistemi di elaborazione del linguaggio naturale.

Il vero punto di discontinuità si verificò, tuttavia, nel 2017, con la pubblicazione del lavoro "Attention Is All You Need" da parte di Vaswani ed altri presso Google Brain [17]. L'architettura *Transformer* ivi descritta abbandonava completamente la struttura ricorrente delle LSTM in favore di un meccanismo di *self-attention* che permetteva a ogni elemento di una sequenza di interagire direttamente con ogni altro elemento, indipendentemente dalla distanza. Questo consentiva un addestramento altamente parallelizzabile su hardware moderno e una modellazione delle dipendenze linguistiche di qualità nettamente superiore.

1.2 L'architettura Transformer e il meccanismo di attenzione

Il Transformer è composto da due blocchi principali: un *encoder*, che elabora la sequenza di input producendo una rappresentazione contestualizzata, e un *decoder*, che genera la sequenza di output un elemento alla volta, condizionato sulla rappresentazione prodotta dall'encoder. Nei modelli generativi moderni, come la famiglia GPT, viene utilizzata soltanto la porzione decoder, che genera testo in modo autoregressivo: ogni nuovo token viene predetto sulla base di tutti i token precedentemente generati.

Il meccanismo centrale dell'architettura è la *scaled dot-product attention*, che opera su tre matrici denominate *query*, *key* e *value*. Data una sequenza di token, ciascun token viene proiettato in questi tre spazi vettoriali. Il punteggio di attenzione tra due token viene calcolato come il prodotto scalare tra la query del primo e la chiave del secondo, normalizzato per la radice quadrata della dimensione dello spazio vettoriale e poi convertito in una distribuzione di probabilità tramite la funzione *softmax*. Il risultato finale per ciascun token è la somma pesata dei valori di tutti gli altri token, con pesi determinati dai punteggi di attenzione [17].

La variante *multi-head attention* replica questo meccanismo in parallelo su più sottospazi vettoriali, consentendo al modello di catturare simultaneamente diversi tipi di relazioni linguistiche: relazioni sintattiche, dipendenze semantiche, coreferenza anaforica. I risultati delle diverse attention head vengono poi concatenati e riproiettati in un'unica rappresentazione.

A questi blocchi di attenzione si affiancano reti neurali *feed-forward* applicate indipendentemente a ciascuna posizione, strati di normalizzazione e connessioni residuali. L'insieme di questi componenti viene ripetuto per un numero variabile di volte, da poche decine, nei modelli più piccoli, fino a oltre cento strati, nei modelli di maggiori dimensioni. La profondità di questa struttura, combinata con la capacità del meccanismo di attenzione di catturare relazioni arbitrariamente complesse, è alla base della notevole capacità espressiva dei modelli moderni.

1.3 Dalla previsione del token successivo alla comprensione del linguaggio

Il paradigma con cui vengono addestrati i grandi modelli linguistici è noto come *self-supervised learning*: il modello impara a prevedere il token successivo in una sequenza, senza bisogno di etichette annotate manualmente. Dato un corpus testuale di dimensioni sufficientemente grandi, questo compito apparentemente banale si rivela straordinariamente ricco di informazione: per predire correttamente la parola successiva il modello deve acquisire implicitamente conoscenze grammaticali, fattuali e di ragionamento, distribuite nei miliardi di parametri della rete.

La fase di *pre-training* su corpus di enorme scala, tipicamente dell'ordine di centinaia di miliardi di token provenienti da pagine web, libri, articoli scientifici e codice sorgente, produce un modello con capacità generali di comprensione del linguaggio [2]. Questo modello di base viene poi raffinato attraverso una fase di *fine-tuning* supervisionato su dati annotati

da annotatori umani, seguita da una procedura di ottimizzazione basata sul feedback umano nota come *Reinforcement Learning from Human Feedback* (RLHF) [12]. L'RLHF consente di allineare il comportamento del modello alle preferenze umane, rendendolo più utile, meno incline a produrre contenuti dannosi e più capace di seguire istruzioni esplicite.

Il risultato di questo processo di addestramento a più fasi è ciò che viene comunemente denominato *Large Language Model* (LLM): un sistema con miliardi di parametri, capace di svolgere un'ampia gamma di compiti linguistici in modalità *zero-shot* o *few-shot*, ovvero senza necessità di addestramento specifico per ciascun compito [2].

1.4 Le famiglie di modelli principali

L'ecosistema degli LLM si è sviluppato rapidamente a partire dal 2018, con la pubblicazione di BERT da parte di Google [4] e di GPT da parte di OpenAI [13]. Questi due modelli hanno inaugurato due filoni distinti: BERT, basato sull'encoder del Transformer, si è rivelato particolarmente efficace nei compiti di classificazione e comprensione del testo; GPT, basato sul decoder, si è affermato come il paradigma dominante per la generazione di testo.

Nel corso degli anni successivi, diverse organizzazioni hanno sviluppato proprie famiglie di modelli. OpenAI ha rilasciato la famiglia Chat GPT, tra cui GPT-4, il primo modello multimodale [10]. Google ha sviluppato PaLM, Gemini [5] e le versioni successive. Meta ha reso disponibili i modelli della famiglia LLaMA in modalità open-weight [16], stimolando una vivace comunità di ricerca e sviluppo applicativo. Anthropic, fondata nel 2021 da ex ricercatori di OpenAI, ha sviluppato la famiglia Claude, progettata con particolare attenzione alla sicurezza e all'affidabilità del comportamento [1].

Ai fini del presente lavoro, la classificazione più rilevante è quella per *dimensione e velocità*: i modelli di maggiori dimensioni offrono capacità di ragionamento superiori ma comportano latenze e costi per chiamata più elevati, mentre i modelli più piccoli e veloci si prestano a scenari operativi in cui la risposta deve essere quasi immediata.

1.5 Il prompt engineering e il few-shot prompting

Una delle caratteristiche più rilevanti degli LLM moderni è la loro sensibilità alle istruzioni fornite attraverso il *prompt*, ovvero il testo di input che precede la generazione. La disciplina nota come *prompt engineering* studia come formulare queste istruzioni in modo da ottenere comportamenti coerenti, accurati e controllabili dal modello [20].

La tecnica del *few-shot prompting*, introdotta da Brown ed altri con GPT-3 [2], consiste nel fornire al modello un numero ridotto di esempi completi di input e output desiderato all'interno del prompt stesso, prima di presentare il caso da elaborare. Questa tecnica si è rivelata estremamente efficace nel guidare il modello verso un formato di output specifico e nel vincolare il suo comportamento a un insieme ben definito di risposte possibili.

Un'evoluzione fondamentale in questo ambito è rappresentata dal *Chain-of-Thought* (CoT) prompting, introdotto da Wei ed altri nel 2022 [18]. Questa tecnica estende il paradigma few-shot (o zero-shot) istruendo il modello a generare esplicitamente una sequenza di passaggi logici intermedi prima di fornire la risposta finale. Il CoT ha dimostrato di migliorare drasticamente le prestazioni degli LLM in compiti complessi che richiedono ragionamento aritmetico, algoritmico o logico-deduttivo, sbloccando capacità latenti che non emergevano con il prompting standard.

A partire dai modelli più recenti, il *Chain-of-Thought* ha smesso di essere una semplice tecnica di prompting per diventare uno standard architetturale. Attualmente, i principali modelli linguistici integrano questo processo in modo nativo: prima di generare l'out-

put finale, elaborano autonomamente una catena di ragionamento latente in background (*inference-time compute*). Ciò permette al sistema di dedicare più risorse computazionali e tempo alla risoluzione di problemi complessi, senza richiedere istruzioni esplicite da parte dell'utente [11].

Una variante particolarmente utile per le applicazioni gestionali è il cosiddetto *system prompt*, un blocco di istruzioni generali, separato dall'input dell'utente, che definisce il ruolo del modello, le sue limitazioni operative e il formato atteso delle risposte. I modelli moderni trattano il system prompt con una priorità più alta rispetto alle istruzioni inserite dall'utente, il che lo rende uno strumento potente per costruire assistenti specializzati con comportamenti determinati e prevedibili [1].

1.6 Le API e l'integrazione applicativa

I principali fornitori di LLM espongono i propri modelli attraverso interfacce REST (*Application Programming Interface*). Un'applicazione che intende sfruttare le capacità di un LLM invia al servizio una richiesta HTTP contenente il prompt e una serie di parametri di configurazione, tra cui il modello da utilizzare, il numero massimo di token da generare ed, eventualmente, la *temperatura*; quest'ultimo è un parametro che controlla il grado di casualità nella generazione, con valori prossimi a zero che favoriscono risposte deterministiche e valori più alti che aumentano la varietà dell'output [9].

Il servizio risponde con un oggetto JSON contenente il testo generato, metadati sulla richiesta elaborata e informazioni sul consumo di token, quest'ultimo direttamente proporzionale al costo della chiamata. Questo modello di consumo *pay-per-token* rende gli LLM accessibili anche a progetti di piccola scala, ma impone di prestare attenzione al dimensionamento del prompt, che incide in modo diretto sul costo complessivo dell'integrazione.

Un aspetto architetturale di rilievo riguarda la custodia delle chiavi API: queste credenziali, analogamente alle password di accesso a un database, devono essere conservate esclusivamente in ambienti server controllati e mai incluse nel codice sorgente distribuito o nelle applicazioni client. La violazione di questo principio espone l'organizzazione a rischi di utilizzo non autorizzato e a costi imprevisti.

1.7 Gli LLM come traduttori di intenti

Un caso d'uso di crescente rilevanza industriale è l'impiego degli LLM come strato di comprensione del linguaggio naturale all'interno di applicazioni gestionali preesistenti. In questo paradigma, il modello non viene utilizzato per generare testo libero bensì per svolgere un compito strutturato: ricevere un'istruzione espressa in linguaggio naturale e produrre in output un oggetto dati strutturato, tipicamente JSON o XML, che un sistema software tradizionale può consumare ed elaborare [14].

Questo approccio presenta vantaggi concreti rispetto alle interfacce vocali basate su riconoscimento di parole chiave (*keyword spotting*) o su grammatiche formali predefinite. Un sistema basato sulle parole chiave richiede che l'utente formuli i comandi in un formato rigido e predeterminato, tipicamente memorizzando una lista di frasi valide. Un LLM, al contrario, è in grado di comprendere formulazioni variabili, colloquiali e imprecise dello stesso intento, correggendo automaticamente piccoli errori di pronuncia o di trascrizione e tollerando l'omissione di informazioni non strettamente necessarie [7].

La capacità di abbinare entità nominate pronunciate dall'utente a stringhe esatte presenti in un elenco di riferimento, operazione nota come *entity linking*, è particolarmente preziosa in contesti gestionali dove i nomi dei clienti, dei prodotti o delle procedure devono corrispondere

esattamente ai valori registrati nel sistema informatico. Un LLM con accesso agli elenchi di riferimento è in grado di eseguire tale abbinamento sfruttando sia la somiglianza fonetica sia la relazione semantica tra le parole, con risultati superiori a quelli ottenibili da algoritmi di distanza tra stringhe, come la distanza di Levenshtein [8].

È proprio in questo paradigma che si inscrive il sistema descritto nella presente tesi: l'assistente vocale di MyDelivery utilizza un LLM come motore di comprensione degli intenti, producendo in output strutture JSON che l'applicazione mobile può consumare direttamente, senza richiedere all'autista di adattare il proprio linguaggio a formati predefiniti.

CAPITOLO 2

L'applicazione MyDelivery

In questo capitolo viene descritto il contesto in cui si è svolto il progetto, a partire dall'azienda Infoservice e dal settore in cui opera. Si analizzerà il sistema MyDelivery nella sua interezza, partendo dall'architettura tecnica composta da backoffice web e applicazione mobile, per poi percorrere nel dettaglio ogni schermata e funzionalità offerta agli autisti consegnatari. Si esamineranno il paradigma offline-first su cui l'applicazione è costruita, il meccanismo di sincronizzazione bidirezionale dei dati e le tecnologie che lo rendono possibile. Infine si identificheranno le criticità operative che hanno portato alla decisione di integrare un assistente vocale basato su Intelligenza Artificiale e si definiranno i vincoli progettuali entro cui tale integrazione è stata progettata.

2.1 L'azienda Infoservice

Infoservice s.r.l. è un'azienda con sede a San Benedetto del Tronto, in provincia di Ascoli Piceno, che opera nel settore dello sviluppo di software gestionali e soluzioni informatiche per le imprese. L'azienda è partner del gruppo Zucchetti, la principale realtà italiana nel campo del software gestionale, e possiede le certificazioni ISO 9001 per il sistema di gestione della qualità. Il portafoglio di prodotti di Infoservice comprende diverse soluzioni rivolte alle piccole e medie imprese del settore trasporti, tra cui MyDelivery, la quale rappresenta il prodotto di punta per la gestione operativa delle consegne sul campo.

Il lavoro di tirocinio oggetto della presente tesi si è svolto all'interno del team di sviluppo di Infoservice, con l'obiettivo di progettare e realizzare un componente di Intelligenza Artificiale vocale integrabile nell'applicazione MyDelivery. Il lavoro è stato condotto in collaborazione con il team tecnico dell'azienda, che ha fornito la documentazione funzionale, l'accesso all'infrastruttura di test e il supporto necessario per comprendere a fondo le logiche operative del sistema.

2.2 Il sistema MyDelivery: panoramica generale

MyDelivery è un sistema TMS (*Transport Management System*) studiato per fornire alle aziende di trasporto un supporto completo nella gestione e nella pianificazione delle spedizioni. La piattaforma è pensata per essere accessibile da dispositivi mobili, come smartphone e tablet, attraverso un'applicazione installabile direttamente dagli autisti consegnatari.

L'idea alla base di MyDelivery nasce dall'osservazione di un problema ricorrente nelle piccole e medie imprese del settore logistico, ovvero la necessità di disporre di strumenti digitali facili da usare, veloci da implementare, scalabili nel tempo, economici e integrabili con il proprio gestionale aziendale (ERP, sistema di e-commerce, etc). Le PMI, infatti, non hanno tipicamente bisogno di sistemi TMS particolarmente complessi; esse cercano soluzioni che risolvano in modo pratico i problemi quotidiani legati alla gestione delle consegne.

Tramite l'applicazione MyDelivery, ogni autista consegnatario può svolgere le seguenti operazioni:

- visualizzare le consegne a lui assegnate, sia per singola giornata che per periodo;
- aggiornare lo stato della consegna della merce;
- effettuare la riscossione di crediti aperti da parte del cliente;
- effettuare una resa cauzioni contestualmente allo scarico della merce;
- sfruttare le funzionalità del dispositivo, come il sistema di navigazione su mappa per raggiungere il punto di scarico.

I vantaggi derivanti dall'uso di MyDelivery riguardano diversi aspetti del processo logistico; in particolare, evidenziamo:

- una migliore organizzazione della fase di consegna della merce;
- una maggiore accuratezza nella registrazione dello stato di consegna;
- tempi di rendicontazione al rientro dalla consegna più rapidi;
- l'aggiornamento in tempo reale delle consegne effettuate, visibile anche dagli altri autisti e dal personale amministrativo.

Il sistema viene fornito in modalità SaaS (*Software as a Service*), con la parte di configurazione gestita direttamente da Infoservice attraverso un portale web dedicato. I documenti di consegna vengono attualmente inviati tramite inserimento diretto nel database, sebbene nell'evoluzione futura della piattaforma sia prevista l'implementazione di modalità di interscambio tramite servizi API REST standardizzati.

2.3 L'architettura tecnica

Il sistema MyDelivery si basa su un'architettura che combina un'applicazione mobile *offline-first* con un frontend web di configurazione. Questa scelta architetturale è motivata dall'obiettivo di garantire la massima affidabilità anche in condizioni di connettività limitata o assente, una caratteristica fondamentale per gli autisti che operano sul territorio e si trovano frequentemente in zone con scarsa copertura di rete.

L'ecosistema si articola in tre componenti principali: un backoffice web per la configurazione e l'amministrazione, un'applicazione mobile per l'operatività sul campo, e un'infrastruttura di sincronizzazione dei dati che tiene allineati i due livelli.

2.3.1 Il backoffice web

Il backoffice è un'interfaccia web di configurazione accessibile dal personale amministrativo dell'azienda di trasporti. Esso è realizzato in PHP con una struttura modulare che comprende tutte le risorse necessarie al suo corretto funzionamento.

Attraverso questa interfaccia, l'amministrazione dell'azienda può personalizzare gli aspetti fondamentali dell'applicazione. L'accesso al pannello avviene tramite credenziali composte da un codice utente e un dominio aziendale (ad esempio, `MARIO@DEMO`), che il sistema di gateway utilizza per identificare il cliente di destinazione, in modo da determinare i parametri di connessione appropriati al database corretto.

Una volta effettuato l'accesso, come mostrato nella Figura 2.1, la schermata principale presenta tre sezioni nella barra laterale, ciascuna dedicata a un aspetto specifico della configurazione. Tali sezioni sono:

- *La sezione utenti*, che consente la creazione e la modifica degli account con accesso all'applicazione mobile, associando ciascun utente a un codice vettore e a un'azienda di appartenenza.
- *La sezione parametri*, che permette di personalizzare aspetti specifici del comportamento dell'app attraverso coppie nome-valore di diversi tipi (stringhe, numeri, booleani).
- *La sezione metodi di pagamento*, che consente di configurare le modalità di incasso disponibili per i consegnatari, ciascuna identificata da un codice univoco (`CONT` per contanti, `BANC` per bonifico bancario, `CARD` per carta di credito, `ASSE` per assegno), da una descrizione, da un ordine di visualizzazione e da uno stato di attivazione.

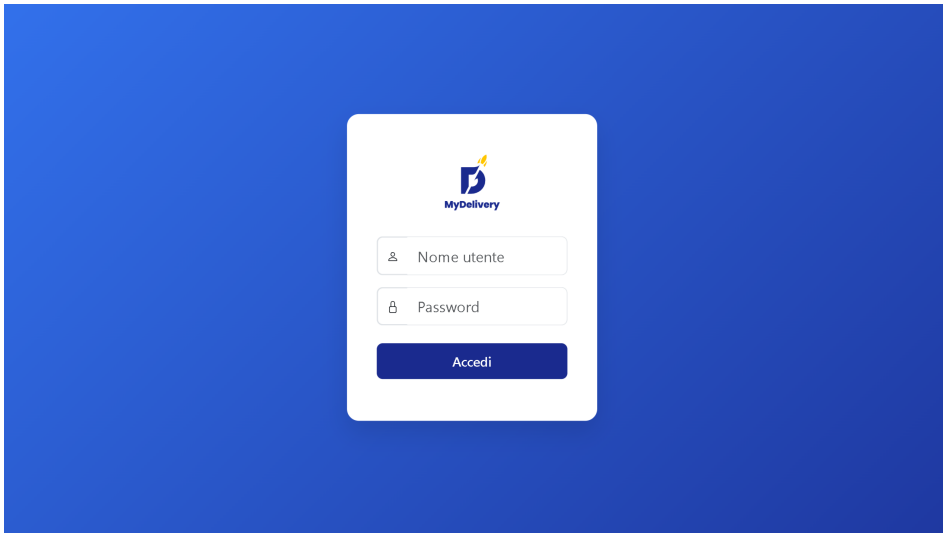


Figura 2.1: Schermata di login del pannello di configurazione web di MyDelivery

Le sezioni del pannello sono illustrate, rispettivamente nelle Figure 2.2, 2.3 e 2.4. La configurazione dei metodi di pagamento riveste un ruolo di rilievo nel contesto del presente progetto, poiché il sistema di Intelligenza Artificiale vocale deve essere in grado di "mappare" correttamente le espressioni dell'autista (come "contanti", "cash", "carta" o "assegno") ai codici interni configurati in questa sezione (`CONT`, `BANC`, `CARD`, `ASSE`).

Sul versante del portale di gestione centralizzato è possibile creare nuove aziende all'interno della piattaforma e attivare le utenze dei vari autisti e consegnatari. L'anagrafica delle aziende associa ciascuna realtà a un codice che i sistemi integrati devono utilizzare per

inviare i documenti da consegnare, mentre l'anagrafica degli utenti collega ciascun operatore all'azienda di appartenenza e al codice del vettore presente nei documenti di trasporto.

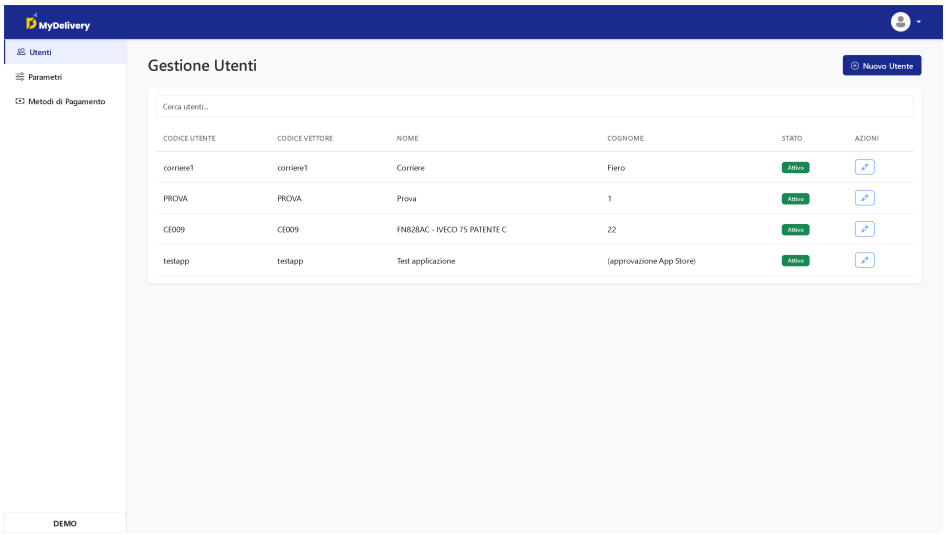


Figura 2.2: Sezione di gestione degli utenti nel pannello di configurazione

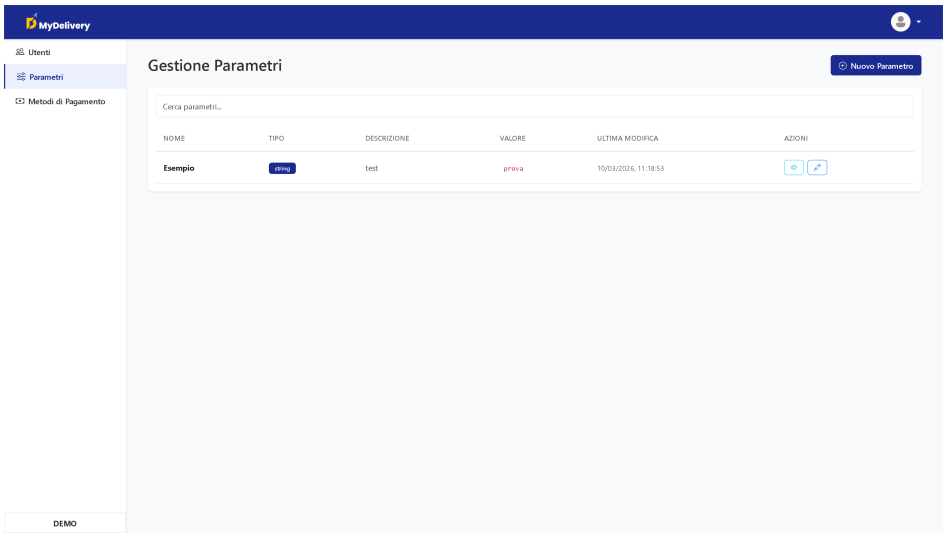


Figura 2.3: Sezione di gestione dei parametri nel pannello di configurazione

2.3.2 L'applicazione mobile

L'applicazione client è sviluppata in Flutter, un framework open source creato da Google che consente di mantenere un'unica base di codice sorgente per generare applicazioni native sia per iOS che per Android. La scelta di Flutter è motivata dalla necessità di ridurre i tempi di sviluppo e manutenzione: invece di gestire due progetti separati con linguaggi e ambienti diversi, il team di sviluppo lavora su un'unica codebase che viene compilata in modo nativo per entrambe le piattaforme.

L'architettura dell'applicazione si fonda sul paradigma *offline-first*: il dispositivo mobile è in grado di funzionare in modo completamente autonomo anche senza connessione a Internet, garantendo che l'autista possa svolgere ogni operazione senza interruzioni, indipendentemente dalle condizioni di rete. Questo aspetto è fondamentale nel contesto d'uso reale,

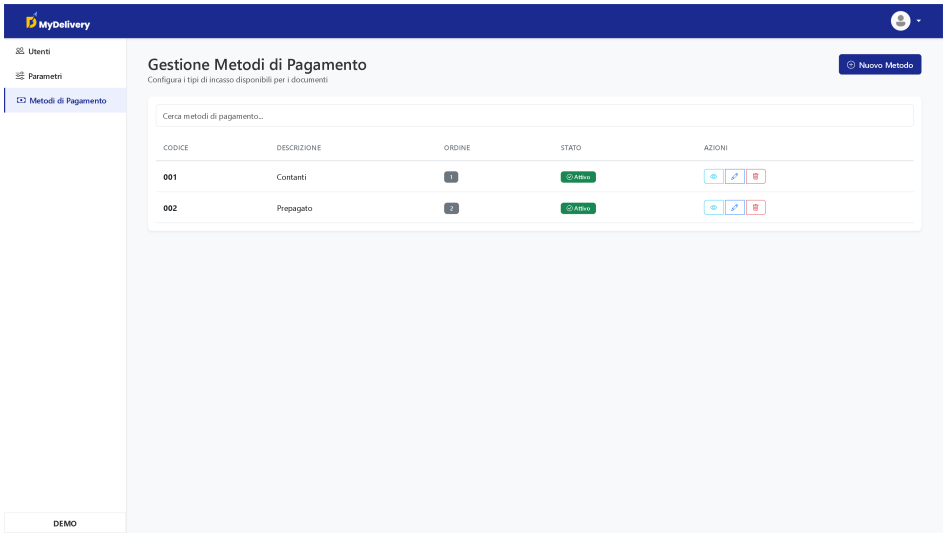


Figura 2.4: Sezione di configurazione dei metodi di pagamento nel backoffice

dove gli autisti si trovano frequentemente in zone industriali, magazzini sotterranei o aree rurali con copertura cellulare limitata o assente.

Il cuore di questa architettura è costituito dal database locale SQLite, un motore di database relazionale leggero, robusto e integrato direttamente nel dispositivo. Ogni operazione compiuta dall'autista — la registrazione di un incasso, l'aggiornamento dello stato di una consegna, il reso di una pedana — viene salvata prima nel database locale. La sincronizzazione con il server centrale avviene attraverso Electric SQL, un sistema di sincronizzazione bidirezionale che gestisce automaticamente la propagazione dei dati tra il database SQLite locale e il database centrale PostgreSQL.

Il database centrale PostgreSQL rappresenta la fonte di verità per tutti i dati aziendali. Lo schema del database aziendale è progettato in modo da essere simile a quello locale utilizzato dall'applicazione mobile, facilitando, così, la replica automatica dei dati. Il sistema prevede complessivamente due livelli di database: un database centrale per la gestione del gateway (che contiene le credenziali e la configurazione dei clienti) e un database separato per ogni azienda cliente. Quando il dispositivo rileva una connessione disponibile, Electric SQL si occupa di risolvere eventuali conflitti di sincronizzazione e di allineare i due database, mantenendo la coerenza delle informazioni. Questo meccanismo garantisce che i dati registrati offline dall'autista vengano propagati al server centrale non appena possibile, e che eventuali aggiornamenti provenienti dal backoffice raggiungano il dispositivo mobile in tempo reale.

2.3.3 Il gateway di autenticazione

Per la gestione della separazione dei database dei diversi clienti e delle licenze è stato progettato un sistema basato su gateway. Questo componente rappresenta il punto di controllo per l'accesso all'intero sistema; esso verifica la validità della licenza del cliente, gestisce l'autenticazione degli utenti e determina le credenziali di accesso al database corretto.

Il meccanismo funziona analizzando il dominio contenuto nelle richieste di login; ad esempio, un utente che effettua l'accesso come `MARIO@DEMO` viene indirizzato al database dell'azienda "DEMO", mentre un utente che accede come `LUIGI@ACME` viene instradato verso il database dell'azienda "ACME". Questo approccio garantisce che ogni azienda acceda esclusivamente ai propri dati, mantenendo un isolamento completo tra i diversi clienti della piattaforma.

2.4 Le funzionalità dell'applicazione mobile

In questa sezione vengono descritte nel dettaglio tutte le schermate e le funzionalità offerte dall'applicazione mobile MyDelivery, seguendo il flusso operativo tipico di una giornata lavorativa dell'autista, dall'accesso all'app fino alla generazione dei report di fine giornata.

2.4.1 Accesso e sincronizzazione iniziale

L'applicazione richiede le credenziali per l'accesso, configurabili dall'amministrazione attraverso la sezione utenti del backoffice descritta nella Sezione 2.3.1. Come per il pannello di configurazione, è necessario specificare il dominio aziendale insieme al nome utente.

Una volta effettuato l'accesso, l'applicazione avvia una fase di sincronizzazione con i server di MyDelivery, durante la quale vengono scaricati i dati più recenti: giri di consegna assegnati, documenti, informazioni sui clienti, listini cauzioni e configurazioni aziendali. Questa operazione potrebbe richiedere alcuni istanti al primo avvio, mentre le sincronizzazioni successive sono incrementali, e quindi più rapide. La schermata di login e quella di inizializzazione vengono mostrate nella Figura 2.5.

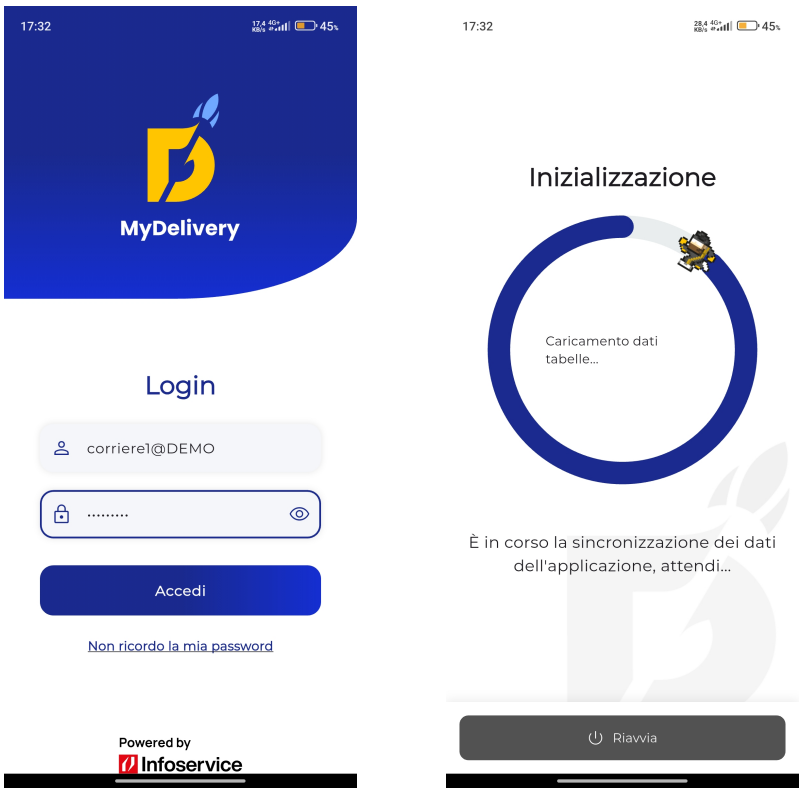


Figura 2.5: Schermate di login e di sincronizzazione iniziale dell'applicazione mobile

2.4.2 La lista dei giri di consegna

Al termine della sincronizzazione, l'interfaccia iniziale dell'applicazione presenta la lista dei giri di consegna assegnati all'autista. Il concetto di "giro di consegna" è l'unità organizzativa fondamentale di MyDelivery; esso rappresenta il raggruppamento di uno o più documenti da consegnare in una specifica giornata, tipicamente organizzati per zona geografica o per logica di percorso ottimale.

I giri vengono generati dal gestionale di riferimento dell'azienda di trasporto e inviati ai server di MyDelivery, da cui l'applicazione li scarica durante la fase di sincronizzazione. Nella schermata principale, ogni giro è identificato da un nome (ad esempio "Giro Firenze Centro" o "Giro Prato Nord"), dal codice vettore associato e da un indicatore che mostra quanti documenti sono stati consegnati rispetto al totale.

La lista dei giri può essere filtrata utilizzando appositi parametri di ricerca. I filtri disponibili consentono di visualizzare tutti i giri, solo quelli in consegna (ovvero quelli su cui l'autista sta lavorando attivamente) o solo quelli già completati. La schermata è illustrata nella Figura 2.6.

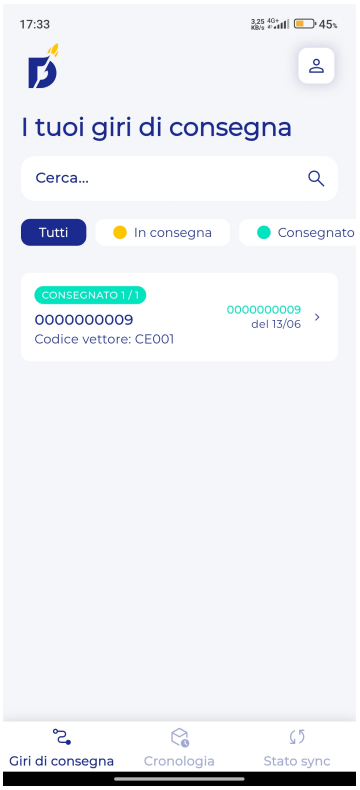


Figura 2.6: Schermata principale dell'applicazione con la lista dei giri di consegna e i filtri di stato

2.4.3 Il dettaglio del giro di consegna

Selezionando uno dei giri presenti nella lista, l'autista accede alla pagina di dettaglio che presenta l'elenco dei documenti assegnati al giro selezionato. Ogni documento corrisponde a una consegna da effettuare presso un determinato cliente e riporta le informazioni essenziali, ovvero numero del documento, data, ragione sociale del cliente, indirizzo di consegna e numero di telefono.

La visualizzazione dei documenti all'interno del giro è disponibile in due modalità distinte, ovvero:

- *la vista a elenco testuale*, che presenta i documenti in formato lista verticale con le informazioni principali di ciascuna consegna;
- *la vista su mappa*, attivabile con un apposito pulsante, che mostra la posizione geografica di tutti i punti di consegna su una mappa interattiva, consentendo all'autista di avere una visione d'insieme del percorso.

Da questa schermata è, inoltre, possibile riordinare i documenti, premendo il pulsante di riordinamento posto in cima all'elenco e trascinando i documenti nell'ordine desiderato. Questa funzionalità consente all'autista di ottimizzare il percorso in base alla propria esperienza o a condizioni contingenti (traffico, chiusure stradali, orari di apertura dei clienti). I documenti possono essere filtrati in base al loro stato: tutti, da fare o gestiti. Il dettaglio di un singolo giro viene mostrato nella Figura 2.7.



Figura 2.7: Dettaglio di un giro di consegna

2.4.4 La gestione del singolo documento di consegna

Selezionando un documento dall'elenco del giro, l'autista accede alla corrispettiva pagina di gestione, che rappresenta il fulcro operativo dell'intera applicazione. Questa schermata raccoglie tutte le informazioni relative alla consegna e offre l'accesso a tutte le azioni che l'autista può compiere, come illustrato nella Figura 2.8.

Nella parte superiore della pagina sono riportate le informazioni identificative del documento, ovvero il numero (ad esempio "BOL012/2024"), la data, la ragione sociale del cliente, l'indirizzo della sede di consegna, il CAP e il metodo di pagamento previsto. Una sezione riepilogativa mostra il dettaglio degli articoli contenuti nel documento con le relative quantità e i relativi importi, il numero di colli, il peso netto, il totale del documento, l'eventuale importo delle cauzioni e il totale complessivo. La pagina presenta, inoltre, una mappa che indica la posizione di consegna, utile all'autista per orientarsi o per avviare la navigazione verso il punto di scarico.

Al di sotto delle informazioni, la sezione "Azioni" raggruppa tutte le operazioni che l'autista può compiere sul documento:

- gestisci consegna;
- aggiungi incasso;

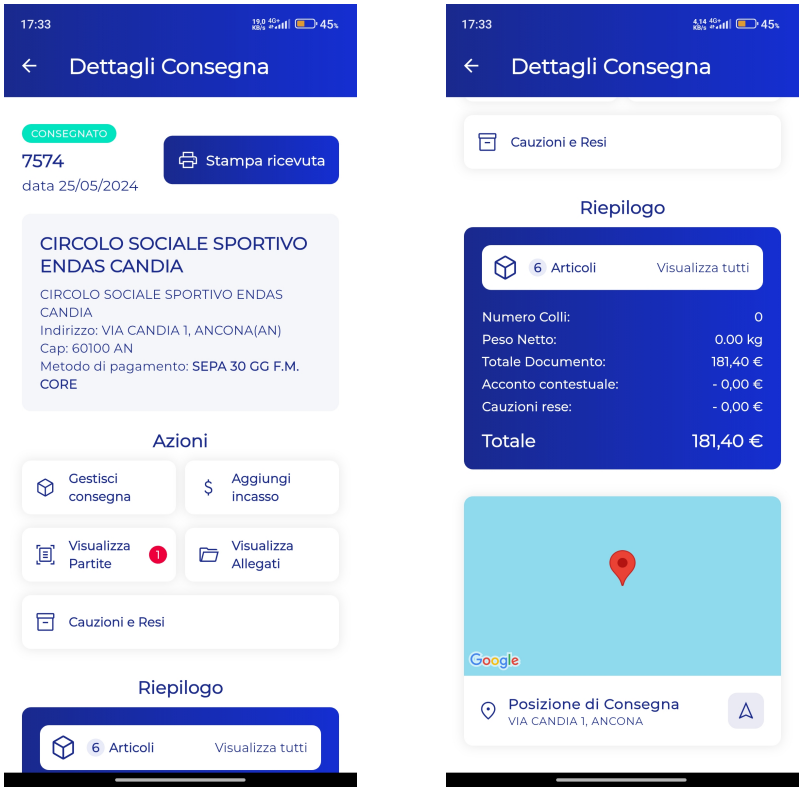


Figura 2.8: Pagina di gestione del singolo documento di consegna con le azioni disponibili (sinistra) e il riepilogo (destra)

- visualizza partite;
- visualizza allegati;
- cauzioni e resi.

Una volta completata la consegna, è, inoltre, disponibile la funzionalità di stampa della ricevuta. Le sottosezioni seguenti descrivono ciascuna di queste azioni nel dettaglio.

Gestione della consegna

All'interno della voce "Gestisci consegna" si apre un pannello che consente all'autista di registrare l'esito dell'operazione, come mostrato nella Figura 2.9. Le informazioni che possono essere inserite sono tre, ovvero:

- *l'esito della consegna*, selezionabile tra due opzioni: "Consegna effettuata" e "Non consegnato";
- il nome del referente, ovvero la persona fisica che ha ricevuto la merce presso il cliente (ad esempio, "Mario Rossi");
- le annotazioni, un campo di testo libero in cui l'autista può inserire note aggiuntive relative alla consegna (ad esempio, "lasciato sul retro" o "merce danneggiata").

Una volta compilati i campi desiderati, l'autista conferma l'operazione premendo il pulsante di conferma. Questa è una delle operazioni più frequenti nella giornata lavorativa di un consegnatario: l'autista, con le mani occupate dai colli appena consegnati, potrebbe pronunciare un comando come "segna la consegna come effettuata, ha ritirato Mario, nota: lasciato in portineria".

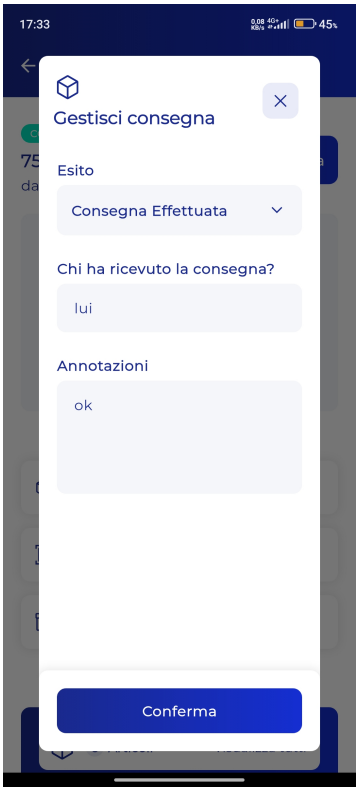


Figura 2.9: Pannello di gestione dell'esito della consegna con i campi per referente e annotazioni

Registrazione degli incassi

All'interno della voce "Aggiungi incasso" l'autista accede all'interfaccia di gestione degli incassi relativi al documento selezionato, illustrata nella Figura 2.10. Questa sezione consente di registrare il pagamento ricevuto dal cliente, con la possibilità di ripartire l'importo attraverso le diverse modalità di pagamento configurate dall'amministrazione nel backoffice.

La schermata degli incassi mostra in alto la ragione sociale del cliente, il metodo di pagamento previsto e la data dell'incasso. Subito sotto sono riportati il totale del documento e l'importo già pagato. Per ciascun metodo di pagamento attivo (ad esempio assegno, bonifico bancario, carta di credito) è presente un campo numerico in cui inserire l'importo parziale. Il pulsante "Carica" consente di assegnare automaticamente l'importo rimanente a un singolo metodo di pagamento, velocizzando l'operazione nei casi più comuni in cui il cliente paga l'intero importo con un unico metodo.

Questa funzionalità presenta una complessità strutturale di rilievo per il sistema vocale, poiché l'autista potrebbe dover comunicare incassi suddivisi su più metodi in un'unica frase, come, ad esempio, "aggiungi un incasso di 150 euro in contanti e 50 euro con assegno". Il sistema deve essere in grado di decomporre questa istruzione nei suoi componenti elementari e di "mapparla" correttamente sui codici interni dei metodi di pagamento.

Visualizzazione e incasso delle partite aperte

All'interno della voce "Visualizza partite" l'autista può consultare tutte le partite aperte collegate al cliente della consegna selezionata. Le partite aperte rappresentano crediti pregressi che il cliente deve ancora saldare e che non sono legati alla consegna in corso: si tratta tipicamente di fatture scadute, note di debito o documenti contabili non ancora regolati.



Figura 2.10: Interfaccia di gestione degli incassi con la ripartizione per metodo di pagamento

Per ciascuna partita viene mostrato il riferimento del documento, il numero della partita, la data di scadenza, l'importo totale e l'importo già pagato. Premendo il pulsante "Incassa" o "Modifica" è possibile registrare un pagamento associato a una specifica partita, con la possibilità di suddividere l'incasso tra le diverse modalità di pagamento, analogamente a quanto avviene per gli incassi sui documenti correnti. La schermata è mostrata nella Figura 2.11.

Questa funzionalità estende il ruolo dell'autista oltre la semplice consegna della merce, trasformandolo in un punto di contatto per la gestione dei crediti aziendali. Dal punto di vista del sistema vocale, un autista potrebbe pronunciare un comando come "incassa 112 euro con carta di credito per la partita aperta di Alimentari Freschi".

Gestione delle cauzioni e dei resi

All'interno della voce "Cauzioni e resi" l'autista accede alla gestione dei vuoti e degli imballi a rendere, mostrata nella Figura 2.12. Questa sezione consente di visualizzare il saldo vuoti presso il cliente e di inserire eventuali resi cauzione.

Per ciascun articolo cauzionale (ad esempio, fusti di birra da 30 litri, fusti di Coca-Cola da 18 litri, bancali, bombole, frigoriferi) la schermata mostra tre colonne: la giacenza attuale presso il cliente, la quantità consegnata con il documento corrente e il numero di pezzi resi. L'autista può incrementare o decrementare la quantità resa utilizzando i pulsanti "+" e "-" accanto a ciascun articolo.

Se il cliente ha attiva la gestione a valore delle cauzioni, in base al listino di appartenenza il sistema provvede automaticamente a stornare l'importo corrispondente dal documento in consegna. MyDelivery tiene traccia automaticamente delle giacenze, mostrando sempre un saldo aggiornato dei vuoti presso ciascun cliente. All'inserimento di un reso cauzioni, se

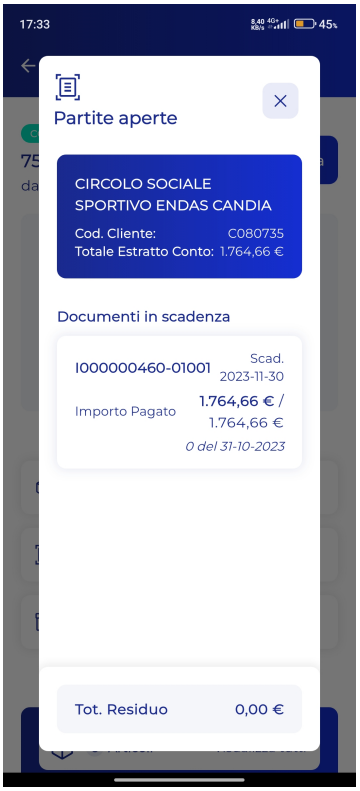


Figura 2.11: Schermata di visualizzazione delle partite aperte del cliente con possibilità di incasso

l'autista è supportato da una stampante mobile, può effettuare la stampa di una ricevuta che attesta il reso delle cauzioni ricevute.

Dettaglio degli articoli e degli allegati

All'interno della pagina di gestione del documento, l'autista ha accesso a due ulteriori sezioni informative. La prima è la visualizzazione degli articoli contenuti nel documento di consegna, mostrata nella Figura 2.13; per ciascun articolo vengono mostrati il codice, la descrizione, la quantità (espressa nell'unità di misura appropriata, come chilogrammi o litri), il prezzo unitario e il totale. La seconda è la visualizzazione degli allegati, ovvero dei documenti caricati da fonti esterne come il gestionale aziendale, che possono includere bolle di accompagnamento, certificati di conformità o altri documenti correlati alla spedizione.

2.4.5 La stampa della ricevuta di consegna

Una volta terminata la consegna e completate tutte le operazioni associate (esito, incassi, resi), l'operatore può stampare la ricevuta generata da MyDelivery, il cui esempio è illustrato nella Figura 2.14. Tale ricevuta funge da riepilogo complessivo dell'operazione e contiene:

- i dati identificativi del documento (numero, data, totale);
- il dettaglio degli incassi registrati, suddivisi per metodo di pagamento, con i relativi importi;
- il dettaglio delle cauzioni, con giacenza, quantità consegnata e quantità resa per ciascun articolo;
- lo spazio per le firme del corriere e del destinatario.

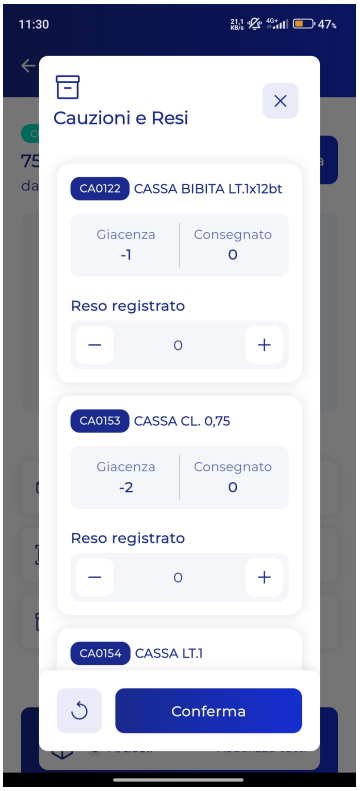


Figura 2.12: Schermata di gestione delle cauzioni e dei resi vuoti con il saldo per articolo

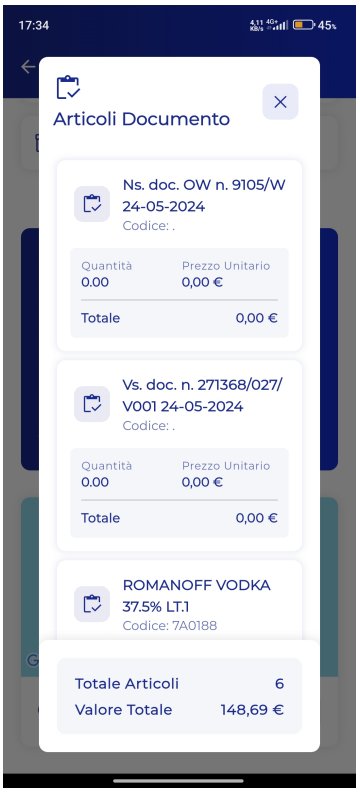


Figura 2.13: Dettaglio degli articoli contenuti in un documento di consegna

La stampa può avvenire tramite stampante mobile collegata al dispositivo via Bluetooth,

oppure la ricevuta può essere condivisa in formato digitale. Oltre alla ricevuta generata da MyDelivery, è possibile stampare anche gli allegati caricati da fonti esterne.



Figura 2.14: Esempio di ricevuta di consegna generata da MyDelivery con il riepilogo degli incassi e delle cauzioni

2.4.6 La cronologia e i report di fine giornata

Una volta effettuate tutte le consegne della giornata, l'autista o il consegnatario ha la possibilità di visualizzare e condividere un riepilogo di quanto fatto durante il periodo lavorativo. Questa funzionalità è accessibile dalla sezione "Cronologia", raggiungibile attraverso la barra di navigazione laterale dell'applicazione, accessibile tramite il menù ad hamburger in alto a sinistra.

All'interno della sezione "Cronologia", mostrata nella Figura 2.15, l'autista può selezionare un periodo di riferimento specificando una data di inizio e una data di fine, oppure utilizzare la scorciatoia "Oggi" per visualizzare i dati della giornata corrente. Una volta impostato il periodo, sono disponibili due tipi di report generabili, ovvero:

- *il riepilogo delle consegne e dei pagamenti*, che elenca tutte le consegne effettuate nel periodo selezionato con il relativo stato, l'importo del documento e l'importo incassato, oltre agli eventuali incassi derivanti da partite aperte;
- *il riepilogo degli incassi*, che riporta gli importi incassati suddivisi per cliente e per metodo di pagamento, fornendo un totale complessivo utile per la rendicontazione di cassa a fine giornata.

Entrambi i report possono essere stampati tramite stampante mobile o condivisi in formato digitale.



Figura 2.15: Schermata della cronologia con la selezione del periodo e le opzioni di generazione dei report

2.4.7 Le impostazioni e la firma dell'autista

Nella sezione relativa alle impostazioni, accessibile dalla barra laterale, l'autista può configurare la propria firma digitale. Questa firma viene utilizzata per la conferma delle consegne e viene apposta automaticamente sulle ricevute generate dall'applicazione. L'acquisizione della firma avviene tramite il touch screen del dispositivo, con la possibilità di cancellarla e riacquisirla in qualsiasi momento.

2.5 Il problema operativo e la necessità dell'assistente vocale

Tutte le funzionalità descritte nelle sezioni precedenti richiedono, nella loro implementazione attuale, un'interazione manuale con l'interfaccia touch dello smartphone o del tablet. L'autista deve toccare pulsanti, scorrere elenchi, digitare importi numerici, selezionare voci da menù a tendina e compilare campi di testo. Il documento di analisi funzionale di MyDelivery, nella sezione dedicata alle osservazioni, riporta esplicitamente questa necessità con una frase sintetica: "avere la possibilità di poter gestire, tramite una mano, tutta l'app".

Questa annotazione riflette un problema strutturale che pervade l'intera esperienza d'uso dell'applicazione nel suo contesto operativo reale. Le condizioni in cui un autista si trova durante la giornata lavorativa rendono spesso difficile o impossibile interagire con lo schermo: le mani occupate dai colli, la guida del veicolo, il freddo, i guanti da lavoro. La conseguenza diretta è che alcune operazioni vengono rimandate a momenti successivi, dimenticate o eseguite in modo approssimativo. Un incasso registrato con il metodo di pagamento sbagliato, una nota di consegna omessa, un reso cauzionale non segnalato sono tutte imprecisioni che si accumulano nel corso della giornata e generano disallineamenti

contabili e operativi che il personale amministrativo deve, poi, risolvere manualmente al rientro dell'autista.

L'esigenza emersa dall'azienda è, quindi, quella di consentire agli autisti di governare l'intera applicazione utilizzando la voce, in modo da poter operare con una sola mano o in modalità completamente *hands-free*. Il sistema di Intelligenza Artificiale deve fungere da interprete universale, capace di recepire il linguaggio naturale dell'autista — spesso frammentario, con inflessioni dialettali, pronuncia imprecisa dei nomi dei clienti e formulazioni colloquiali — e di tradurlo in comandi operativi diretti per l'applicazione, senza richiedere all'utente di adattare il proprio modo di parlare a un formato rigido o a parole chiave predefinite. L'analisi dettagliata delle condizioni d'uso e dei requisiti che ne derivano è svolta nel Capitolo 3.

CAPITOLO 3

Analisi dei requisiti

In questo capitolo vengono identificati e descritti i requisiti che il sistema di assistente vocale deve soddisfare per essere integrato nell'ecosistema MyDelivery. Si parte dall'analisi del contesto operativo reale, che definisce il profilo degli utenti e le condizioni d'uso attese, per poi procedere con la specifica dei requisiti funzionali, dei requisiti non funzionali e dei vincoli progettuali imposti dall'azienda. Si conclude con la definizione degli scenari operativi che il sistema deve coprire, organizzati per categoria funzionale.

3.1 Il contesto operativo e il profilo degli utenti

Prima di definire i requisiti del sistema, è necessario comprendere con precisione il contesto in cui esso dovrà operare e il profilo delle persone che lo utilizzeranno. Gli utenti principali di MyDelivery sono autisti e consegnatari che svolgono la loro attività sul campo: operatori pratici, abituati all'applicazione e alle sue funzionalità, ma che si trovano frequentemente in situazioni in cui l'interazione manuale con il dispositivo è scomoda, rischiosa o del tutto impraticabile.

Il profilo tipico dell'utente finale del sistema vocale esclude per sua natura chi ha tempo e libertà di movimento per consultare menù, scorrere elenchi e compilare campi di testo con la tastiera virtuale. Quel tipo di interazione rimane possibile e invariata; il sistema vocale serve a coprire tutte le situazioni in cui l'interfaccia touch risulta impraticabile, offrendo un'alternativa che si inserisce nel flusso di lavoro senza richiedere all'operatore di fermarsi o di modificare le proprie abitudini operative.

Le condizioni d'uso che il sistema dovrà gestire sono state ricavate dall'osservazione diretta del contesto operativo durante il tirocinio presso Infoservice e dalla lettura del documento funzionale di MyDelivery. Le situazioni più ricorrenti sono le seguenti:

- l'operatore trasporta colli e non può liberare una o entrambe le mani per interagire con il dispositivo;
- l'operatore si trova alla guida del veicolo e deve registrare un dato senza fermarsi, oppure vuole pianificare la prossima operazione prima di scendere dal mezzo;
- l'operatore lavora in condizioni di freddo, pioggia o indossa guanti da lavoro che rendono il touch screen poco reattivo o inutilizzabile;

- l'operatore deve inserire dati complessi, come importi numerici con decimali o nomi di referenti, che richiederebbero l'apertura della tastiera virtuale e una digitazione precisa.

In tutti questi scenari, la soluzione vocale deve essere in grado di recepire un comando espresso in linguaggio naturale italiano, potenzialmente frammentario o con inflessioni dialettali, e di tradurlo in un'azione operativa strutturata senza richiedere all'utente di adattare il proprio modo di parlare a formati rigidi o parole chiave predefinite. L'interazione deve risultare il più possibile simile a quella che l'operatore avrebbe con un collega a cui chiede di svolgere un'operazione al posto suo.

Vale la pena sottolineare che le condizioni elencate non sono casi limite o eccezioni; esse infatti rappresentano la normalità della giornata lavorativa di un consegnatario. Progettare per questi scenari non significa, quindi, aggiungere un livello di robustezza opzionale, ma definire il caso d'uso principale attorno a cui l'intero sistema deve essere costruito.

3.2 I requisiti funzionali

I requisiti funzionali descrivono le capacità operative che il sistema deve possedere per soddisfare le esigenze degli utenti. Essi sono stati derivati dall'analisi del documento funzionale di MyDelivery e dalle osservazioni raccolte durante il tirocinio presso Infoservice, e sono organizzati per area funzionale in modo da rispecchiare la struttura dell'applicazione.

3.2.1 Ricezione e interpretazione dei comandi vocali

Il sistema deve essere in grado di ricevere una trascrizione testuale di un comando vocale pronunciato dall'utente e di identificare correttamente l'intento operativo espresso, anche quando la formulazione è ellittica, colloquiale o grammaticalmente incompleta. La trascrizione viene prodotta dal motore di riconoscimento vocale nativo del dispositivo mobile (Android o iOS) e consegnata al sistema sotto forma di stringa di testo.

Il sistema deve gestire correttamente almeno i seguenti scenari di formulazione:

- comandi espressi in forma completa e grammaticalmente corretta ("aggiungi un incasso di 150 euro in contanti per Alimentari Freschi");
- comandi espressi in forma telegrafica senza struttura grammaticale completa ("segna consegna alimentari effettuata");
- comandi che contengono nomi pronunciati in modo approssimativo o parziale ("aprimi il giro di Firenze" al posto di "Firenze Centro");
- comandi che contengono sinonimi o varianti colloquiali dei termini tecnici ("cash" al posto di "contanti", "carta" al posto di "carta di credito").

3.2.2 Correzione degli errori di pronuncia tramite fuzzy matching

Il sistema deve confrontare ogni nome pronunciato dall'utente con le liste di riferimento fornite dall'applicazione per la sessione corrente, selezionando la corrispondenza più plausibile dal punto di vista semantico e fonetico. Questo meccanismo deve operare su tutte le categorie di entità nominate che compaiono nei comandi vocali, ovvero:

- *nomi dei clienti*, che devono essere ricondotti alla ragione sociale esatta presente nei documenti del giro corrente;

- *nomi dei giri di consegna*, che devono corrispondere ai nomi configurati nel sistema;
- *descrizioni degli articoli cauzionali*, che devono corrispondere alle descrizioni presenti nel listino cauzioni;
- *designazioni dei metodi di pagamento*, che devono essere normalizzate ai codici interni configurati nel backoffice (`CONT`, `BANC`, `CARD`, `ASSE`).

Quando il nome pronunciato non presenta una corrispondenza sufficientemente vicina a nessun elemento delle liste disponibili, il sistema non deve inventare una corrispondenza: deve invece produrre una risposta di errore che segnala il problema all'utente e lo invita a ripetere il comando in modo più specifico.

3.2.3 Produzione di output strutturati

Il sistema deve produrre, per ogni comando ricevuto, un oggetto JSON strutturato e conforme al contratto di comunicazione definito con l'applicazione Flutter. L'oggetto JSON deve contenere sempre e soltanto due nodi principali: un campo `risposta_vocale`, con la frase da leggere all'utente tramite sintesi vocale, e un campo `dati_operazione`, con l'azione da eseguire e i relativi parametri.

Il sistema non deve mai produrre testo libero al di fuori di questa struttura: l'output deve essere sempre e soltanto un file JSON valido, "parsabile" senza pre-elaborazione da parte del dispatcher Flutter. In caso di errore, la struttura JSON deve essere comunque prodotta e conforme al formato atteso, con l'azione `errore_comprensione` e un messaggio vocale che descrive il problema in modo comprensibile per l'utente.

3.2.4 Copertura delle funzionalità dell'applicazione

Il sistema deve coprire la totalità delle funzionalità operative di MyDelivery che si prestano al controllo vocale. Dall'analisi delle schermate e dei flussi operativi descritti nel Capitolo 2 sono stati identificati quattordici scenari operativi distinti, raggruppati nelle cinque categorie funzionali presentate nella Tabella 3.1.

3.2.5 Supporto al pagamento multiplo

Il sistema deve essere in grado di gestire comandi di incasso in cui l'importo viene ripartito su più metodi di pagamento in un'unica frase. L'utente deve poter pronunciare, ad esempio, "aggiungi un incasso di 150 euro in contanti e 50 euro con assegno" e il sistema deve produrre un array di oggetti pagamento, ciascuno con il proprio codice del metodo e il proprio importo, senza richiedere comandi separati per ciascun metodo.

3.2.6 Conversione di espressioni numeriche e temporali

Il sistema deve essere in grado di convertire numeri espressi in lettere nel corrispondente valore numerico (ad esempio, "tre pedane" deve produrre la quantità `3`). Analogamente, per i comandi relativi alla generazione dei report di cronologia, il sistema deve convertire espressioni temporali in linguaggio naturale (ad esempio, "dal 12 al 13 agosto di quest'anno") in date nel formato `YYYY-MM-DD` atteso dal dispatcher.

Categoria	Scenari coperti
Navigazione e visualizzazione	Filtraggio dei giri per stato; apertura del dettaglio di un giro specifico; cambio della modalità di visualizzazione dei documenti (lista o mappa); riordinamento dei documenti all'interno di un giro.
Gestione operativa	Aggiornamento dell'esito di una consegna con referente e annotazioni; visualizzazione delle sezioni del documento (articoli, allegati, partite aperte).
Gestione amministrativa	Registrazione di incassi sul documento corrente con supporto al pagamento multiplo; incasso di crediti pregressi (partite aperte); registrazione dei resi di vuoti e imballi.
Reportistica e impostazioni	Stampa della ricevuta di consegna; impostazione delle date nella sezione cronologia; generazione dei report di fine giornata; apertura della schermata per la firma digitale dell'autista.
Gestione degli errori	Risposta strutturata in caso di comando non compreso, dati mancanti o corrispondenza assente nelle liste di riferimento.

Tabella 3.1: Scenari operativi identificati per il sistema di assistente vocale, raggruppati per categoria funzionale.

3.3 I requisiti non funzionali

I requisiti non funzionali definiscono le caratteristiche qualitative che il sistema deve possedere indipendentemente dalle specifiche funzionalità implementate. Essi condizionano in modo determinante le scelte architetturali e tecnologiche descritte nel Capitolo 4.

3.3.1 Latenza di risposta

In un contesto operativo in cui l'utente si aspetta una risposta quasi immediata dopo aver pronunciato un comando, il tempo totale trascorso dalla ricezione della trascrizione alla restituzione del file JSON strutturato deve essere compatibile con un'interazione fluida. Il limite di riferimento accettabile è stato fissato a cinque secondi complessivi, a cui si somma la latenza di rete tra il dispositivo mobile e il server middleware. Nella pratica, l'obiettivo progettuale è mantenere il tempo di elaborazione del middleware al di sotto dei tre secondi per la grande maggioranza dei comandi.

3.3.2 Affidabilità dell'output

Il sistema deve produrre un output JSON strutturalmente valido per ogni richiesta ricevuta, senza eccezioni. In nessun caso il dispatcher Flutter deve ricevere una risposta non "parsabile" o strutturalmente non conforme al contratto definito. Questa garanzia deve valere anche in presenza di errori interni, timeout o risposte anomale da parte del modello linguistico: in tutti questi casi il sistema deve degradare in modo controllato, restituendo comunque un file JSON conforme con l'azione `errore_comprensione`.

3.3.3 Sicurezza delle credenziali

Le chiavi API del servizio di Intelligenza Artificiale non devono mai essere esposte al dispositivo mobile dell'utente, né transitare nelle risposte HTTP, né essere incluse nel codice sorgente dell'applicazione Flutter in nessuna forma, nemmeno offuscata. La loro gestione

deve avvenire esclusivamente nell'ambiente server, tramite variabili d'ambiente caricate da file non versionati nel sistema di controllo del codice sorgente.

3.3.4 Scalabilità economica

Il costo per singola richiesta al modello linguistico, misurato in token consumati, deve essere contenuto e compatibile con un modello di business sostenibile in produzione. Ipotizzando flotte di decine di autisti che effettuano ciascuno decine di comandi vocali al giorno, il costo complessivo delle chiamate API deve rimanere nell'ordine di grandezza di pochi euro al giorno. Questo requisito orienta la scelta verso modelli ottimizzati per bassa latenza e basso costo per token, escludendo modelli ad alta potenza computazionale che, pur offrendo capacità di ragionamento superiori, avrebbero un impatto economico incompatibile con l'uso massiccio previsto.

3.3.5 Indipendenza dall'infrastruttura esistente

Il sistema deve funzionare senza modificare in alcun modo il codice sorgente dell'applicazione Flutter esistente, senza accedere direttamente ai database aziendali e senza richiedere modifiche all'infrastruttura server già in uso da Infoservice. Il componente vocale deve presentarsi all'applicazione come un servizio esterno, accessibile tramite una chiamata HTTP standard, e il pacchetto Flutter che incapsula la logica di acquisizione vocale deve essere integrabile come dipendenza senza alterare il codice preesistente.

3.4 I vincoli progettuali

I vincoli progettuali sono stati definiti in accordo con il team di sviluppo di Infoservice e derivano da considerazioni di sicurezza, stabilità e manutenibilità del sistema esistente. A differenza dei requisiti funzionali e non funzionali, che descrivono ciò che il sistema deve fare, i vincoli descrivono ciò che il sistema non può fare o non può essere; essi delimitano lo spazio delle soluzioni architetturali ammissibili e hanno determinato in modo diretto la struttura del componente realizzato.

Il primo vincolo riguarda l'accesso al codice sorgente. L'integrazione dell'Intelligenza Artificiale non prevede la modifica diretta del codice Flutter dell'applicazione mobile, che resta sotto la responsabilità esclusiva del team interno di Infoservice. Il contributo esterno si limita alla progettazione e all'implementazione di un componente indipendente — il server middleware e il prompt di sistema — che comunica con l'app attraverso un'interfaccia di scambio dati ben definita, basata su chiamate HTTP e risposte JSON. Il team interno si occuperà di integrare il pacchetto Flutter nell'applicazione e di sviluppare il dispatcher che traduce i file JSON ricevuti in azioni native sul database SQLite locale. Questo vincolo ha una ricaduta positiva immediata: qualsiasi modifica al componente vocale — aggiornamento del modello, revisione del prompt, aggiunta di nuovi scenari — può avvenire senza toccare l'applicazione esistente e senza rischiare di introdurre regressioni nel codice di produzione.

Il secondo vincolo riguarda l'accesso ai dati. Il modello di Intelligenza Artificiale non deve avere accesso diretto né al database SQLite locale del dispositivo né al database centrale PostgreSQL. Tutte le informazioni di contesto necessarie per la corretta interpretazione dei comandi — i nomi dei clienti del giorno, i giri assegnati, i metodi di pagamento configurati nel backoffice, gli articoli cauzionali validi — vengono fornite dall'applicazione al momento in cui viene effettuata ciascuna richiesta, sotto forma di liste di riferimento allegate alla trascrizione vocale. Il modello lavora, quindi, sempre su un contesto circoscritto, aggiornato e relativo alla singola sessione, senza possibilità di accumulare o memorizzare informazioni tra una

richiesta e l'altra. Questo approccio non costituisce di per sé il meccanismo che isola i dati tra aziende diverse — tale isolamento è garantito a monte dal gateway di autenticazione descritto nella Sezione 2.3.3, che instrada ogni richiesta verso il database della sola azienda corretta sulla base del dominio specificato nelle credenziali — ma ne costituisce un rafforzamento complementare: poiché il modello riceve, per ciascuna richiesta, soltanto il sottoinsieme di dati che l'applicazione Flutter ha già estratto dal database dell'azienda corretta, e non conserva alcuna informazione tra una richiesta e l'altra, esso non rappresenta un punto in cui dati di aziende o di giornate lavorative diverse potrebbero accidentalmente mescolarsi, e ciò avviene senza richiedere alcuna forma di addestramento o personalizzazione specifica per ciascun cliente.

Il terzo vincolo riguarda la custodia delle credenziali. Le chiavi API del servizio di Intelligenza Artificiale non devono mai transitare sul dispositivo mobile dell'autista, né essere incluse nel codice sorgente dell'applicazione in nessuna forma, nemmeno offuscata. La loro gestione deve avvenire esclusivamente nell'ambiente server, tramite variabili d'ambiente caricate da file non versionati nel sistema di controllo del codice sorgente. Questo vincolo, oltre a rispondere a ovvie esigenze di sicurezza, impedisce che una compromissione del dispositivo mobile — dovuto a furto, smarrimento o accesso non autorizzato — possa esporre le credenziali dell'infrastruttura di Intelligenza Artificiale.

Presi insieme, questi tre vincoli convergono verso un'architettura a tre livelli completamente disaccoppiata (*loose coupling*), in cui l'applicazione mobile, il server middleware e il motore di Intelligenza Artificiale operano come entità indipendenti che comunicano attraverso interfacce ben definite. La progettazione concreta di questa architettura, con le scelte tecnologiche che ne derivano, è descritta nel Capitolo 4.

3.5 Il contratto di comunicazione tra i livelli

Un aspetto fondamentale dell'analisi dei requisiti riguarda la definizione del contratto di comunicazione tra l'applicazione Flutter e il server middleware. Questo contratto deve essere sufficientemente espressivo da coprire tutti i quattordici scenari operativi identificati, e, al tempo stesso, sufficientemente semplice da essere implementato e mantenuto senza ambiguità.

L'applicazione Flutter invia al middleware una richiesta HTTP POST contenente due campi, ovvero la trascrizione testuale del comando vocale dell'utente e le liste di riferimento della sessione corrente. Le liste di riferimento sono strutture JSON che contengono i nomi dei clienti del giorno, i nomi dei giri assegnati all'autista, i codici dei metodi di pagamento configurati nel backoffice e le descrizioni degli articoli cauzionali validi per la sessione. Nella Figura 3.1 viene illustrata la struttura di una richiesta completa.

Il middleware risponde con un oggetto JSON strutturato contenente la risposta vocale da leggere all'utente e il payload operativo con l'azione da eseguire e i relativi parametri. Questo formato è fisso e invariante: il dispatcher Flutter deve poter consumare la risposta senza dover gestire formati alternativi in funzione dell'esito della richiesta. La definizione dettagliata di ciascuna azione, con i relativi parametri e gli oggetti JSON di esempio, è presentata nel Capitolo 4.

3.6 Gestione del contesto dinamico

Un requisito di sistema che merita una trattazione separata riguarda la gestione del contesto operativo. Le liste di riferimento che l'applicazione allega a ogni richiesta non sono statiche ma cambiano in funzione della giornata lavorativa, dell'autista connesso e



Figura 3.1: Struttura di una richiesta HTTP POST inviata dall'applicazione Flutter al middleware, con la trascrizione vocale e le liste di riferimento della sessione corrente.

dell'azienda cliente. Un autista che lavora il lunedì a Firenze avrà una lista clienti diversa da quella del martedì a Prato, e un autista di una seconda azienda avrà metodi di pagamento e articoli cauzionali completamente diversi.

Questo significa che il sistema non deve richiedere alcuna forma di addestramento o personalizzazione specifica per ciascuna azienda cliente: è sufficiente che l'applicazione Flutter estragga le informazioni corrette dal proprio database SQLite locale al momento della richiesta e le alleghi alla trascrizione. Il modello di Intelligenza Artificiale riceve ogni volta un contesto fresco e completo, e opera su di esso senza dipendere da alcuna conoscenza pregressa dell'ambiente specifico.

Questo requisito ha implicazioni dirette sulla progettazione del prompt di sistema, che deve contenere istruzioni esplicite su come il modello deve utilizzare le liste di riferimento ricevute per correggere i nomi pronunciati dall'utente e normalizzare i codici dei metodi di pagamento. La progettazione del prompt è descritta nel dettaglio nella Sezione 4.3.

CAPITOLO 4

Progettazione

In questo capitolo viene presentata nel dettaglio l'architettura progettata per integrare l'assistente vocale nell'ecosistema MyDelivery. Si parte dal modello a tre livelli adottato per garantire il disaccoppiamento totale tra l'applicazione mobile, il server middleware e il motore di Intelligenza Artificiale. Si illustrano le ragioni che hanno portato alla scelta del modello linguistico Claude Haiku 4.5 di Anthropic, e si descrive il processo di progettazione del prompt di sistema, con particolare attenzione alla tecnica del fuzzy matching dinamico per la correzione degli errori di pronuncia. Si presenta, infine, la mappatura completa delle quattordici azioni operative tradotte in strutture JSON, che costituisce il contratto di comunicazione tra l'Intelligenza Artificiale e l'applicazione mobile.

4.1 Il modello a tre livelli

I vincoli progettuali descritti nel Capitolo 3 — assenza di accesso al codice sorgente dell'applicazione, assenza di accesso diretto ai database e necessità di custodire le chiavi API esclusivamente lato server — hanno condotto in modo naturale verso un'architettura a tre livelli completamente disaccoppiata. Ciascun livello opera come un'entità autonoma che comunica con gli altri attraverso interfacce ben definite, senza conoscere i dettagli implementativi interni degli altri componenti.

Come illustrato nella Figura 4.1, il flusso di una richiesta attraversa i tre livelli in sequenza: l'applicazione Flutter raccoglie la voce dell'utente e invia la trascrizione al middleware, il middleware assembla il prompt e interroga il modello linguistico, il modello produce il JSON strutturato che torna all'applicazione attraverso lo stesso percorso.

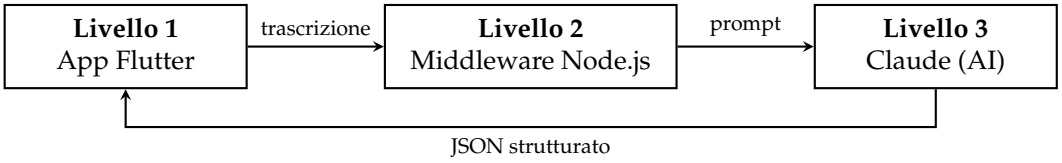


Figura 4.1: Schema dell'architettura a tre livelli: l'applicazione Flutter invia la trascrizione al middleware Node.js, che la inoltra alle API di Claude e restituisce il JSON strutturato.

4.1.1 Livello 1: l'applicazione mobile (Flutter)

Il primo livello è rappresentato dall'applicazione Flutter già esistente, sulla quale il team di sviluppo di Infoservice integrerà due componenti aggiuntivi. Il primo è il pacchetto vocale `mydelivery_voice_assistant`, sviluppato nell'ambito di questo tirocinio, che contiene il pulsante del microfono, la logica di acquisizione audio tramite il motore di *speech-to-text* nativo del dispositivo e la logica di comunicazione HTTP con il middleware. Il secondo è il dispatcher interno, ovvero una funzione che il team di Infoservice svilupperà per ricevere i JSON strutturati restituiti dal server e tradurli in azioni concrete sull'applicazione, come l'aggiornamento del database SQLite locale, il cambio di schermata attiva o la compilazione automatica di un campo.

Quando l'autista preme il pulsante del microfono e pronuncia un comando, il pacchetto vocale trascrive l'audio in testo tramite il motore nativo del dispositivo, assembla la richiesta HTTP allegando le liste di riferimento estratte dal database SQLite locale e la invia al middleware. Al ritorno della risposta, il campo `risposta_vocale` viene letto ad alta voce tramite il motore di *text-to-speech* nativo, mentre il campo `dati_operazione` viene consegnato al dispatcher per l'esecuzione.

4.1.2 Livello 2: il server middleware (Node.js)

Il secondo livello è il cuore infrastrutturale del sistema ed è il componente progettato e implementato nell'ambito di questo tirocinio. Si tratta di un server Node.js basato sul framework Express, ospitato su una macchina Linux Ubuntu dedicata fornita dall'azienda. Il suo ruolo è triplice:

- custodire in totale sicurezza la chiave API di Anthropic, che non transita mai sul dispositivo mobile;
- assemblare il prompt di sistema completo, inserendo le liste di riferimento e la trascrizione vocale nei segnaposto del template;
- inoltrare la richiesta alle API di Claude e restituire la risposta JSON strutturata all'applicazione.

Il middleware opera come un intermediario trasparente: non possiede logica di business propria, non prende decisioni operative e non modifica i dati. Il suo unico compito è fare da ponte sicuro tra il dispositivo mobile e il modello linguistico, garantendo che la chiave API resti confinata nell'ambiente server e che ogni richiesta venga formattata correttamente prima di essere inviata a Claude.

4.1.3 Livello 3: il motore di Intelligenza Artificiale (Claude)

Il terzo livello è il componente intelligente del sistema, rappresentato dal modello linguistico Claude di Anthropic. Tale componente riceve dal middleware il prompt completo — contenente le istruzioni di sistema, le liste di riferimento e la trascrizione vocale dell'utente — e restituisce un file JSON strutturato con l'azione da eseguire e la risposta vocale da leggere all'autista.

Il modello non ha accesso diretto a nessun dato aziendale: conosce i clienti, i giri e gli articoli solo perché l'applicazione li fornisce ad esso esplicitamente in ogni singola richiesta. Questo approccio garantisce che il modello lavori sempre su un contesto fresco e circoscritto, senza possibilità di accumulare o memorizzare informazioni tra una sessione e l'altra.

4.1.4 Il flusso operativo end-to-end

Per rendere concreta l'architettura appena descritta, si consideri il seguente esempio di flusso operativo completo. L'autista preme il pulsante del microfono nell'applicazione Flutter e pronuncia: "incassa 150 euro in contanti per il supermercato centrale". L'applicazione trascrive l'audio in testo tramite il motore di *speech-to-text* nativo e costruisce una richiesta HTTP POST verso il middleware, allegando la trascrizione e le liste di riferimento della sessione corrente. Il server Node.js riceve la richiesta, assembla il prompt completo inserendo la trascrizione e le liste nei segnaposto del template, e inoltra il tutto alle API di Claude. Claude analizza la trascrizione, associa "supermercato centrale" a "Supermercato Centrale Srl" tramite corrispondenza semantica con la lista dei clienti, identifica l'intento di registrazione incasso e produce il file JSON strutturato. Il middleware restituisce il file JSON all'applicazione, che legge ad alta voce la frase contenuta nel campo `risposta_vocale` e consegna il payload operativo al dispatcher. L'intero flusso, dal momento in cui l'autista termina di parlare al momento in cui la risposta vocale viene letta dal dispositivo, si completa tipicamente in un tempo compreso tra uno e tre secondi.

4.2 La scelta del modello linguistico

La scelta del modello di Intelligenza Artificiale da utilizzare come motore di comprensione del linguaggio naturale ha rappresentato una delle decisioni progettuali più rilevanti dell'intero progetto. Il modello selezionato è Claude Haiku 4.5, appartenente alla famiglia di modelli linguistici di grandi dimensioni sviluppata da Anthropic.

4.2.1 I criteri di selezione

I criteri che hanno guidato la scelta del modello sono stati principalmente tre. Il primo è la latenza: in un contesto operativo in cui l'autista si aspetta una risposta quasi immediata dopo aver pronunciato un comando, il tempo di elaborazione deve essere il più basso possibile, idealmente inferiore ai due secondi. Il secondo è il costo per richiesta: ogni chiamata alle API del modello ha un costo proporzionale al numero di token consumati, e in uno scenario di produzione con centinaia di autisti che effettuano decine di chiamate al giorno, il costo complessivo deve rimanere sostenibile. Il terzo è la qualità della comprensione: il modello deve essere in grado di comprendere correttamente comandi in lingua italiana, anche formulati in modo frammentario o con errori di pronuncia, e deve produrre output JSON rigorosamente validi senza aggiungere testo spurio.

4.2.2 La famiglia Claude di Anthropic

Anthropic è un'azienda statunitense fondata nel 2021 da ex ricercatori di OpenAI, focalizzata sullo sviluppo di sistemi di Intelligenza Artificiale sicuri e affidabili. La famiglia di modelli Claude comprende diverse varianti ottimizzate per casi d'uso differenti:

- *Claude Opus*, il modello più potente, pensato per compiti di ragionamento complesso e analisi approfondita;
- *Claude Sonnet*, un modello bilanciato tra potenza e velocità, adatto alla maggior parte dei casi d'uso generici;
- *Claude Haiku*, il modello più veloce ed economico, ottimizzato per compiti che richiedono bassa latenza e alti volumi di richieste.

Claude Haiku 4.5 si è rivelato il candidato ottimale per questo caso d'uso: è il modello più veloce ed economico della famiglia, pur mantenendo capacità di comprensione del linguaggio naturale sufficienti per il tipo di comandi che gli autisti pronunciano. Rispetto ai modelli più potenti della stessa famiglia, Haiku sacrifica la capacità di ragionamento su problemi complessi in favore di una latenza significativamente inferiore, un compromesso perfettamente accettabile nel contesto di MyDelivery, dove i comandi sono relativamente semplici e il contesto è sempre fornito esplicitamente ad ogni richiesta.

Nel contesto di questo progetto, il modello Claude non viene utilizzato come un assistente conversazionale generico, bensì come un traduttore di intenti altamente specializzato; esso riceve un comando in linguaggio naturale e restituisce esclusivamente un file JSON strutturato, senza mai impegnarsi in conversazioni aperte o generare testo libero al di fuori del campo `risposta_vocale`. Questa specializzazione è ottenuta interamente attraverso il prompt di sistema, descritto nella sezione successiva.

4.3 La progettazione del prompt di sistema

Il prompt di sistema rappresenta il componente intellettuale più rilevante dell'intero progetto. Si tratta di un documento di istruzioni dettagliato, lungo circa duemilacinquecento parole (pari a circa 3.400 token), che viene inviato a Claude insieme ad ogni richiesta dell'autista. Il suo compito è trasformare un modello linguistico generico in un traduttore di intenti rigido e deterministico, capace di produrre esclusivamente output JSON validi e conformi alla struttura attesa dall'applicazione Flutter.

4.3.1 La struttura del prompt

Il prompt è organizzato in cinque sezioni principali, ciascuna con un ruolo specifico. La prima è la definizione del ruolo, che stabilisce l'identità del modello come assistente virtuale vocale integrato nell'applicazione MyDelivery, specificando che il suo unico compito è tradurre comandi in linguaggio naturale italiano in azioni JSON strutturate. La seconda è la lista delle regole fondamentali, che impone i vincoli di formato (solo JSON valido), struttura (sempre i due nodi `risposta_vocale` e `dati_operazione`), corrispondenza esatta con le chiavi previste, utilizzo delle liste di riferimento per il matching e gestione degli errori. La terza è la lista degli intenti con esempi, che fornisce al modello quattordici coppie complete di input e output atteso, una per ciascun tipo di operazione. La quarta è la sezione dei segnaposto, dove il middleware inserisce a runtime le liste di riferimento e la trascrizione vocale della sessione corrente. La quinta è il marcatore di inizio elaborazione, che segnala al modello di procedere con la conversione.

La tecnica di prompting adottata è quella del *few-shot prompting*: invece di spiegare al modello in modo astratto cosa deve fare, si forniscono ad esso esempi concreti di input e output attesi per ciascun tipo di operazione. Questa tecnica si è rivelata molto efficace nel garantire la conformità degli output, poiché il modello tende a replicare fedelmente la struttura degli esempi forniti, anche per varianti del comando che non erano state esplicitamente anticipate.

4.3.2 Il fuzzy matching dinamico

Una delle sfide più impegnative nell'interazione vocale con un'applicazione gestionale è la discrepanza tra ciò che l'utente pronuncia e ciò che è effettivamente registrato nel sistema. Un autista potrebbe dire "Firenze Centrale" quando nel database il giro si chiama "Firenze Centro", oppure "il supermercato" quando la ragione sociale completa è "Supermercato

Centrale Srl". Se il sistema cercasse una corrispondenza esatta, la stragrande maggioranza delle richieste fallirebbe.

Per risolvere questo problema è stata progettata una tecnica di fuzzy matching dinamico, interamente delegata al modello Claude attraverso le istruzioni nel prompt. Il meccanismo funziona nel modo seguente: ad ogni chiamata, l'applicazione Flutter allega alla trascrizione vocale le liste di riferimento della sessione corrente, ovvero gli array JSON contenenti i nomi esatti dei clienti del giorno, dei giri di consegna disponibili, degli articoli cauzionali e dei metodi di pagamento. Il server middleware inietta queste liste nel prompt di sistema nel segnaposto dedicato. Il prompt contiene un'istruzione esplicita che ordina al modello di confrontare ogni nome pronunciato dall'utente con le stringhe presenti nelle liste e di selezionare quella più simile dal punto di vista fonetico o semantico. Il modello restituisce nel file JSON di output la stringa esatta presente nella lista, non quella pronunciata dall'utente.

Questo approccio presenta un vantaggio fondamentale rispetto agli algoritmi di fuzzy matching tradizionali, come la distanza di Levenshtein o l'algoritmo Soundex: il modello linguistico è in grado di comprendere la relazione semantica tra le parole, non solo la somiglianza fonetica. Di conseguenza, è capace di associare "il bar" a "Bar Roma" o "contanti" al codice `CONT`, operazioni che un algoritmo puramente basato sulla distanza tra stringhe non potrebbe eseguire correttamente.

Un aspetto rilevante di questa scelta progettuale è la sua generalità: poiché il matching è delegato al modello e le liste di riferimento sono fornite dinamicamente ad ogni richiesta, il sistema funziona correttamente per qualsiasi azienda cliente di MyDelivery senza richiedere alcuna personalizzazione del middleware o del prompt. Una nuova azienda con clienti, giri e articoli cauzionali completamente diversi viene gestita automaticamente non appena la sua applicazione Flutter invia le proprie liste di riferimento.

4.3.3 La memoria operativa e la gestione dello stato

Un'ulteriore dimensione del prompt di sistema riguarda la gestione della continuità contestuale tra comandi successivi. In una sessione operativa reale, un autista non enuncia ogni comando come un'istruzione completamente autonoma: egli pronuncia "aprilo" dopo aver nominato un giro, o "il primo" dopo aver richiesto un elenco. Per consentire al modello di risolvere correttamente questi riferimenti relativi, il prompt include un insieme di regole di memoria e il sistema implementa un meccanismo di stato persistente tra una richiesta e la successiva.

A ogni risposta, il modello produce obbligatoriamente un oggetto `nuovo_stato` contenente tre campi: `citta_selezionata`, `giro_selezionato` e `ultimo_cliente_citato`. Il widget Flutter riceve questo oggetto, lo conserva in memoria e lo reinvia al middleware come campo `stato_sessione` nella richiesta successiva. Il prompt istruisce il modello a consultare questo stato per risolvere i riferimenti impliciti presenti nel nuovo comando, producendo risposte contestualmente coerenti anche in presenza di formulazioni ellittiche.

Il prompt prevede inoltre regole di proattività sui dati: se l'utente richiede un'azione come un incasso senza specificare l'importo, ma questo è disponibile nel campo `dettaglio_clienti` delle liste di riferimento, il modello lo recupera autonomamente e lo propone nella risposta vocale, chiedendo conferma all'autista prima di procedere.

4.3.4 La gestione degli errori

Il prompt prevede esplicitamente un meccanismo di gestione degli errori che entra in gioco quando il modello non riesce a comprendere il comando dell'utente, quando mancano dati essenziali per completare l'operazione richiesta, o quando il nome pronunciato non presenta una corrispondenza plausibile con nessun elemento delle liste di riferimento.

In tutti questi casi, il modello è istruito a restituire un file JSON con l'azione `errore_comprensione` e una `risposta_vocale` che spiega il problema in modo chiaro e invita l'utente a ripetere il comando con le informazioni mancanti. Questo garantisce che l'applicazione Flutter riceva sempre un file JSON valido e correttamente strutturato, anche in caso di fallimento, evitando crash o comportamenti imprevisti nel dispatcher. Il modello non deve mai inventare valori per colmare lacune informative: se un importo non è stato specificato in una richiesta di incasso, la risposta corretta è un errore esplicito, non un importo arbitrario.

4.4 La mappatura delle azioni operative

Il contratto di comunicazione tra l'Intelligenza Artificiale e l'applicazione Flutter è costituito da un insieme di quattordici azioni operative, ciascuna identificata da una stringa univoca nel campo `action` del file JSON di risposta. Queste azioni coprono la totalità delle funzionalità dell'applicazione che si prestano al controllo vocale e sono organizzate in cinque categorie funzionali. La Tabella 4.1 riepiloga tutte le azioni con la relativa descrizione.

Azione	Descrizione
`filtra_giri_consegna`	filtra i giri per stato (tutti, in consegna, consegnati, da fare)
`apri_giro_consegna`	apre il dettaglio di un giro specifico
`cambia_vista_documenti`	alterna tra vista a elenco e vista su mappa
`riordina_documenti`	sposta una consegna nell'elenco del giro
`gestisci_consegna`	aggiorna l'esito di una consegna con referente e annotazioni
`visualizza_dettagli_documento`	apre una sezione del documento (articoli, allegati, partite)
`registra_incasso_documento`	registra un incasso con supporto al pagamento multiplo
`incassa_partita_aperta`	incassa un credito pregresso
`registra_resi_cauzioni`	registra il reso di vuoti e imballi
`stampa_ricevuta_consegna`	stampa la ricevuta per il cliente corrente
`filtra_cronologia`	imposta le date nella sezione cronologia
`stampa_report_cronologia`	genera un report di fine giornata
`impostazioni_firma`	apre la schermata per la firma digitale dell'autista
`errore_comprensione`	comando non compreso o dati essenziali mancanti

Tabella 4.1: Riepilogo delle quattordici azioni operative mappate nel prompt di sistema.

4.4.1 La struttura del JSON di risposta

Ogni risposta prodotta dal modello contiene tre nodi principali, come mostrato nel Listato 4.1.

```
{
  "risposta_vocale": "Frase da leggere all'autista.",
  "dati_operazione": {
    "action": "nome_azione",
    "parameters": { ... }
  },
  "nuovo_stato": {
    "citta_selezionata": "nome o null",
    "giro_selezionato": "nome o null",
```

```
      "ultimo_cliente_citato": "nome o null"
    }
  }
```

Listato 4.1: Struttura base del file JSON di risposta.

Il campo `risposta_vocale` contiene una frase discorsiva, concisa e professionale che l'applicazione leggerà ad alta voce tramite il motore di *text-to-speech* del dispositivo. Il campo `dati_operazione` contiene il payload tecnico con l'azione da eseguire e i relativi parametri. Il campo `nuovo_stato` è il meccanismo con cui il modello aggiorna la propria memoria operativa tra un comando e il successivo: esso conserva la città e il giro attualmente selezionati e l'ultimo cliente citato, consentendo al sistema di risolvere riferimenti relativi, come "aprilo", "il primo" o "quello di prima" nelle richieste successive. Questa separazione netta tra la componente vocale, pensata per l'essere umano, e la componente operativa, pensata per il software, consente al dispatcher Flutter di ignorare completamente il testo della risposta vocale e di concentrarsi esclusivamente sull'interpretazione dei parametri strutturati.

4.4.2 Le azioni di navigazione e visualizzazione

La prima categoria comprende le azioni che consentono all'autista di navigare tra le schermate dell'applicazione e di modificare la modalità di visualizzazione dei dati. L'azione `filtra_giri_consegna` consente di filtrare l'elenco dei giri per stato: i valori validi per il parametro `stato` sono `tutti`, `in_consegna`, `consegnati` e `da_fare`. L'azione `apri_giro_consegna` consente di entrare nel dettaglio di uno specifico giro, identificato dal parametro `nome_giro` che viene "valorizzato" con la stringa esatta presente nella lista di riferimento dopo il fuzzy matching. L'azione `cambia_vista_documenti` consente di passare dalla visualizzazione a elenco a quella su mappa, e viceversa, tramite il parametro `tipo_vista`, che accetta i valori `lista` e `mappa`. L'azione `riordina_documenti` consente di spostare una consegna all'interno dell'elenco, specificando il nome del cliente da spostare e la nuova posizione desiderata.

4.4.3 Le azioni di gestione operativa

La seconda categoria copre le operazioni legate alla gestione dello stato delle consegne. L'azione `gestisci_consegna` consente di aggiornare l'esito di un documento: i parametri includono il nome del cliente mappato dalla lista, l'esito selezionabile tra "Consegna Effettuata" e "Non Consegnato", il nome del referente e le eventuali annotazioni. Questa azione aggrega in un unico comando vocale tre informazioni che nell'interfaccia touch richiedono l'interazione con tre campi separati. L'azione `visualizza_dettagli_documento` consente di aprire una sezione specifica del documento corrente; il parametro `sezione` accetta i valori `articoli`, `allegati` e `partite`.

4.4.4 Le azioni di gestione amministrativa

La terza categoria rappresenta il nucleo contabile dell'applicazione e comprende le azioni più articolate dal punto di vista della struttura JSON. L'azione `registra_incasso_documento` gestisce la registrazione di un incasso sul documento corrente. La sua complessità risiede nel supporto al pagamento multiplo; il parametro `pagamenti` è un array di oggetti, ciascuno con un campo `metodo_pagamento` e un campo `importo`, che consente di rappresentare in un'unica struttura incassi suddivisi su più metodi. Nel Listato 4.2 è mostrato un esempio concreto per un comando del tipo "aggiungi un incasso di 150 euro in contanti e 50 euro con assegno".

```
{
  "risposta_vocale": "Sto per registrare un incasso diviso
    in 150 euro in contanti e 50 euro in assegni.",
  "dati_operazione": {
    "action": "registra_incasso_documento",
    "parameters": {
      "pagamenti": [
        { "metodo_pagamento": "CONT", "importo": 150.00 },
        { "metodo_pagamento": "ASSE", "importo": 50.00 }
      ],
      "annotazioni": ""
    }
  },
  "nuovo_stato": {
    "citta_selezionata": "Firenze",
    "giro_selezionato": "Firenze Centro",
    "ultimo_cliente_citato": "Supermercato Centrale Srl"
  }
}
```

Listato 4.2: Esempio di file JSON per un incasso con pagamento multiplo.

I codici dei metodi di pagamento vengono normalizzati dal modello a partire da sinonimi comuni: “contanti” o “cash” producono il codice `CONT`, “carta” o “carta di credito” producono `CARD`, “bonifico” produce `BANC` e “assegno” produce `ASSE`. L’azione `incassa_partita_aperta` gestisce, invece, l’incasso di crediti pregressi non legati alla consegna corrente. L’azione `registra_resi_cauzioni` gestisce il rientro di vuoti e imballi; il parametro `articoli_resi` è un array di oggetti, ciascuno con la descrizione dell’articolo (mappata dalla lista cauzioni) e la quantità resa. Nel Listato 4.3 è mostrato un esempio per un comando del tipo “reso di tre pedane e due contenitori”.

```
{
  "risposta_vocale": "Sto per registrare il reso di
    3 pedane e 2 contenitori refrigerati.",
  "dati_operazione": {
    "action": "registra_resi_cauzioni",
    "parameters": {
      "articoli_resi": [
        { "descrizione": "Pedane", "quantita": 3 },
        { "descrizione": "Contenitori refrigerati",
          "quantita": 2 }
      ]
    }
  },
  "nuovo_stato": {
    "citta_selezionata": "Firenze",
    "giro_selezionato": "Firenze Centro",
    "ultimo_cliente_citato": "Alimentari Freschi"
  }
}
```

Listato 4.3: Esempio di file JSON per un reso vuoti e cauzioni.

4.4.5 Le azioni di reportistica e impostazioni

La quarta categoria comprende le azioni legate alla generazione di report e alla configurazione dell'applicazione. L'azione `stampa_ricevuta_consegna` avvia la stampa della ricevuta per il documento corrente e non richiede parametri aggiuntivi, poiché il contesto del documento è già noto all'applicazione. L'azione `filtra_cronologia` consente di impostare le date nella sezione di cronologia; il modello converte le espressioni temporali dell'utente in date nel formato `YYYY-MM-DD`. L'azione `stampa_report_cronologia` genera i riepiloghi giornalieri e accetta un parametro `tipo_report` con i valori `consegne_pagamenti` e `riepilogo_incassi`. L'azione `impostazioni_firma` apre la schermata per l'acquisizione della firma digitale dell'autista.

4.4.6 L'azione di gestione degli errori

La quinta categoria contiene la sola azione `errore_comprensione`, prodotta dal modello quando il comando dell'utente non rientra in nessuno dei quattordici scenari definiti, quando mancano dati essenziali per completare l'operazione o quando un nome pronunciato non trova corrispondenza nelle liste di riferimento. Il campo `parameters` è sempre un oggetto vuoto, e la `risposta_vocale` contiene una spiegazione chiara del problema che guida l'utente verso la formulazione corretta del comando successivo. In nessun caso il modello deve tentare di completare un'operazione con dati parziali o inventati.

4.5 Le liste di riferimento e il contesto dinamico

Un aspetto fondamentale dell'architettura è il meccanismo con cui il contesto operativo viene fornito al modello ad ogni richiesta. Come anticipato nel Capitolo 3, le liste di riferimento non sono statiche; esse cambiano in funzione della giornata lavorativa, dell'autista connesso e dell'azienda cliente. L'applicazione Flutter estrae queste liste dal proprio database SQLite locale al momento della richiesta e le allega alla trascrizione.

A differenza della struttura piatta originariamente progettata, il sistema trasmette al middleware una gerarchia completa Città → Giri → Clienti, come mostrato nel Listato 4.4. Il middleware si occupa di appiattire tale struttura tramite la funzione `appiattisciStruttura()`, che produce le liste piatte e la mappa `dettaglio_clienti` prima di iniettarle nel prompt. Questo approccio consente al modello di ricevere un contesto compatto e già elaborato, riducendo il numero di token in ingresso e la latenza complessiva.

```
{
  "struttura_dati": {
    "citta": [
      {
        "nome": "Firenze",
        "giri": [
          {
            "nome": "Firenze Centro",
            "clienti": [
              {
                "nome": "Alimentari Freschi",
```

```
                "da_incassare": 150.00,
                "articoli_cauzioni": ["Pedane"]
              }
            ]
          }
        ]
      }
    ],
    "metodi_pagamento": ["CONT", "BANC", "CARD", "ASSE"],
    "articoli_cauzioni": ["Pedane", "Contenitori
   refrigerati"]
  }
}
```

Listato 4.4: Struttura gerarchica delle liste di riferimento inviate dall'applicazione al middleware.

La lista `clienti` (nella forma appiattita) contiene le ragioni sociali dei clienti presenti nei giri di consegna della giornata corrente. La mappa `dettaglio_clienti` associa ciascun cliente ai propri dati operativi: l'importo da incassare, gli articoli cauzionali attesi e i riferimenti al giro e alla città di appartenenza. Questa struttura di dettaglio consente al modello di applicare le regole di proattività descritte nella Sezione 4.3.3, recuperando autonomamente i dati necessari quando l'utente omette informazioni presenti nel contesto.

Quando l'applicazione non fornisce le liste, il middleware inserisce nel prompt la stringa "Nessun dato fornito" e il modello gestisce il caso producendo un'azione `errore_comprensione` ogni volta che una lista sarebbe stata necessaria per completare l'operazione richiesta.

4.6 La progettazione del pacchetto Flutter

Parallelamente alla progettazione del middleware, è stato definito il contratto di integrazione del pacchetto Flutter `mydelivery_voice_assistant`, che i programmatori di Infoservice dovranno integrare nell'applicazione MyDelivery. Il pacchetto è progettato come un componente isolato e indipendente dal codice sorgente dell'applicazione esistente: è sufficiente importarlo come dipendenza, inserire il widget del microfono nelle schermate desiderate, fornire le liste di riferimento estratte dal database SQLite locale e implementare il dispatcher nel callback dedicato.

4.6.1 L'architettura interna del pacchetto

Il pacchetto è organizzato in cinque aree funzionali distinte. I modelli dati definiscono la struttura della richiesta HTTP (`VoiceCommandRequest`), della risposta JSON (`VoiceCommandResponse` e `OperationData`) e degli stati del widget (`AssistantState`). I servizi incapsulano la logica di riconoscimento vocale (`SpeechRecognitionService`), di sintesi vocale (`TextToSpeechService`) e di comunicazione HTTP con il middleware (`MiddlewareClient`). Il widget principale (`VoiceAssistantButton`) orchestra l'intero flusso, gestisce lo stato di sessione in memoria e fornisce il feedback visivo all'autista. La configurazione (`AssistantConfig`) consente di personalizzare l'URL del server, il timeout, la lingua e la velocità della sintesi vocale. La classe di utilità `DataFlattener`, infine, espone un metodo statico per appiattire la struttura gerarchica dei dati in liste piatte e in una mappa di dettaglio, replicando lato client la stessa trasformazione che il middleware esegue prima di costruire il prompt.

4.6.2 La macchina a stati del widget

Il widget del microfono implementa una macchina a stati con cinque stati distinti, ciascuno associato a un feedback visivo specifico che guida l'utente durante il ciclo di elaborazione del comando. Lo stato `idle` corrisponde all'attesa della pressione del pulsante e viene rappresentato con il colore blu e l'icona del microfono. Lo stato `listening` si attiva alla pressione del pulsante: il colore diventa rosso e il pulsante pulsa con un'animazione, segnalando che il sistema è in ascolto. Lo stato `processing` si attiva quando la trascrizione è stata inviata al server: il colore arancione e un indicatore di caricamento circolare comunicano all'utente che l'elaborazione è in corso. Lo stato `speaking` si attiva alla ricezione della risposta: il colore verde e l'icona dell'altoparlante indicano che il sistema sta leggendo la risposta vocale. Lo stato `error` si attiva in caso di errore non recuperabile e torna automaticamente a `idle` dopo tre secondi.

Il diagramma della macchina a stati è illustrato nella Figura 4.2, che mostra le transizioni tra i cinque stati in risposta agli eventi del sistema.

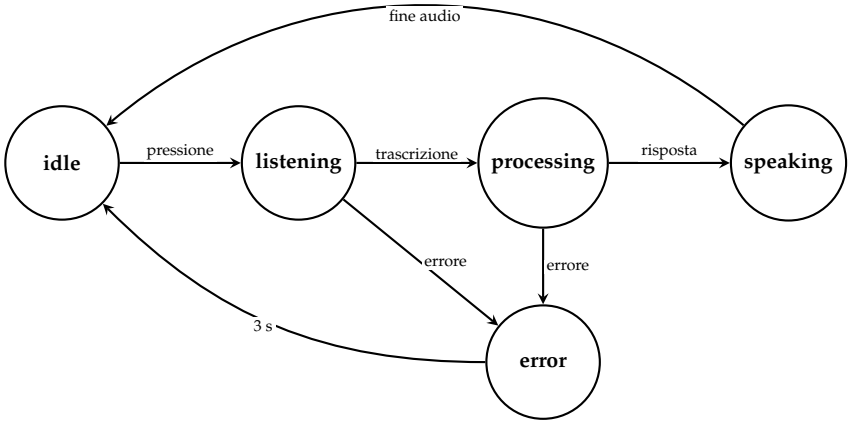


Figura 4.2: Macchina a stati del widget `VoiceAssistantButton`; sono evidenziati i cinque stati e le transizioni tra essi in risposta agli eventi del ciclo di elaborazione.

4.6.3 Il contratto di integrazione

Il contratto di integrazione tra il pacchetto e l'applicazione principale è volutamente minimo. Il team di sviluppo di Infoservice dovrà svolgere tre attività. La prima è importare il pacchetto come dipendenza nel file `pubspec.yaml` dell'applicazione e configurare i permessi per il microfono nei manifest di Android e iOS. La seconda è inserire il widget `VoiceAssistantButton` nelle schermate dell'applicazione, passando ad esse la configurazione con l'URL del middleware e le liste di riferimento estratte dal database SQLite locale. La terza è implementare la logica del dispatcher nel callback `onActionReceived`, che riceve l'oggetto `VoiceCommandResponse` e, tramite uno *switch* sul campo `action`, esegue le operazioni native corrispondenti sull'applicazione.

4.7 La sicurezza della comunicazione

La comunicazione tra i tre livelli dell'architettura è progettata per garantire la protezione dei dati in transito e delle credenziali di accesso al servizio di Intelligenza Artificiale. La chiave API di Anthropic è conservata esclusivamente nel file `.env` sul server middleware e non viene mai esposta al client mobile né inclusa nel codice sorgente versionato. Il file `.env`

è escluso dal sistema di controllo della versione tramite `.gitignore`, impedendo che venga accidentalmente incluso in un repository.

Per l'ambiente di test utilizzato durante il tirocinio, il server è stato configurato per accettare connessioni sulla porta 3000 in modalità HTTP locale. Per un ambiente di produzione, Infoservice prevede di posizionare un reverse proxy (come Caddy o Nginx) davanti al server Node.js, configurando la terminazione TLS con certificati Let's Encrypt e garantendo che tutto il traffico tra l'applicazione mobile e il middleware transiti su connessione HTTPS cifrata. In tale configurazione, la porta 3000 del server Node.js non sarebbe esposta direttamente a internet, ma sarebbe raggiungibile solo internamente attraverso il reverse proxy sulla porta 443. Gli aspetti relativi alla messa in sicurezza dell'ambiente di produzione sono discussi più in dettaglio nel Capitolo 6, nella sezione dedicata ai possibili sviluppi futuri.

4.8 L'alternativa locale: SLM e Ollama

È importante premettere una distinzione netta, che vale per l'intera sezione: il sistema descritto fino a questo punto e consegnato in produzione utilizza esclusivamente Claude Haiku 4.5 tramite le API di Anthropic, con il prompt di sistema completo di circa 3400 token e il meccanismo di prompt caching descritto nel Capitolo 5. Quanto segue descrive, invece, un esperimento di ricerca e sviluppo condotto in parallelo, che ha richiesto un prompt diverso, più corto e progettato specificamente per un modello di dimensioni ridotte; questo prompt sperimentale non sostituisce, né modifica in alcun modo, il prompt di produzione utilizzato con Claude. Le due architetture — quella di produzione basata su Claude e quella sperimentale basata su Ollama — restano completamente separate per tutta la trattazione che segue, ed è bene non confondere i numeri relativi all'una con quelli relativi all'altra.

Parallelamente alla progettazione e all'implementazione del sistema basato sulle API di Anthropic, è stata condotta un'analisi sperimentale sulla fattibilità di sostituire Claude con un modello linguistico di piccole dimensioni (*Small Language Model*, SLM) eseguito direttamente sulla macchina server, senza dipendere da servizi cloud di terze parti. Questa analisi, pur non essendo parte del sistema consegnato in produzione, ha prodotto risultati quantitativi dettagliati e ha permesso di definire un quadro preciso dei trade-off tra le due architetture, utile sia per orientare le decisioni di sviluppo futuro sia per comprendere i limiti strutturali dell'approccio locale nel contesto operativo di MyDelivery.

4.8.1 La motivazione e il contesto

L'ipotesi di partenza era quella di valutare se fosse possibile, in linea di principio, eliminare la dipendenza dalle API di Anthropic sostituendo il terzo livello dell'architettura con un modello in esecuzione locale tramite Ollama, un runtime open source per modelli in formato GGUF. L'obiettivo era duplice: stimare l'eventuale azzeramento dei costi per chiamata API e, soprattutto, stimare un limite inferiore di fattibilità per un'eventuale futura esecuzione completamente on-device sul dispositivo mobile dell'autista, che permetterebbe il funzionamento dell'assistente anche in assenza di connessione di rete.

È importante chiarire fin da subito la portata reale di questo esperimento, per evitare un'interpretazione eccessiva dei suoi risultati. Il test qui descritto è stato condotto eseguendo Ollama sulla stessa macchina server già utilizzata per il middleware, non sul dispositivo mobile dell'autista. Di conseguenza, l'esperimento non dimostra né verifica direttamente la fattibilità di un'esecuzione on-device, ma fornisce un dato preliminare necessario a tale valutazione: se anche su un server dedicato, con risorse di calcolo superiori a quelle di uno smartphone, un modello di 1,5 miliardi di parametri presenta tempi di elaborazione incompatibili con l'uso vocale in tempo reale (come verrà mostrato nelle sezioni seguenti), questo

costituisce un'indicazione, seppur non una prova rigorosa, che l'esecuzione sullo stesso modello su un dispositivo mobile, dotato di risorse di calcolo inferiori, sarebbe difficilmente più favorevole. Il test server-side va quindi inteso come una stima di un limite inferiore per la latenza ottenibile, utile per orientare la valutazione architetturale, e non come una dimostrazione diretta delle capacità di un'esecuzione realmente offline e on-device, che non è stata oggetto di misurazione in questa tesi e che richiederebbe test specifici su hardware mobile per essere valutata con rigore.

Dal punto di vista architetturale, la sostituzione è concettualmente semplice: il middleware Node.js rimane invariato in tutti i suoi componenti — endpoint, prompt, funzioni di estrazione e validazione del file JSON — e l'unica modifica riguarda la funzione di chiamata al modello, che passa da `anthropic.messages.create()` a una richiesta HTTP verso l'API locale di Ollama sulla porta 11434. Il flusso operativo rimane identico a quello descritto nella sezione precedente, con l'app Flutter che invia la trascrizione al middleware e riceve un file JSON strutturato in risposta. Il secondo server, denominato `server-local.js`, è stato configurato per ascoltare sulla porta 3001, consentendo l'esecuzione parallela ai due sistemi sulla stessa macchina per facilitare il confronto diretto.

4.8.2 Il modello scelto e i criteri di selezione

Il modello selezionato per i test è stato Qwen2.5-1.5B-Instruct, sviluppato da Alibaba Cloud e disponibile su Hugging Face in formato GGUF quantizzato. La scelta è ricaduta su questo modello per tre ragioni principali: la buona comprensione nativa dell'italiano rispetto ad alternative come Llama-3.2-1B, prevalentemente addestrato in inglese; la capacità di generare output JSON strutturati superiore a quella di SmolLM2-1.7B nella stessa fascia dimensionale; e un'occupazione di memoria contenuta, con un peso quantizzato in formato Q4_K_M di circa un gigabyte, compatibile con ambienti server a risorse limitate.

Sono stati considerati anche altri candidati prima di convergere sul modello Qwen2.5-1.5B. NuExtract-tiny (0.5B), basato su Phi-3 e addestrato specificamente per l'estrazione di dati strutturati, è risultato inadatto al fuzzy matching semantico richiesto dal sistema; la sua architettura privilegia l'estrazione da testi predefiniti rispetto alla comprensione di intenti espressi in linguaggio naturale colloquiale. Phi-3.5-mini (3.8B) offre capacità di ragionamento superiori ma richiede oltre sei gigabyte di RAM e produce latenze incompatibili con un uso vocale in tempo reale anche su hardware server dedicato. Qwen2.5-0.5B è stato utilizzato nelle primissime fasi di prototipazione rapida per verificare il funzionamento dell'infrastruttura, ma la sua precisione sul dominio specifico si è rivelata insufficiente per le azioni più articolate, come la gestione degli incassi con pagamento multiplo e il fuzzy matching su nomi di clienti espressi in forma abbreviata.

CAPITOLO 5

Implementazione

In questo capitolo vengono descritte nel dettaglio tutte le fasi operative che hanno portato dall'architettura progettata nel capitolo precedente a un sistema funzionante e testabile. Si parte dalla configurazione della macchina server Ubuntu fornita dall'azienda, documentando ogni passaggio con i log reali della console. Si presenta, poi, il codice sorgente completo del server middleware Node.js, analizzandone ogni sezione e spiegandone le scelte implementative. Si descrivono i file di configurazione e di supporto necessari al funzionamento del sistema, incluso lo script di test interattivo utilizzato per la validazione dei comandi vocali. Si documentano i test di funzionamento effettuati sia sulla macchina server tramite `curl` *e script interattivo, sia su un dispositivo mobile Android fisico tramite l'applicazione Flutter di esempio, includendo le problematiche di raggiungibilità del server emerse durante la fase di test e le soluzioni adottate. Si presenta, infine, il pacchetto Flutter* `mydelivery_voice_assistant`*, descrivendone la struttura, i componenti interni e il contratto di integrazione con l'applicazione principale. Il capitolo si chiude con la descrizione del server gemello sviluppato per la valutazione sperimentale dell'alternativa locale basata su Ollama.*

5.1 L'infrastruttura server

Per ospitare il middleware, l'azienda Infoservice ha messo a disposizione una macchina server Linux Ubuntu dedicata, accessibile esclusivamente tramite protocollo SSH. Questa macchina rappresenta l'ambiente in cui il server Node.js viene eseguito e costituisce il secondo livello dell'architettura a tre livelli descritta nel Capitolo 4.

5.1.1 Parametri di accesso

I parametri di accesso alla macchina server sono i seguenti:

- indirizzo IP: `80.211.124.113`;
- porta SSH: 54022 (modificata dalla porta standard 22 per ragioni di sicurezza);
- autenticazione: tramite username e password riservati.

La scelta di utilizzare una porta SSH non standard riduce il volume dei tentativi di accesso automatizzati da parte di bot che eseguono scansioni di massa sulla porta 22. Questa misura, pur non rappresentando una protezione definitiva di cybersecurity, contribuisce a ridurre il rumore nei log di sistema e a rendere la macchina meno visibile alle scansioni automatiche.

5.1.2 Primo accesso e ricognizione del sistema

Il primo accesso alla macchina è stato effettuato da un terminale Windows tramite il client SSH nativo. Nel Listato 5.1 viene riportato il log completo della sessione di primo accesso, che documenta la verifica dell'impronta digitale del server, l'accettazione della chiave e le informazioni di sistema restituite al momento del login.

```
C:\Users\fabio>ssh -p 54022 aiuser@80.211.124.113
The authenticity of host '[80.211.124.113]:54022
  ([80.211.124.113]:54022)' can't be established.
ED25519 key fingerprint is
  SHA256:pDGeI1PE1AQEcGDc9qnYpFlIGNkYUuA7eKSRFhFPzTM.
This key is not known by any other names.
Are you sure you want to continue connecting
  (yes/no/[fingerprint])? yes
Warning: Permanently added '[80.211.124.113]:54022'
  (ED25519) to the list of known hosts.
aiuser@80.211.124.113's password:
Welcome to Ubuntu 24.04.2 LTS
  (GNU/Linux 6.8.0-106-generic x86_64)

  System load: 0.08        Processes:    125
  Usage of /:  13.6% of 48.91GB
  Memory usage: 1%
  Swap usage:   0%
  Users logged in: 0
  IPv4 address for ens18: 192.168.114.198

Last login: Mon Mar 30 11:06:38 2026
  from 192.168.11.154
```

Listato 5.1: Log del primo accesso SSH alla macchina server.

Il log rivela diverse informazioni utili sulla macchina: il sistema operativo è Ubuntu 24.04.2 LTS con kernel Linux 6.8, il disco ha una capacità di circa 49 GB con il 13,6% utilizzato, la memoria è praticamente inutilizzata (1%) e il carico del sistema è minimo (0.08). Queste caratteristiche confermano che la macchina è dedicata esclusivamente al middleware e dispone di risorse ampiamente sufficienti per il carico di lavoro previsto.

Un dettaglio rilevante emerso dal log è l'indirizzo IPv4 dell'interfaccia di rete: `192.168.114.198`, un indirizzo appartenente allo spazio privato RFC 1918. Questo indica che la macchina è posizionata dietro un NAT gestito dal provider di hosting, e che l'indirizzo IP pubblico `80.211.124.113` è assegnato all'infrastruttura del provider, che si occupa di inoltrare il traffico alla macchina virtuale. Come si vedrà nella sezione dedicata ai test su dispositivo mobile, questa configurazione di rete ha avuto implicazioni dirette sulla raggiungibilità del server dall'esterno.

5.2 Configurazione dell'ambiente

Una volta effettuato l'accesso alla macchina, sono state eseguite le operazioni di configurazione necessarie per predisporre l'ambiente di esecuzione del server middleware. Queste operazioni sono state svolte in sequenza e vengono documentate di seguito nell'ordine in cui sono state effettuate.

5.2.1 Aggiornamento del sistema operativo

La prima operazione eseguita è stata l'aggiornamento completo del sistema operativo, per assicurarsi che tutti i pacchetti installati fossero alla versione più recente e che eventuali vulnerabilità di sicurezza note fossero state corrette:

```
sudo apt update && sudo apt upgrade -y
```

Listato 5.2: Aggiornamento del sistema operativo.

Il comando `apt update` aggiorna l'indice dei pacchetti disponibili dai repository configurati, mentre `apt upgrade` installa le versioni più recenti di tutti i pacchetti già presenti nel sistema. Il flag `-y` conferma automaticamente tutte le richieste di installazione, evitando la necessità di intervento manuale durante il processo.

5.2.2 Installazione di Node.js

L'installazione di Node.js è stata effettuata tramite il repository ufficiale di NodeSource, che fornisce versioni aggiornate e certificate del runtime. La scelta è ricaduta sulla versione 20 LTS (*Long Term Support*), che garantisce supporto e aggiornamenti di sicurezza a lungo termine:

```
curl -fsSL https://deb.nodesource.com/setup_20.x \
  | sudo -E bash -
sudo apt-get install -y nodejs
```

Listato 5.3: Installazione di Node.js 20 LTS tramite NodeSource.

Questa modalità di installazione è stata preferita rispetto al pacchetto `nodejs` disponibile nei repository standard di Ubuntu, che tipicamente contiene versioni più datate del runtime. Il progetto richiede una versione di Node.js pari o superiore alla 18 per il supporto completo delle funzionalità utilizzate dall'SDK di Anthropic, come le *top-level await* e le API `fetch` native. Dopo l'installazione, la corretta configurazione è stata verificata con i comandi riportati nel Listato 5.4.

```
node --version  # deve restituire v20.x o superiore
npm --version
```

Listato 5.4: Verifica della versione di Node.js e npm.

5.2.3 Installazione di PM2

PM2 è un process manager per applicazioni Node.js che consente di mantenere il server attivo in modo permanente, riavviandolo automaticamente in caso di crash e rilanciandolo al boot del sistema operativo. La sua installazione è stata effettuata globalmente tramite npm:

```
sudo npm install -g pm2
```

Listato 5.5: Installazione globale di PM2.

PM2 è stato scelto per la sua semplicità d'uso e per la capacità di gestire il ciclo di vita dell'applicazione senza richiedere la configurazione di unit file systemd personalizzati. In un ambiente di test come quello utilizzato per il tirocinio, PM2 rappresenta il giusto compromesso tra affidabilità e velocità di configurazione.

5.2.4 Creazione della cartella di progetto

La cartella di lavoro è stata creata nella home directory dell'utente `aiuser`:

```
mkdir ~/mydelivery
cd ~/mydelivery
npm init -y
npm install express dotenv @anthropic-ai/sdk cors
```

Listato 5.6: Creazione della cartella di progetto e inizializzazione.

Il comando `npm init -y` genera il file `package.json` con le impostazioni predefinite. Il comando `npm install` scarica e installa le quattro dipendenze necessarie al funzionamento del middleware:

- `express`: il framework web che gestisce le rotte HTTP e il parsing delle richieste JSON;
- `dotenv`: la libreria che carica le variabili d'ambiente dal file `.env` e le rende disponibili al processo Node.js;
- `@anthropic-ai/sdk`: l'SDK ufficiale di Anthropic per la comunicazione con le API di Claude;
- `cors`: la libreria che aggiunge automaticamente le intestazioni HTTP necessarie per consentire richieste cross-origin, indispensabile per il corretto funzionamento durante le sessioni di test con ngrok e per la futura integrazione con il reverse proxy di produzione.

Al termine dell'installazione, la struttura della cartella risultante è quella mostrata nel Listato 5.7.

```
aiuser@my-ai:~/mydelivery$ ls -la
total 52
drwxrwxr-x  3 aiuser aiuser  4096 Mar 30 13:18 .
drwxr-x---  6 aiuser aiuser  4096 Mar 30 13:39 ..
-rw-rw-r--  1 aiuser aiuser    95 Mar 30 12:32 .env
drwxrwxr-x 73 aiuser aiuser  4096 Mar 30 12:31
   node_modules
-rw-rw-r--  1 aiuser aiuser   345 Mar 30 12:31 package.
   json
-rw-rw-r--  1 aiuser aiuser 29456 Mar 30 12:31 package-
   lock.json
-rw-rw-r--  1 aiuser aiuser 14523 Mar 30 13:18 server.js
```

Listato 5.7: Struttura della cartella di progetto.

5.3 Il file di configurazione .env

Il file `.env` contiene le variabili d'ambiente necessarie al funzionamento del server. Questo file non viene mai versionato nel sistema di controllo del codice sorgente e risiede esclusivamente sulla macchina server, garantendo che le credenziali sensibili non siano esposte:

```
ANTHROPIC_API_KEY= xxxxxxxxxx
CLAUDE_MODEL=claude-haiku-4-5-20251001
MAX_TOKENS=1024
PORT=3000
```

Listato 5.8: Contenuto del file `.env`.

Ciascuna variabile ha un ruolo specifico:

- `ANTHROPIC_API_KEY`: la chiave segreta fornita da Anthropic per l'accesso alle API di Claude, ottenuta dal team tecnico dell'azienda.
- `CLAUDE_MODEL`: l'identificativo del modello da utilizzare. L'identificativo corretto del modello, come verificato durante la fase di test descritta in seguito, segue il formato che antepone il nome della variante al numero di versione. Il modello configurato nella versione definitiva del sistema è `claude-haiku-4-5-20251001`, corrispondente alla release di ottobre 2025 di Claude Haiku 4.5. Qualsiasi scostamento da questo schema produce un errore 404 nell'API, come documentato nella Sezione 5.6.
- `MAX_TOKENS`: il numero massimo di token che il modello può generare in ogni risposta, impostato a 1024 per contenere i costi mantenendo uno spazio sufficiente per i JSON più complessi.
- `PORT`: la porta su cui il server Node.js resta in ascolto, impostata a 3000.

È importante sottolineare che la porta 3000 è la porta interna del server Node.js e non ha alcuna relazione con la porta 54022, che è esclusivamente la porta SSH utilizzata per l'accesso remoto alla macchina ai fini di amministrazione. Le due porte operano su protocolli diversi (HTTP per la 3000, SSH per la 54022) e hanno scopi completamente indipendenti.

5.4 Il codice sorgente del server

Il file `server.js` contiene l'intera logica del middleware. Di seguito viene presentato il codice completo, suddiviso in sezioni logiche, con una spiegazione dettagliata di ciascuna parte.

5.4.1 Inizializzazione e configurazione

La prima sezione del file si occupa di importare le dipendenze, caricare le variabili d'ambiente e verificare che tutte le configurazioni necessarie siano presenti, come mostrato nel Listato 5.9.

Rispetto alla versione iniziale, il server include anche il pacchetto `cors`, installato tramite `npm install cors` e registrato come middleware globale tramite `app.use(cors())`. Questo consente al server di rispondere correttamente alle richieste provenienti da origini diverse, comprese quelle generate dal browser durante le sessioni di test con ngrok, senza che il meccanismo di *pre-flight* del protocollo HTTP blocchi le chiamate.

```
const express = require('express');
const Anthropic = require('@anthropic-ai/sdk');
const cors = require('cors');
require('dotenv').config();
```

```
const PORT = process.env.PORT || 3000;
const ANTHROPIC_API_KEY = process.env.ANTHROPIC_API_KEY;
const CLAUDE_MODEL = process.env.CLAUDE_MODEL
  || 'claude-haiku-4-5-20251001';
const MAX_TOKENS =
  parseInt(process.env.MAX_TOKENS, 10) || 1024;

if (!ANTHROPIC_API_KEY) {
  console.error(
    '[ERRORE FATALE] ANTHROPIC_API_KEY non configurata'
  );
  process.exit(1);
}

const app = express();
app.use(cors());
app.use(express.json({ limit: '100kb' }));

const anthropic = new Anthropic({
  apiKey: ANTHROPIC_API_KEY,
});
```

Listato 5.9: Sezione di inizializzazione del server.

La chiamata a `require('dotenv').config()` carica le variabili dal file `.env` e le rende disponibili attraverso l'oggetto `process.env`. Per ciascuna variabile viene fornito un valore predefinito tramite l'operatore logico `||`, in modo che il server possa avviarsi anche se una variabile opzionale non è stata configurata. La chiave API è obbligatoria: in sua assenza il server termina immediatamente con un messaggio di errore e il codice di uscita 1. Il middleware `cors()`, registrato prima di qualsiasi rotta, aggiunge automaticamente le intestazioni HTTP necessarie per consentire richieste provenienti da origini diverse, comprese quelle generate dal browser durante le sessioni di test con ngrok. Il middleware `express.json()` configura il parsing automatico del corpo delle richieste HTTP in formato JSON, con un limite di dimensione di 100 KB per prevenire l'invio di payload abnormi.

5.4.2 Il template del prompt di sistema

Il prompt di sistema è definito come una stringa costante contenente l'intero testo delle istruzioni per Claude, con due segnaposto (`{liste_di_riferimento}` e `{user_input}`) che vengono sostituiti a runtime con i dati effettivi di ciascuna richiesta. Il contenuto completo del prompt è quello descritto nel Capitolo 4 e non viene riportato qui per ragioni di spazio, ma la sua integrazione nel codice avviene tramite una semplice assegnazione a una costante, come mostrato nel Listato 5.10.

```
const SYSTEM_PROMPT_TEMPLATE = `SEI UN ASSISTENTE
  VIRTUALE VOCALE INTEGRATO NELL'APP "MYDELIVERY"...
  // (testo completo del prompt)
  ...
  Liste di riferimento fornite dal sistema:
  {liste_di_riferimento}

  Trascrizione utente:
```

```
    {user_input}

    JSON:`;
```

Listato 5.10: Dichiarazione del template del prompt nel codice.

Il prompt è stato inserito direttamente nel codice sorgente del server anziché in un file esterno, per semplificare il deployment e garantire che il template sia sempre allineato con la versione del server in esecuzione. Nella versione aggiornata del server, il template non viene più concatenato in un'unica stringa da inviare come campo `content` di tipo stringa; esso viene, invece, inserito come primo elemento di un array di blocchi di contenuto, annotato con la direttiva di caching descritta nella Sezione 5.4. I segnaposto `{liste_di_riferimento}` e `{user_input}` rimangono presenti nel template per compatibilità con il percorso di fallback, ma nella modalità operativa principale la composizione del messaggio avviene direttamente nell'endpoint, separando il contesto statico dalla trascrizione dinamica dell'utente. In uno scenario di produzione con aggiornamenti frequenti del prompt, potrebbe essere opportuno esternalizzarlo in un file JSON o YAML separato.

5.4.3 Funzioni di utilità

Il server implementa tre funzioni di utilità che gestiscono la composizione del prompt, l'estrazione del JSON dalla risposta di Claude e la validazione della struttura. Una quarta funzione di utilità, `appiattisciStruttura()`, è responsabile della trasformazione della struttura gerarchica ricevuta dall'applicazione Flutter in un formato più compatto e direttamente utilizzabile dal modello, come mostrato nel Listato 5.11.

```
function appiattisciStruttura(struttura) {
  const flat = {
    citta: [], giri: [], clienti: [],
    dettaglio_clienti: {},
    metodi_pagamento: struttura.metodi_pagamento || [],
    articoli_cauzioni: struttura.articoli_cauzioni || []
  };
  if (struttura.citta && Array.isArray(struttura.citta)) {
    struttura.citta.forEach(c => {
      flat.citta.push(c.nome);
      if (c.giri && Array.isArray(c.giri)) {
        c.giri.forEach(g => {
          flat.giri.push(g.nome);
          if (g.clienti && Array.isArray(g.clienti)) {
            g.clienti.forEach(cl => {
              flat.clienti.push(cl.nome);
              flat.dettaglio_clienti[cl.nome] = {
                da_incassare: cl.da_incassare,
                articoli_cauzioni: cl.articoli_cauzioni,
                giro: g.nome,
                citta: c.nome
              };
            });
          }
        });
      }
```

```
      });
    }
    return flat;
  }
```

Listato 5.11: Funzione di appiattimento della struttura dati gerarchica.

La funzione riceve la struttura gerarchica Città → Giri → Clienti e la trasforma in sei liste piatte: i nomi delle città, dei giri e dei clienti, la mappa `dettaglio_clienti`, che associa ogni cliente ai propri dati operativi, e le liste globali dei metodi di pagamento e degli articoli cauzionali. Questa ottimizzazione riduce la quantità di testo iniettato nel prompt rispetto all'alternativa di serializzare l'intera struttura gerarchica, abbassando il numero di token in ingresso e, di conseguenza, la latenza di ogni richiesta.

La funzione `compilaPrompt()`, mostrata nel Listato 5.12, effettua una semplice sostituzione dei segnaposto nel template con i dati reali della richiesta corrente.

```
function compilaPrompt(listeRiferimento, userInput) {
  return SYSTEM_PROMPT_TEMPLATE
    .replace('{liste_di_riferimento}',
      listeRiferimento)
    .replace('{user_input}', userInput);
}
```

Listato 5.12: Funzione di composizione del prompt.

La funzione `estraiJson()`, mostrata nel Listato 5.13, implementa tre strategie progressive per estrarre un file JSON valido dalla risposta di Claude. Sebbene il prompt istruisca il modello a restituire esclusivamente JSON puro, in rari casi il modello potrebbe aggiungere testo introduttivo o blocchi markdown attorno al JSON. Le tre strategie gestiscono questi casi in ordine di probabilità decrescente: prima si tenta il parsing diretto, poi si cerca il primo blocco delimitato da parentesi graffe, infine si rimuovono eventuali marcatori markdown.

```
function estraiJson(testoRisposta) {
  // strategia 1: il testo e' gia' JSON puro
  try {
    return JSON.parse(testoRisposta);
  } catch (_) {}
  // strategia 2: cerca il primo blocco { ... }
  const match = testoRisposta.match(/\{[\s\S]*\}/);
  if (match) {
    try {
      return JSON.parse(match[0]);
    } catch (_) {}
  }
  // strategia 3: rimuovi marcatori markdown
  const pulito = testoRisposta
    .replace(/```json\s*/gi, '')
    .replace(/```\s*/g, '')
    .trim();
  try {
    return JSON.parse(pulito);
  } catch (_) {}
  throw new Error(
```

```
    'Impossibile estrarre JSON valido.'
  );
}
```

Listato 5.13: Funzione di estrazione del file JSON dalla risposta.

La funzione `validaStrutturaRisposta()`, mostrata nel Listato 5.14, verifica che il file JSON prodotto da Claude contenga la struttura minima attesa dal dispatcher Flutter; quest'ultima consiste in un campo `risposta_vocale` di tipo stringa e in un oggetto `dati_operazione` contenente almeno i campi `action` e `parameters`.

```
function validaStrutturaRisposta(json) {
  return (
    json !== null &&
    typeof json === 'object' &&
    typeof json.risposta_vocale === 'string' &&
    json.dati_operazione !== null &&
    typeof json.dati_operazione === 'object' &&
    typeof json.dati_operazione.action === 'string'
    && json.dati_operazione.parameters !== undefined
  );
}
```

Listato 5.14: Funzione di validazione della struttura JSON.

5.4.4 La struttura della richiesta al modello e il caching

Una delle ottimizzazioni introdotte nella versione definitiva del server riguarda il modo in cui la richiesta viene costruita e inviata all'API di Anthropic. Nella versione iniziale, il prompt completo veniva assemblato in un'unica stringa tramite la funzione `compilaPrompt()`, che sostituiva i segnaposto con i dati della sessione corrente, e inviato come singolo messaggio di tipo `user`. Questo approccio, pur funzionalmente corretto, comportava che l'intera porzione statica del prompt — le istruzioni di sistema e il contesto dei dati appiattiti, che insieme costituiscono la parte dominante dell'input — venisse rielaborata integralmente ad ogni richiesta.

La versione aggiornata struttura il messaggio come un array di due blocchi di contenuto distinti, come mostrato nel Listato 5.15.

```
const messaggioUser = {
  role: 'user',
  content: [
    {
      type: 'text',
      text: `ISTRUZIONI DI SISTEMA E LISTE:\n`
          + `${SYSTEM_PROMPT_TEMPLATE}\n\n`
          + `DATI DI CONTESTO:\n${contextPerClaude}`,
      cache_control: { type: 'ephemeral' }
    },
    {
      type: 'text',
      text: `STATO SESSIONE: `
          + `${JSON.stringify(stato_sessione || {})}\n\n`
```

```
            + `COMANDO UTENTE: "${trascrizionePulita}"`
      }
    ]
  };
```

Listato 5.15: Composizione del messaggio con prompt caching.

Il primo blocco contiene le istruzioni di sistema e il contesto appiattito della sessione ed è annotato con `cache_control: { type: 'ephemeral' }`; questa direttiva segnala all'API di Anthropic di conservare in cache questa porzione del prompt per le richieste successive che presentano lo stesso contenuto. Poiché le istruzioni di sistema rimangono invariate per tutta la durata della sessione operativa e il contesto dei dati cambia solo se l'autista cambia giro o azienda, il meccanismo di caching elimina la necessità di rielaborare integralmente questa porzione ad ogni chiamata, riducendo la latenza percepita e il numero di token effettivamente fatturati.

Il secondo blocco contiene lo stato di sessione aggiornato e la trascrizione vocale dell'utente: questi due elementi variano ad ogni richiesta e non vengono memorizzati nella cache. La separazione netta tra la parte statica e quella dinamica del contesto è il principio fondamentale che rende efficace il caching: spostare anche solo una parola nel blocco annotato invaliderebbe la cache e ne azzererebbe il beneficio.

L'effetto concreto del meccanismo è visibile nel campo `tokens_input` del blocco `_meta` incluso in ogni risposta: al primo comando della sessione il valore riflette l'intero prompt, poiché la cache non è ancora popolata; a partire dal secondo comando, il valore scende drasticamente poiché all'API vengono trasmessi soltanto il secondo blocco — stato di sessione e trascrizione dell'utente — mentre la porzione statica viene recuperata dalla cache senza essere riconteggiata tra i token in ingresso fatturati. I dati concreti osservati durante le sessioni di test sono discussi nelle Sezioni 5.6 e 6.4.

5.4.5 L'endpoint principale

L'endpoint `POST /api/v1/comando-vocale` rappresenta il punto di ingresso per tutte le richieste provenienti dall'applicazione Flutter. La sua implementazione, riportata nel Listato 5.16, gestisce l'intero ciclo di vita di una richiesta: validazione dell'input, composizione del prompt, chiamata alle API di Claude, estrazione e validazione del file JSON di risposta, e restituzione del risultato al client.

```
app.post('/api/v1/comando-vocale', async (req, res) => {
  const inizio = Date.now();
  try {
    const { trascrizione, liste_riferimento,
            stato_sessione } = req.body;

    if (!trascrizione
        || typeof trascrizione !== 'string'
        || trascrizione.trim().length === 0) {
      return res.status(400).json({
        errore: 'Il campo "trascrizione" e\' obbligatorio.
  '
      });
    }

    const trascrizionePulita = trascrizione.trim();
```

```
    // Ottimizzazione: appiattimento della struttura dati
    const datiOriginali =
      liste_riferimento && liste_riferimento.
   struttura_dati
        ? liste_riferimento.struttura_dati
        : liste_riferimento;

    let contextPerClaude = 'Nessun dato fornito.';
    if (datiOriginali) {
      const flat = appiattisciStruttura(datiOriginali);
      contextPerClaude = JSON.stringify(flat, null, 2);
    }

    // Composizione del messaggio con caching
    const messaggioUser = {
      role: 'user',
      content: [
        {
          type: 'text',
          text: `ISTRUZIONI DI SISTEMA E LISTE:\n`
            + `${SYSTEM_PROMPT_TEMPLATE}\n\n`
            + `DATI DI CONTESTO:\n${contextPerClaude}`,
          cache_control: { type: 'ephemeral' }
        },
        {
          type: 'text',
          text: `STATO SESSIONE: `
            + `${JSON.stringify(stato_sessione || {})}\n\n
   `
            + `COMANDO UTENTE: "${trascrizionePulita}"`
        }
      ]
    };

    const rispostaClaude =
      await anthropic.messages.create({
        model: CLAUDE_MODEL,
        max_tokens: MAX_TOKENS,
        messages: [messaggioUser],
      });

    const testoGrezzo = rispostaClaude.content[0].text;
    let jsonRisposta;
    try {
      jsonRisposta = estraiJson(testoGrezzo);
    } catch (erroreEstrazione) {
      return res.status(502).json({
        risposta_vocale: 'Si e\' verificato un errore.
   Riprova.',
```

```
        dati_operazione: {
          action: 'errore_comprensione', parameters: {}
        },
      });
    }

    if (!validaStrutturaRisposta(jsonRisposta)) {
      return res.status(502).json({
        risposta_vocale: 'Comando compreso parzialmente.',
        dati_operazione: {
          action: 'errore_comprensione', parameters: {}
        },
      });
    }

    const durata = Date.now() - inizio;
    jsonRisposta._meta = {
      modello: CLAUDE_MODEL,
      durata_ms: durata,
      tokens_input:
        rispostaClaude.usage?.input_tokens || null,
      tokens_output:
        rispostaClaude.usage?.output_tokens || null,
    };
    return res.json(jsonRisposta);

  } catch (errore) {
    const durata = Date.now() - inizio;
    if (errore.status === 429) {
      return res.status(429).json({
        risposta_vocale: 'Servizio sovraccarico. Riprova.'
 ,
        dati_operazione: {
          action: 'errore_comprensione', parameters: {}
        },
      });
    }
    if (errore.status === 401) {
      return res.status(500).json({
        risposta_vocale: 'Errore di configurazione.',
        dati_operazione: {
          action: 'errore_comprensione', parameters: {}
        },
      });
    }
    return res.status(500).json({
      risposta_vocale: 'Errore imprevisto. Riprova.',
      dati_operazione: {
        action: 'errore_comprensione', parameters: {}
      },
```

```
    });
  }
});
```

Listato 5.16: Endpoint principale del server middleware aggiornato.

L'endpoint aggiornato introduce tre novità rispetto alla versione iniziale. La prima è la lettura del campo opzionale `stato_sessione` dal corpo della richiesta: questo oggetto, prodotto dal modello nella risposta precedente e conservato dal widget Flutter, viene passato al modello come parte del secondo blocco di contenuto, consentendo la risoluzione dei riferimenti relativi descritta nella Sezione 4.3.3.

La seconda novità è l'adozione del *prompt caching*: invece di comporre un'unica stringa di testo con il prompt completo, la richiesta viene strutturata come un array di blocchi di contenuto. Il primo blocco, contenente le istruzioni di sistema e il contesto appiattito, è annotato con `cache_control: { type: 'ephemeral' }`, che segnala all'API di Anthropic di congelare questa porzione del prompt in cache per le richieste successive provenienti dallo stesso client entro una finestra temporale definita. Poiché il prompt di sistema è identico per tutti i comandi dell'autista nel corso della sessione, il caching elimina la necessità di rielaborarlo integralmente ad ogni chiamata, riducendo sia la latenza sia il costo in token. Il secondo blocco, contenente lo stato di sessione e la trascrizione dell'utente, varia ad ogni richiesta e non viene memorizzato nella cache.

La terza novità è la gestione esplicita dell'errore HTTP 401 restituito dall'API di Anthropic in caso di chiave non valida o scaduta: l'errore viene registrato nei log del server e restituito al client come codice 500, evitando che un problema di configurazione si propaghi silenziosamente all'utente finale senza lasciare traccia diagnostica.

Un aspetto rilevante dell'implementazione è che ogni risposta di errore mantiene la stessa struttura JSON attesa dal dispatcher Flutter, con un campo `risposta_vocale` leggibile dall'utente e un'azione `errore_comprensione`. Questo garantisce che l'applicazione mobile non debba mai gestire formati di risposta diversi in base al tipo di errore: il dispatcher riceve sempre un file JSON conforme e può far leggere il messaggio vocale all'utente indipendentemente dall'esito della richiesta.

Il campo `_meta`, aggiunto alla risposta prima della restituzione al client, contiene metadati informativi utili per il debug e il monitoraggio; tali metadati riguardano il modello utilizzato, la durata della richiesta in millisecondi e il numero di token consumati in input e in output. Questo campo può essere ignorato dalla logica del dispatcher.

5.4.6 L'endpoint di health check e l'avvio del server

Il server espone, inoltre, un endpoint di controllo `GET /api/v1/health` che restituisce lo stato del servizio e può essere utilizzato da script di monitoraggio per verificare che il middleware sia attivo e raggiungibile. Il codice di avvio del server, riportato nel Listato 5.17, chiude il file `server.js`.

```
app.get('/api/v1/health', (req, res) => {
  res.json({
    status: 'ok',
    timestamp: new Date().toISOString(),
    modello: CLAUDE_MODEL,
  });
});

app.use((req, res) => {
```

```
  res.status(404).json({
    errore: `Rotta ${req.method} ${req.path} `
      + `non trovata.`
  });
});

app.listen(PORT, () => {
  console.log('='.repeat(50));
  console.log(` MyDelivery`);
  console.log(` In ascolto sulla porta ${PORT}`);
  console.log(` Modello: ${CLAUDE_MODEL}`);
  console.log('='.repeat(50));
});
```

Listato 5.17: Endpoint di health check e avvio del server.

5.5 Avvio del server e verifica del funzionamento

Una volta completata la configurazione di tutti i file, il server è stato avviato per la prima volta con il comando `node server.js`. L'output prodotto dalla console, riportato nel Listato 5.18, conferma il corretto caricamento delle variabili d'ambiente.

```
aiuser@my-ai:~/mydelivery$ node server.js
[dotenv@17.3.1] injecting env (4) from .env
==================================================
  MyDelivery
  In ascolto sulla porta 3000
  Modello: claude-haiku-4-5-20251001
==================================================
```

Listato 5.18: Output della console al primo avvio del server.

L'output conferma che il file `.env` è stato caricato correttamente con tutte e quattro le variabili d'ambiente, che il server è in ascolto sulla porta 3000 e che il modello configurato è Claude Haiku 4.5 nella versione del 1 ottobre 2025.

5.5.1 Avvio con PM2 per l'esecuzione persistente

In alternativa all'avvio diretto tramite `node server.js`, che mantiene il processo attivo solo per la durata della sessione SSH, è possibile registrare il server in PM2 per garantirne la persistenza, come mostrato nel Listato 5.19.

```
pm2 start server.js --name "mydelivery"
pm2 save
pm2 startup
```

Listato 5.19: Registrazione del server in PM2.

Il comando `pm2 start` avvia il processo e lo registra con il nome specificato. Il comando `pm2 save` salva la lista dei processi attivi, in modo che PM2 possa ripristinarli dopo un riavvio del sistema. Il comando `pm2 startup` genera e installa uno script di avvio automatico, garantendo che PM2 stesso venga lanciato al boot del sistema operativo. Vale la pena notare

che l'avvio tramite PM2, pur garantendo la persistenza del processo, non visualizza i log direttamente nel terminale corrente. Nella fase di test è stato quindi preferito l'avvio diretto con `node server.js` per mantenere visibili in tempo reale i log delle richieste e le azioni identificate, informazioni indispensabili per la validazione del sistema.

5.5.2 Configurazione del firewall

Per garantire che le porte necessarie fossero raggiungibili, è stata verificata la configurazione del firewall UFW (*Uncomplicated Firewall*), come mostrato nel Listato 5.20.

```
sudo ufw allow 54022/tcp
sudo ufw status
```

Listato 5.20: Verifica e configurazione del firewall.

La porta 54022 è la porta SSH per l'amministrazione remota della macchina. La porta 3000, su cui ascolta il server Node.js, non viene esposta direttamente a Internet: in un ambiente di test locale, come quello utilizzato durante il tirocinio, le richieste transitano internamente alla macchina stessa tramite `localhost`, mentre in un ambiente di produzione il traffico transiterebbe attraverso un reverse proxy sulla porta 443, e la porta 3000 resterebbe accessibile solo localmente.

5.6 Test di funzionamento sulla macchina server

I test sono stati effettuati su due fronti distinti. Il primo, tramite il comando `curl` da un secondo terminale SSH collegato alla stessa macchina, ha permesso di verificare il comportamento del sistema con richieste complete includendo le liste di riferimento. Il secondo, tramite uno script di test interattivo, ha consentito di simulare in modo più agevole il flusso operativo reale, inserendo le trascrizioni vocali in modo diretto e ripetibile.

5.6.1 Test dell'endpoint di health check

Il primo test eseguito verifica che il server sia attivo e che l'endpoint di controllo risponda correttamente, come mostrato nel Listato 5.21.

```
curl http://localhost:3000/api/v1/health
```

Listato 5.21: Test dell'endpoint di health check.

La risposta ottenuta, riportata nel Listato 5.22, conferma il corretto funzionamento.

```
{"status":"ok","timestamp":"2026-03-30T16:58:04.045Z",
"modello":"claude-haiku-4-5-20251001"}
```

Listato 5.22: Risposta dell'endpoint di health check.

5.6.2 Primo test del comando vocale e risoluzione dell'errore sul nome del modello

Il primo test del comando vocale completo ha rivelato una criticità legata all'identificativo del modello. La trascrizione è stata inviata correttamente al server, come mostrato nel Listato 5.23, ma questo ha restituito una risposta di errore.

```
curl -X POST http://localhost:3000/api/v1/comando-vocale \
  -H "Content-Type: application/json" \
  -d '{
    "trascrizione": "Mostrami i giri da fare",
    "liste_riferimento": {
      "clienti": ["Alimentari Freschi", "Bar Roma"],
      "giri": ["Firenze Centro", "Prato Nord"],
      "metodi_pagamento": ["CONT","BANC","CARD","ASSE"],
      "articoli_cauzioni": [
        "Pedane", "Contenitori refrigerati"
      ]
    }
  }'
```

Listato 5.23: Primo test del comando vocale completo.

Nel terminale in cui il server era in esecuzione è apparso il messaggio di errore riportato nel Listato 5.24, che ha permesso di identificare immediatamente il problema.

```
[2026-03-30T16:58:49.763Z] Richiesta:
  "Mostrami i giri da fare"
[ERRORE] (412ms): 404 {"type":"error","error":
  {"type":"not_found_error",
  "message":"model: claude-4-5-haiku-20251001"},
  "request_id":"req_011CZZcvt5n97SokhSDgtGSJ"}
```

Listato 5.24: Log di errore nel terminale del server.

L'errore 404 restituito da Anthropic indica che il nome del modello configurato nel file `.env` non corrispondeva a nessun modello disponibile. L'identificativo utilizzato inizialmente era `claude-4-5-haiku-20251001`, ma il formato corretto atteso dall'API di Anthropic antepone il nome della variante al numero di versione, producendo `claude-haiku-4-5-20251001`. La correzione è stata applicata sia nel file `.env` che nel valore di default nel codice di `server.js`, e il server è stato riavviato.

5.6.3 Test dopo la correzione del nome del modello

Dopo aver aggiornato il nome del modello e riavviato il server, il medesimo comando ha prodotto la risposta corretta, riportata nel Listato 5.25.

```
{"risposta_vocale":"Filtro i giri di consegna
  mostrando solo quelli da fare.",
  "dati_operazione":{"action":"filtra_giri_consegna",
  "parameters":{"stato":"da_fare"}},
  "nuovo_stato":{"citta_selezionata":null,
  "giro_selezionato":null,
  "ultimo_cliente_citato":null},
  "_meta":{"modello":"claude-haiku-4-5-20251001",
  "durata_ms":1164,"tokens_input":3424,
  "tokens_output":88}}
```

Listato 5.25: Risposta corretta dopo la correzione del modello.

Il sistema ha identificato correttamente l'intento di filtraggio, ha prodotto lo stato `da_fare` e ha restituito l'oggetto `nuovo_stato` con tutti i campi a `null`, poiché nessuna città né nessun cliente sono stati nominati in questo comando. L'elaborazione si è completata in 1164 millisecondi consumando 3424 token in input e 88 in output.

5.6.4 Test dell'incasso con pagamento multiplo

Un secondo test è stato eseguito per verificare la gestione degli incassi con ripartizione su più metodi di pagamento, uno dei casi strutturalmente più complessi per il sistema. La risposta del server, riportata nel Listato 5.26, mostra la corretta scomposizione del comando in due oggetti pagamento distinti.


```
{"risposta_vocale":"Sto per registrare un incasso
  diviso in 150 euro in contanti e 50 euro in assegni.",
  "dati_operazione":{"action":"registra_incasso_documento"
   ,
  "parameters":{"pagamenti":[
    {"metodo_pagamento":"CONT","importo":150},
    {"metodo_pagamento":"ASSE","importo":50}],
  "annotazioni":""}},
  "nuovo_stato":{"citta_selezionata":"Firenze",
  "giro_selezionato":"Firenze Centro",
  "ultimo_cliente_citato":"Alimentari Freschi"},
  "_meta":{"modello":"claude-haiku-4-5-20251001",
  "durata_ms":1431,"tokens_input":3439,
  "tokens_output":153}}
```


Listato 5.26: Risposta del server per il comando di incasso multiplo.

Il modello ha correttamente normalizzato "contanti" in `CONT` e "assegno" in `ASSE`, completando l'elaborazione in 1431 millisecondi.

5.6.5 Test del fuzzy matching su nome cliente impreciso

Il terzo test verifica il meccanismo di fuzzy matching con una trascrizione che usa una forma abbreviata del nome di un cliente presente nelle liste. La risposta, riportata nel Listato 5.27, mostra la corretta associazione semantica.


```
{"risposta_vocale":"Sto per segnare la consegna per
  Alimentari Freschi come effettuata a nome di Marco.",
  "dati_operazione":{"action":"gestisci_consegna",
  "parameters":{"cliente":"Alimentari Freschi",
  "esito":"Consegna Effettuata",
  "referente":"Marco","annotazioni":""}},
  "nuovo_stato":{"citta_selezionata":"Firenze",
  "giro_selezionato":"Firenze Centro",
  "ultimo_cliente_citato":"Alimentari Freschi"},
  "_meta":{"modello":"claude-haiku-4-5-20251001",
  "durata_ms":2929,"tokens_input":3436,
  "tokens_output":131}}
```


Listato 5.27: Risposta del server per il test di fuzzy matching.

Il modello ha associato correttamente “gli alimentari” a “Alimentari Freschi”, la stringa esatta presente nella lista, estraendo il referente “Marco” e lasciando vuoto il campo annotazioni. Lo stesso comando, ripetuto in una sessione successiva, ha prodotto un risultato identico con una durata di 1644 millisecondi, confermando la stabilità del comportamento.

5.6.6 Lo script di test interattivo

Per semplificare la fase di validazione ed evitare di riscrivere manualmente il corpo della richiesta HTTP ad ogni comando, è stato creato lo script Bash interattivo `test.sh`, riportato nel Listato 5.28, che mantiene le liste di riferimento fisse per l’intera sessione e accetta la trascrizione come input da tastiera.

```
#!/bin/bash
LISTE='{
  "struttura_dati": {
    "citta": [
      {
        "nome": "Firenze",
        "giri": [
          {
            "nome": "Firenze Centro",
            "clienti": [
              {
                "nome": "Alimentari Freschi",
                "da_incassare": 150.00,
                "articoli_cauzioni": ["Pedane"]
              },
              {
                "nome": "Supermercato Centrale",
                "da_incassare": 75.50,
                "articoli_cauzioni": []
              },
              {
                "nome": "Ristorante La Tavola",
                "da_incassare": 220.00,
                "articoli_cauzioni": []
              }
            ]
          }
        ]
      },
      { "nome": "Firenze Sud", "giri": [] },
      { "nome": "Prato",
        "giri": [
          { "nome": "Prato Nord", "clienti": [] },
          { "nome": "Prato Ovest", "clienti": [] }
        ]
      },
      { "nome": "Pistoia",
        "giri": [
          { "nome": "Pistoia Centro", "clienti": [] }
```

```
          ]
        },
        { "nome": "Arezzo",
          "giri": [
            { "nome": "Arezzo Nord", "clienti": [] }
          ]
        },
        { "nome": "Siena",
          "giri": [
            { "nome": "Siena Est", "clienti": [] }
          ]
        }
      ],
      "metodi_pagamento": ["CONT","BANC","CARD","ASSE"],
      "articoli_cauzioni": [
        "Pedane", "Contenitori refrigerati"
      ]
    }
}'
echo "Scrivi il comando vocale (o 'esci' per uscire):"
while true; do
  read -p "> " COMANDO
  if [ "$COMANDO" = "esci" ]; then
    echo "Fine test."
    break
  fi
  curl -s -X POST \
    http://localhost:3000/api/v1/comando-vocale \
    -H "Content-Type: application/json" \
    -d "{
      \"trascrizione\": \"$COMANDO\",
      \"liste_riferimento\": $LISTE
    }" | python3 -m json.tool
  echo ""
done
```

Listato 5.28: Contenuto del file test.sh.

Lo script viene avviato con il comando `bash test.sh` e mantiene un ciclo interattivo in cui ogni riga digitata viene inviata al middleware come trascrizione. La risposta viene formattata in modo leggibile tramite `python3 -m json.tool`.

Di seguito vengono riportati tre esempi consecutivi tratti da una sessione reale di test, che mostrano contestualmente il funzionamento del meccanismo di caching e della gestione dello stato di sessione. I tre comandi sono stati pronunciati in sequenza all'interno della stessa sessione operativa e la progressione dei valori `tokens_input` nel campo `_meta` evidenzia in modo diretto l'effetto del caching: il primo comando, che popola la cache, consuma l'intero prompt; a partire dal secondo la porzione statica viene recuperata dalla cache e all'API vengono trasmessi soltanto lo stato di sessione aggiornato e la nuova trascrizione dell'utente.

Il Listato 5.29 mostra il primo comando della sessione, una richiesta di visualizzazione di tutti i giri disponibili, mappata correttamente sull'azione `filtra_giri_consegna` con il parametro `stato` impostato a `tutti`. Il valore `tokens_input: 3424` riflette la lunghezza

dell'intero prompt, poiché la cache viene popolata con il primo invio, memorizzando la porzione statica per le richieste successive.

```
> mostrami i giri di consegna
{
  "risposta_vocale": "Ecco i tuoi giri di consegna di
    oggi: Firenze Centro, Firenze Sud, Prato Nord,
    Prato Ovest, Pistoia Centro, Arezzo Nord e Siena
    Est. Quale giro vuoi aprire?",
  "dati_operazione": {
    "action": "filtra_giri_consegna",
    "parameters": {
      "stato": "tutti"
    }
  },
  "nuovo_stato": {
    "citta_selezionata": null,
    "giro_selezionato": null,
    "ultimo_cliente_citato": null
  },
  "_meta": {
    "modello": "claude-haiku-4-5-20251001",
    "durata_ms": 2131,
    "tokens_input": 3424,
    "tokens_output": 162
  }
}
```

Listato 5.29: Sessione di test: primo comando — richiesta dei giri di consegna.

Il Listato 5.30 mostra il secondo comando della stessa sessione, l'apertura del primo giro. Il valore `tokens_input: 59` conferma che la cache è attiva: all'API vengono trasmessi soltanto il nuovo stato di sessione e la trascrizione corrente. Il modello ha tuttavia accesso all'intero contesto grazie alla porzione recuperata dalla cache, come dimostra la risposta corretta che elenca i tre clienti del giro con i rispettivi importi. Il campo `nuovo_stato` aggiornato — con `citta_selezionata: "Firenze"` e `giro_selezionato: "Firenze Centro"` — viene conservato dal widget Flutter e reinviato al middleware nel comando successivo, consentendo la risoluzione di riferimenti relativi come "il primo" o "il prossimo".

```
> apri il primo giro
{
  "risposta_vocale": "Apro il giro Firenze Centro.
    Hai 3 clienti: Alimentari Freschi con 150 euro
    da incassare, Supermercato Centrale con 75,50
    euro da incassare, e Ristorante La Tavola con
    220 euro da incassare.",
  "dati_operazione": {
    "action": "apri_giro_consegna",
    "parameters": {
      "nome_giro": "Firenze Centro"
    }
  },
```

```
  "nuovo_stato": {
    "citta_selezionata": "Firenze",
    "giro_selezionato": "Firenze Centro",
    "ultimo_cliente_citato": null
  },
  "_meta": {
    "modello": "claude-haiku-4-5-20251001",
    "durata_ms": 2262,
    "tokens_input": 59,
    "tokens_output": 202
  }
}
```

Listato 5.30: Sessione di test: secondo comando — apertura del primo giro.

Il Listato 5.31 mostra il terzo comando, la registrazione dell'incasso per il primo cliente. Il valore `tokens_input: 68` è appena superiore al precedente, di soli nove token, per effetto del campo `nuovo_stato` aggiornato che viene allegato al secondo blocco dinamico. Il modello, avendo a disposizione lo stato di sessione che indica come giro corrente "Firenze Centro" e come primo cliente "Alimentari Freschi", risolve correttamente il riferimento relativo "il primo cliente" senza che questa informazione debba essere ritrasmessa integralmente all'API. In assenza del caching, questo stesso scambio avrebbe richiesto oltre tremila token in ingresso ad ogni turno.

```
> segna l'incasso per il primo cliente
{
  "risposta_vocale": "Segno un incasso di 150 euro
    per Alimentari Freschi in contanti?",
  "dati_operazione": {
    "action": "registra_incasso_documento",
    "parameters": {
      "cliente": "Alimentari Freschi",
      "pagamenti": [
        {
          "metodo_pagamento": "CONT",
          "importo": 150
        }
      ],
      "annotazioni": ""
    }
  },
  "nuovo_stato": {
    "citta_selezionata": "Firenze",
    "giro_selezionato": "Firenze Centro",
    "ultimo_cliente_citato": "Alimentari Freschi"
  },
  "_meta": {
    "modello": "claude-haiku-4-5-20251001",
    "durata_ms": 2482,
    "tokens_input": 68,
    "tokens_output": 201
  }
```

```
}
```

Listato 5.31: Sessione di test: terzo comando — registrazione dell'incasso per il primo cliente.

I tre esempi mostrati confermano congiuntamente due proprietà del sistema. La prima è l'efficacia del caching: il consumo di token in ingresso per i comandi successivi al primo si colloca nell'ordine delle decine di unità anziché delle migliaia (scendendo a 59 e 68 token), con una riduzione che, nella sessione mostrata, supera il 98%, poiché la porzione statica viene recuperata dalla cache senza essere riconteggiata. La seconda è la continuità contestuale garantita dal meccanismo di stato: il modello risolve correttamente "il primo giro" e "il primo cliente" grazie al campo `nuovo_stato` che viaggia nel blocco dinamico, senza necessità di rielaborare l'intero contesto.

I comandi di test inseriti tramite lo script replicano deliberatamente lo stile telegrafico che un autista userebbe in situazioni operative reali: frasi brevi, senza formalismi, che contengono solo le informazioni essenziali.

Aspetti degni di nota emersi da questa e dalle sessioni analoghe sono: la conversione automatica di riferimenti ordinali ("il primo" risolto correttamente grazie allo stato di sessione), il recupero autonomo dell'importo da incassare dal campo `dettaglio_clienti` delle liste di riferimento, e la gestione corretta dei campi omessi (il metodo di pagamento viene proposto ma non confermato, in attesa della risposta dell'autista).

5.7 Test di funzionamento su dispositivo mobile

Dopo aver validato il corretto funzionamento del middleware tramite i test sulla macchina server, si è proceduto con la verifica del flusso end-to-end su un dispositivo mobile Android fisico, utilizzando l'applicazione Flutter di esempio contenuta nella cartella `example` del pacchetto `mydelivery_voice_assistant`. Questa fase ha comportato la risoluzione di diverse problematiche legate alla raggiungibilità del server, alla configurazione dei permessi Android e alla compatibilità con il sistema di build del progetto Flutter.

5.7.1 Il problema della raggiungibilità del server

Il primo ostacolo emerso durante la preparazione dei test su dispositivo mobile è stato l'impossibilità di raggiungere il server middleware dalla rete esterna. Tutti i test descritti nella sezione precedente erano stati condotti internamente alla macchina server, utilizzando `curl http://localhost:3000/...`, e di conseguenza non avevano evidenziato questa criticità.

Il tentativo di aprire la porta 3000 nel firewall UFW della macchina ha rivelato che il firewall risultava inattivo (`Status: inactive`), il che escludeva UFW come causa del blocco. La verifica tramite il comando `ss` confermava che il server Node.js era correttamente in ascolto sulla porta 3000 su tutte le interfacce. Nonostante ciò, l'accesso dall'esterno tramite browser all'indirizzo `http://80.211.124.113:3000/api/v1/health` non produceva alcuna risposta.

L'analisi del log di primo accesso SSH ha fornito la spiegazione: l'interfaccia di rete della macchina presenta l'indirizzo IP privato `192.168.114.198`, non l'indirizzo pubblico `80.211.124.113`. Questo indica che la macchina è posizionata dietro un NAT gestito dal provider di hosting, che ha configurato il port forwarding esclusivamente per la porta SSH 54022. La porta 3000, pur essendo aperta sulla macchina, non risulta raggiungibile dall'esterno perché il provider non inoltra il traffico su quella porta.

5.7.2 La soluzione tramite tunneling con ngrok

Per superare il blocco imposto dalla configurazione di rete del provider senza dover richiedere modifiche all'infrastruttura, si è adottata una soluzione basata su *tunneling inverso* tramite ngrok, uno strumento che crea un ponte sicuro tra una porta locale del server e un URL pubblico su Internet. Poiché la connessione viene iniziata dall'interno del server verso l'esterno, il tunnel bypassa completamente il firewall e il NAT del provider, senza richiedere l'apertura di alcuna porta aggiuntiva.

La scelta di ngrok è stata effettuata dopo aver testato anche un'alternativa, Localtunnel, che si è rivelata inadatta per l'uso con l'applicazione Flutter. Pur avendo dimostrato che il tunnel funzionava correttamente (il server rispondeva alle richieste provenienti dall'URL pubblico generato), Localtunnel presentava una schermata di sicurezza (*Bypass Warning*) che richiedeva un'interazione manuale nel browser per sbloccare l'accesso. Questa schermata intercettava le chiamate HTTP automatiche dell'applicazione Flutter, restituendo una pagina HTML anziché il file JSON atteso e producendo un errore nel dispatcher. L'installazione di ngrok sulla macchina server è stata effettuata tramite il repository ufficiale, come mostrato nel Listato 5.32.

```
curl -s https://ngrok-agent.s3.amazonaws.com/ngrok.asc \
  | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null
echo "deb https://ngrok-agent.s3.amazonaws.com \
  buster main" \
  | sudo tee /etc/apt/sources.list.d/ngrok.list
sudo apt update
sudo apt install ngrok
```

Listato 5.32: Installazione di ngrok sulla macchina server.

Dopo aver configurato il token di autenticazione fornito dal servizio gratuito di ngrok, il tunnel è stato avviato puntando alla porta 3000 del server Node.js con il comando `ngrok http 3000`. La dashboard di ngrok, riportata nel Listato 5.33, ha mostrato l'URL pubblico generato e le connessioni in tempo reale.

```
Session Status    online
Account           ftvs2801@gmail.com (Plan: Free)
Version           3.37.3
Region            Europe (eu)
Latency           13ms
Forwarding        https://proscholastic-
  pseudoreformatory-annemarie.ngrok-free.dev
  -> http://localhost:3000

HTTP Requests
-------------
11:11:23.998 UTC POST /api/v1/comando-vocale 200 OK
11:11:01.374 UTC POST /api/v1/comando-vocale 200 OK
```

Listato 5.33: Dashboard di ngrok con il tunnel attivo.

5.7.3 Preparazione del progetto Flutter di esempio

L'applicazione di esempio contenuta nella cartella `example` del pacchetto `mydelivery_voice_assistant` è composta esclusivamente dal codice sorgente Dart e non costituisce

un progetto Flutter completo. È stato, quindi, necessario creare la struttura del progetto attorno ad essa. Sul PC di sviluppo è stato creato un nuovo progetto Flutter tramite il comando `flutter create mydelivery_example`, e la cartella del pacchetto vocale è stata posizionata accanto al progetto appena creato.

Il file `pubspec.yaml` del progetto di esempio è stato modificato per referenziare il pacchetto vocale come dipendenza locale tramite percorso relativo. Il file `main.dart` del progetto è stato sostituito con quello contenuto nella cartella `example/lib/` del pacchetto vocale, che include il widget `VoiceAssistantButton`, il dispatcher di tutte le quattordici azioni e la visualizzazione dei risultati sullo schermo. L'URL del middleware è stato aggiornato con l'indirizzo fornito da ngrok.

5.7.4 Configurazione dei permessi Android

Per consentire all'applicazione di utilizzare il microfono e di comunicare con il server middleware, sono state apportate tre modifiche alla configurazione Android del progetto. La prima ha riguardato il file `AndroidManifest.xml`, al quale sono stati aggiunti i permessi per la registrazione audio e l'accesso a Internet. La seconda ha riguardato la configurazione della sicurezza di rete: poiché l'URL del middleware in fase di test utilizzava il protocollo HTTP, e Android 9 e versioni successive bloccano per impostazione predefinita il traffico HTTP non cifrato, è stato necessario creare un file `network_security_config.xml` che abilitasse esplicitamente il traffico in chiaro verso l'indirizzo del server.

La terza modifica ha riguardato la versione minima dell'SDK Android. Il pacchetto `speech_to_text` richiede una versione minima pari a 21. Il progetto Flutter generato utilizzava il file di build in formato Kotlin DSL (`build.gradle.kts`), che richiede una sintassi differente rispetto al formato Groovy tradizionale. Il tentativo iniziale di utilizzare la sintassi Groovy `minSdkVersion 21` ha prodotto un errore di compilazione, risolto sostituendo la riga con la sintassi corretta per il formato Kotlin DSL:

```
minSdk = 21
```

Listato 5.34: Sintassi corretta per il file `build.gradle.kts`.

5.7.5 Test riuscito con ngrok

Dopo aver sostituito l'URL nell'applicazione Flutter con quello generato da ngrok e aver ricompilato l'applicazione, il test sul dispositivo mobile ha prodotto i risultati attesi. I log della dashboard di ngrok hanno confermato la corretta ricezione ed elaborazione delle richieste provenienti dal dispositivo mobile, con codice di risposta HTTP 200 per entrambe le chiamate registrate.

Sul dispositivo mobile, l'applicazione ha visualizzato correttamente tutti gli elementi previsti: la trascrizione riconosciuta dal motore vocale, la risposta vocale restituita dal server, l'azione identificata e i relativi parametri. Il pulsante del microfono ha attraversato correttamente tutti gli stati della macchina a stati descritta nel pacchetto Flutter: blu in attesa (`idle`), rosso durante l'ascolto (`listening`), arancione durante l'elaborazione (`processing`), verde durante la lettura della risposta (`speaking`), e ritorno al blu al termine del ciclo.

5.8 Il pacchetto Flutter per l'acquisizione vocale

Parallelamente allo sviluppo del middleware server, nell'ambito del tirocinio è stato realizzato un pacchetto Flutter autocontenuto, denominato `mydelivery_voice_assistant`,

che i programmatori di Infoservice potranno integrare nell'applicazione MyDelivery. Il pacchetto è stato progettato come un componente isolato e indipendente dal codice sorgente dell'applicazione esistente: è sufficiente importarlo come dipendenza, inserire il widget del microfono nelle schermate desiderate, fornire le liste di riferimento estratte dal database SQLite locale e implementare un dispatcher per tradurre le azioni JSON in operazioni native.

5.8.1 La struttura del pacchetto

Il pacchetto è organizzato in cinque aree funzionali distinte:

- I modelli dati, che definiscono la struttura della richiesta HTTP (`VoiceCommandRequest` con la relativa classe `ReferenceData`), della risposta JSON (`VoiceCommandResponse` e `OperationData`) e degli stati del widget (`AssistantState`).
- I servizi, che incapsulano la logica di riconoscimento vocale (`SpeechRecognitionService`), di sintesi vocale (`TextToSpeechService`) e di comunicazione HTTP con il middleware (`MiddlewareClient`).
- Il widget principale (`VoiceAssistantButton`), che orchestra l'intero flusso e fornisce il feedback visivo all'autista.
- La configurazione (`AssistantConfig`), che consente di personalizzare l'URL del server, il timeout, la lingua e la velocità della sintesi vocale.
- La classe di utilità `DataFlattener`, che espone il metodo statico `flatten()` per trasformare la struttura gerarchica dei dati in liste piatte compatibili con il formato atteso dal middleware, e il metodo `flattenToEncodedJson()` per serializzare direttamente il risultato in stringa JSON.

5.8.2 Il flusso interno del widget

Il widget del microfono implementa una macchina a stati con cinque stati distinti, ciascuno associato a un feedback visivo specifico:

- `idle`: il pulsante è blu con l'icona del microfono, in attesa della pressione;
- `listening`: il pulsante diventa rosso e pulsa con un'animazione, indicando che il sistema sta ascoltando la voce dell'autista;
- `processing`: il pulsante diventa arancione e mostra un indicatore di caricamento circolare, segnalando che la trascrizione è stata inviata al server;
- `speaking`: il pulsante diventa verde con l'icona dell'altoparlante, indicando che il sistema sta leggendo ad alta voce la risposta tramite sintesi vocale;
- `error`: il pulsante diventa grigio con l'icona di errore e torna automaticamente allo stato `idle` dopo tre secondi.

Il flusso operativo completo, dalla pressione del pulsante alla ricezione dell'azione, si articola nei seguenti passaggi:

1. L'autista preme il pulsante del microfono e il widget passa allo stato `listening`.
2. Il servizio di riconoscimento vocale si attiva e trascrive l'audio in tempo reale, aggiornando il callback `onTranscription` ad ogni risultato parziale.

3. Quando il motore di speech-to-text rileva una pausa nella voce dell'utente, il risultato viene considerato finale e l'ascolto si interrompe.

4. Il widget passa allo stato `processing` e il client HTTP invia la trascrizione al middleware, impacchettandola con le liste di riferimento e con l'oggetto `_sessionState` conservato in memoria dal widget a partire dalla risposta precedente.

5. Alla ricezione della risposta, il widget aggiorna `_sessionState` con il valore del campo `nuovoStato` restituito dal modello. Se il TTS è abilitato, il widget passa allo stato `speaking` e legge ad alta voce il campo `risposta_vocale`.

6. Il callback `onActionReceived` viene invocato con l'oggetto `VoiceCommandResponse` completo.

7. Il widget torna allo stato `idle`, pronto per il comando successivo.

5.8.3 Il contratto di integrazione

L'utilizzo del pacchetto richiede tre attività:

1. L'importazione del pacchetto come dipendenza nel `pubspec.yaml` dell'applicazione principale e la configurazione dei permessi per il microfono nei manifest di Android e iOS.

2. L'inserimento del widget `VoiceAssistantButton` nelle schermate dell'applicazione, passando ad esse la configurazione con l'URL del middleware e le liste di riferimento estratte dal database SQLite locale.

3. L'implementazione della logica di dispatch nel callback `onActionReceived`, che riceve l'oggetto `VoiceCommandResponse` e, tramite uno switch sul campo `action`, esegue le operazioni native corrispondenti.

Un'applicazione di esempio completa, contenuta nella cartella `example` del pacchetto, illustra concretamente come svolgere ciascuna di queste attività e include uno scheletro di dispatcher con tutte e quattordici le azioni già predisposte.

5.9 Considerazioni sull'ambiente di test

L'ambiente descritto in questo capitolo è stato configurato con l'obiettivo specifico di validare il funzionamento del flusso end-to-end, dalla ricezione della trascrizione vocale alla produzione del file JSON strutturato. Si tratta di un ambiente di test e non di un ambiente di produzione; alcune misure di sicurezza e di affidabilità che sarebbero necessarie per un deployment completo non sono state implementate in questa fase, poiché il loro sviluppo è a carico del team di Infoservice.

L'utilizzo di ngrok per il tunneling, adottato per superare le limitazioni di rete del provider di hosting durante la fase di test, rappresenta una soluzione temporanea adeguata allo sviluppo e alla validazione, ma non idonea per un ambiente di produzione. Gli URL generati da ngrok nel piano gratuito cambiano ad ogni riavvio del tunnel, rendendo necessario aggiornare la configurazione dell'applicazione Flutter ad ogni sessione di test. Per la messa in produzione, l'azienda dovrà configurare l'infrastruttura di rete in modo che il server sia raggiungibile in modo stabile, attraverso l'apertura delle porte necessarie da parte del provider o la configurazione di un reverse proxy con dominio dedicato.

Tra gli ulteriori aspetti che il team di sviluppo dovrà affrontare per la messa in produzione figurano la configurazione del reverse proxy HTTPS con certificati TLS, l'implementazione di un meccanismo di autenticazione tra l'applicazione Flutter e il middleware tramite token Bearer, la configurazione della rotazione dei log e il monitoraggio dei costi delle chiamate API. Questi aspetti, pur esulando dal perimetro del tirocinio, sono stati identificati e documentati durante la fase di progettazione affinché il team disponga di tutte le informazioni necessarie per completare l'integrazione.

5.10 L'implementazione del server locale con Ollama

In parallelo allo sviluppo del middleware principale, è stato predisposto sulla stessa macchina server Ubuntu un secondo server Node.js denominato `server-local.js`, progettato per sostituire le chiamate alle API di Anthropic con un modello linguistico locale gestito da Ollama. Questo componente, pur non facendo parte del sistema consegnato in produzione, è stato sviluppato e testato con rigore per valutare concretamente la fattibilità dell'architettura locale descritta nel capitolo precedente e per produrre dati quantitativi comparabili con quelli del sistema principale.

5.10.1 Installazione di Ollama e download del modello

L'installazione di Ollama sulla macchina server è avvenuta tramite lo script ufficiale, seguito dal download del modello Qwen2.5-1.5B in formato GGUF quantizzato:

```
curl -fsSL https://ollama.ai/install.sh | sh
ollama --version

# Download del modello da 1.5B parametri (quantizzazione
   Q4)
ollama pull qwen2.5:1.5b

# Verifica del funzionamento con un test minimo
ollama run qwen2.5:1.5b "Rispondi solo con JSON: {\"test\"
   : true}"
```

Listato 5.35: Installazione di Ollama e download del modello sulla macchina server.

Il download del modello ha occupato circa un gigabyte di spazio su disco. Il servizio Ollama espone un'API HTTP sulla porta 11434 e gestisce automaticamente il caricamento del modello in memoria, la sua persistenza tra una richiesta e l'altra e il rilascio delle risorse dopo un periodo di inattività. La verifica iniziale con un prompt minimalista ha prodotto una risposta in cinque-sei secondi, un dato che già lasciava presagire le problematiche di latenza che sarebbero emerse con prompt più complessi.

5.10.2 La struttura del server gemello

Il file `server-local.js` è strutturalmente identico a `server.js` in ogni sua parte: stesso prompt di sistema, stesso endpoint `POST /api/v1/comando-vocale`, stessa logica di validazione e parsing del file JSON con le tre strategie progressive descritte nel capitolo precedente. L'unica differenza funzionale è la funzione `chiamaModelloLocale()`, che sostituisce la chiamata all'SDK di Anthropic con una richiesta HTTP verso l'API di Ollama, e la porta di ascolto, impostata a 3001 per consentire l'esecuzione parallela dei due server sulla stessa macchina durante la fase di test comparativo.

```
async function chiamaModelloLocale(promptCompleto) {
  const response = await fetch(`${OLLAMA_URL}/api/generate
   `, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      model: LOCAL_MODEL,
      prompt: promptCompleto,
      stream: false,
      options: {
        temperature: 0,
        top_p: 1,
        top_k: 1,
        num_predict: 500,
        repeat_penalty: 1.1,
        stop: ['\nInput:', '\n\nInput:', 'COMANDO:'],
      },
    }),
  });

  if (!response.ok) {
    const errore = await response.text();
    throw new Error(
      `Ollama ha risposto con errore ${response.status}: $
   {errore}`
    );
  }

  const data = await response.json();
  return {
    testo: data.response,
    tokens_input: data.prompt_eval_count || null,
    tokens_output: data.eval_count || null,
    durata_ollama_ms: data.total_duration
      ? Math.round(data.total_duration / 1e6)
      : null,
  };
}
```

Listato 5.36: Funzione di chiamata al modello locale tramite Ollama in server-local.js.

I parametri di generazione sono stati calibrati attraverso le iterazioni di test descritte nel capitolo precedente. `temperature: 0` e `top_k: 1` attivano il *greedy decoding*, in cui il modello seleziona sempre il token con la probabilità più alta senza campionamento stocastico: questo elimina la variabilità tra run successivi sullo stesso input, un problema emerso nelle prime sessioni con valori di temperatura pari a 0.05 e 0.1. Il parametro `repeat_penalty` riduce la tendenza del modello a generare sequenze ripetitive, mentre i token di stop — `\nInput:`, `\n\nInput:` e `COMANDO:` — impediscono che il modello continui a generare testo dopo aver completato il file JSON di risposta, replicando la struttura degli esempi nel prompt invece di fermarsi. Il parametro `num_predict` è stato portato a 500 dopo aver osservato troncamenti nei file JSON con strutture complesse: gli incassi con pagamento

multiplo richiedono circa 120-130 token in output, e un limite inferiore produceva file JSON troncati a metà, come documentato nel log:

```
Raw: {"risposta_vocale":"Ho registrato l'incasso di 150
   euro in
 contanti e 50 euro in assegno.",
 "dati_operazione":{"action":"registra_incasso_documento",
   "p
```

Listato 5.37: Esempio di file JSON troncato per num_predict insufficiente.

5.10.3 Le iterazioni di ottimizzazione del prompt

La fase più laboriosa dell'implementazione locale ha riguardato l'ottimizzazione del prompt per renderlo compatibile con i tempi di elaborazione di un modello su CPU. Il primo test con il template completo da circa 3400 token ha prodotto il log riportato nel Capitolo 6, con un tempo di 74 secondi per un singolo comando. Le quattro iterazioni di affinamento sono state condotte tramite script Node.js dedicati (`test-prompt-corto.js`, `test-prompt-medio.js`, `test-prompt-finale.js` e `test-prompt-v4.js`), ciascuno testato sulla stessa batteria di comandi per consentire il confronto diretto tra le versioni.

Il primo script di test, con il prompt compresso a 600 token e zero esempi, ha prodotto i risultati mostrati nel Listato 5.38:

```
Comando: "mostrami i giri da fare"
Durata: 12718ms | Token input: 611 | Token output: 42
Azione: filtra_giri_consegna [OK]

Comando: "aprimi il giro di firenze"
Durata: 2638ms | Token input: 613 | Token output: 42
Azione: apri_giro_consegna [OK - ma nome_giro: "Firenze"
   invece di
"Firenze Centro"]

Comando: "segna consegna alimentari effettuata ha ritirato
    Marco"
Durata: 3627ms | Token input: 619 | Token output: 59
Azione: gestisci_consegna [SBAGLIATO - cliente: "Marco"
   invece di
"Alimentari Freschi"]

Comando: "aggiungi incasso 150 euro contanti e 50 assegno"
Durata: 4650ms | Token input: 622 | Token output: 76
Azione: registra_incasso_documento [OK - importi e metodi
   corretti]

Comando: "reso di tre pedane"
Durata: 4807ms | Token input: 610 | Token output: 87
Azione: ERRORE_PARSING [JSON doppio malformato]

Comando: "stampa ricevuta"
Durata: 3247ms | Token input: 609 | Token output: 61
```

```
Azione: visualizza_ricevuta [SBAGLIATO - azione inventata]
```

Listato 5.38: Risultati del primo script di test con prompt compresso.

Il secondo script, con il prompt medio da 1100 token e quattro esempi selettivi, ha mostrato un miglioramento apprezzabile sui casi precedentemente sbagliati, portando l'accuratezza al 70%, come mostrato nel Listato 5.39:

```
Comando: "segna consegna alimentari effettuata ha ritirato
    Marco"
Durata: 4656ms | Token in: 1099 | Token out: 74
Azione: gestisci_consegna [OK]
Params: {"cliente":"Alimentari Freschi","esito":"Consegna
   Effettuata",
 "referente":"Marco","annotazioni":""}

Comando: "reso di tre pedane"
Durata: 2972ms | Token in: 1090 | Token out: 53
Azione: registra_resi_cauzioni [OK]
Params: {"articoli_resi":[{"descrizione":"Pedane","
   quantita":3}]}

Comando: "aprimi il giro di firenze"
Durata: 2871ms | Token in: 1093 | Token out: 41
Azione: apri_giro_consegna [QUASI - nome_giro: "Firenze"
   invece di "Firenze Centro"]

Comando: "che tempo fa"
Durata: 3851ms | Token in: 1087 | Token out: 66
Azione: filtra_cronologia [SBAGLIATO - doveva dare
   errore_comprensione]
```

Listato 5.39: Risultati parziali del test con prompt medio.

Il terzo script, con il prompt finale da 1368 token e dieci esempi selettivi, ha raggiunto il 93% su quindici casi con un solo errore tecnico di troncamento, come mostrato nel Listato 5.40:

```
Comando: "aprimi il giro di firenze"
Durata: 2605ms | Token in: 1370 | Token out: 42
Azione: apri_giro_consegna [OK]
Params: {"nome_giro":"Firenze Centro"}

Comando: "incassa tutto in contanti"
Durata: 3121ms | Token in: 1367 | Token out: 53
Azione: errore_comprensione [OK]

Comando: "che tempo fa"
Durata: 3064ms | Token in: 1364 | Token out: 48
Azione: errore_comprensione [OK]

Comando: "aggiungi incasso 150 euro contanti e 50 assegno"
Durata: 5190ms | Token in: 1379 | Token out: 82
```

```
Azione: ERRORE_PARSING [JSON troncato per num_predict
   insufficiente]
```

Listato 5.40: Risultati del test con prompt finale, Versione 1.

Il quarto e ultimo script, con il prompt da 1604 token, `temperature: 0` e `num_predict: 500`, ha testato ventidue casi raggiungendo il 18/22 (82%), come mostrato nel Listato 5.41:

```
RISULTATO: 18/22 = 82%

[OK] "mostrami i giri da fare"          2817ms
   filtra_giri_consegna
[OK] "mostrami solo i giri consegnati"  3108ms
   filtra_giri_consegna
[OK] "filtra giri in consegna"          3134ms
   filtra_giri_consegna
[OK] "aprimi il giro di firenze"        3050ms
   apri_giro_consegna
[OK] "apri prato nord"                  2746ms
   apri_giro_consegna
[OK] "mostrami le consegne sulla mappa" 2922ms
   cambia_vista_documenti
[FAIL] "torna alla lista"               2112ms ->
   errore_comprensione
[OK] "segna consegna alimentari..."     4478ms
   gestisci_consegna
[FAIL] "bar roma non consegnato..."     5808ms ->
   ERRORE_PARSING
[OK] "segna supermercato centrale..."   3597ms
   gestisci_consegna
[OK] "fammi vedere gli articoli"        2845ms
   visualizza_dettagli_documento
[OK] "aggiungi incasso 150 contanti..." 5138ms
   registra_incasso_documento
[FAIL] "150 euro cash"                  3124ms ->
   errore_comprensione
[OK] "incassa 200 euro con carta..."    5742ms
   incassa_partita_aperta
[OK] "reso di tre pedane"               4849ms
   registra_resi_cauzioni
[OK] "registra reso 2 contenitori..."   3374ms
   registra_resi_cauzioni
[OK] "stampa ricevuta"                  2458ms
   stampa_ricevuta_consegna
[OK] "genera riepilogo incassi"         2930ms
   stampa_report_cronologia
[OK] "voglio salvare la mia firma"      2682ms
   impostazioni_firma
[OK] "incassa tutto in contanti"        2955ms
   errore_comprensione
[OK] "che tempo fa"                     2512ms
   errore_comprensione
```

```
[FAIL] "registra incasso ditta Acme"    4368ms ->
   ERRORE_PARSING
```

Listato 5.41: Risultati del test con prompt versione 4 e greedy decoding.

5.10.4 L'avvio parallelo e la configurazione

Il file `.env` è stato esteso con tre variabili aggiuntive per la configurazione del server locale:

```
PORT_LOCAL=3001
OLLAMA_URL=http://localhost:11434
LOCAL_MODEL=qwen2.5:1.5b
```

Listato 5.42: Variabili aggiuntive nel file `.env` per il server locale.

Il server locale è stato avviato in parallelo al server principale tramite PM2, con Ollama avviato come processo in background (Listato 5.43):

```
# Avvia Ollama in background
ollama serve &
sleep 3

# Avvia il server locale sulla porta 3001
pm2 start server-local.js --name "mydelivery-local"
pm2 save

# Verifica dello stato di entrambi i server
curl http://localhost:3000/api/v1/health  # risponde
   Claude Haiku
curl http://localhost:3001/api/v1/health  # risponde Qwen
   locale
```

Listato 5.43: Avvio del server locale e di Ollama tramite PM2.

La risposta dell'endpoint di health check del server locale riporta le informazioni specifiche del backend Ollama, differenziandosi dalla risposta del server principale (Listato 5.44):

```
{
 "status": "ok",
 "timestamp": "2026-04-10T09:00:00.000Z",
 "modello": "qwen2.5:1.5b",
 "backend": "ollama-local",
 "ollama_url": "http://localhost:11434"
}
```

Listato 5.44: Risposta dell'health check del server locale.

CAPITOLO 6

Validazione

In questo capitolo vengono presentati i risultati ottenuti dal sistema middleware durante la fase di test, analizzando il comportamento del modello Claude Haiku 4.5 su diverse categorie di comandi vocali. Si esaminano casi di successo, casi limite e situazioni di errore gestito, valutando la qualità delle risposte JSON prodotte, l'efficacia del fuzzy matching dinamico e le prestazioni misurate. Si documentano i risultati dei test condotti su dispositivo mobile Android fisico, che hanno validato il funzionamento dell'intero flusso end-to-end. Viene, poi, presentata l'analisi completa dell'esperimento con il modello locale basato su Ollama: le iterazioni di ottimizzazione del prompt, il confronto diretto con Claude Haiku 4.5 e la valutazione delle prospettive evolutive, inclusa la discussione sulla fattibilità di un'esecuzione on-device e sulla configurazione ibrida. Si discutono, infine, le criticità emerse e i limiti del sistema, e si delineano i possibili sviluppi futuri.

6.1 Metodologia di test

La validazione del sistema è stata condotta attraverso tre modalità complementari. La prima ha utilizzato il comando `curl` da un secondo terminale SSH collegato alla macchina server, inviando richieste HTTP complete con le liste di riferimento per simulare le chiamate che l'applicazione Flutter invierebbe al middleware in produzione. La seconda ha impiegato lo script interattivo `test.sh` descritto nel capitolo precedente, che ha permesso di inserire trascrizioni in modo diretto e di osservare le risposte in tempo reale con un flusso simile a quello di una sessione operativa reale. La terza ha utilizzato l'applicazione Flutter di esempio installata su un dispositivo mobile Android fisico, connessa al server middleware tramite il tunnel ngrok, per verificare il funzionamento dell'intero flusso end-to-end inclusi il riconoscimento vocale nativo del dispositivo, la comunicazione HTTP con il middleware e la sintesi vocale della risposta.

Nelle prime due modalità, il plugin vocale Flutter e il dispatcher non erano ancora integrati nell'applicazione MyDelivery, poiché il loro completamento è a carico del team di sviluppo interno di Infoservice: i test si sono, quindi, concentrati sul middleware, verificando che l'uscita JSON fosse sempre corretta, strutturalmente conforme e pronta per essere consumata dall'applicazione. La terza modalità ha, invece, validato il funzionamento del pacchetto Flutter `mydelivery_voice_assistant` su un dispositivo reale, confermando che l'intero flusso — dalla voce dell'utente al JSON strutturato visualizzato sullo schermo — opera correttamente.

I test sono stati organizzati in modo da coprire tutte le quattordici azioni operative mappate nel prompt di sistema, verificando, per ciascuna di esse, la corretta identificazione dell'intento, l'accuratezza dell'estrazione dei parametri, il funzionamento del fuzzy matching con le liste di riferimento e la conformità strutturale del file JSON prodotto. Le liste di riferimento utilizzate per i test sono riportate nel Listato 6.1.

```
{
  "clienti": [
    "Alimentari Freschi", "Bar Roma",
    "Supermercato Centrale", "Ristorante La Tavola"
  ],
  "giri": [
    "Firenze Centro", "Prato Nord",
    "Pistoia", "Arezzo"
  ],
  "metodi_pagamento": ["CONT", "BANC", "CARD", "ASSE"],
  "articoli_cauzioni": [
    "Pedane", "Contenitori refrigerati",
    "Fusti birra 30L"
  ]
}
```

Listato 6.1: Liste di riferimento utilizzate durante la fase di test.

6.2 Risultati per categoria operativa

In questa sezione vengono presentati i risultati dei test raggruppati per categoria funzionale, analizzando sia i casi in cui il modello ha prodotto risposte corrette sia quelli in cui si sono manifestate criticità.

6.2.1 Navigazione e visualizzazione

I comandi di navigazione hanno mostrato un tasso di successo molto elevato, poiché si tratta di operazioni con una struttura linguistica relativamente semplice e un numero limitato di parametri da estrarre. Il comando "mostrami solo i giri in consegna" ha prodotto correttamente l'azione `filtra_giri_consegna` con lo stato `in_consegna`. Analogamente, varianti più colloquiali come "fammi vedere quelli da fare" o "fai vedere i giri di consegne" sono state interpretate correttamente, dimostrando la capacità del modello di comprendere formulazioni diverse dello stesso intento.

Il comando "aprimi il giro di firenze" ha attivato correttamente il fuzzy matching, producendo l'azione `apri_giro_consegna` con il parametro `nome_giro` impostato a "Firenze Centro", la stringa esatta presente nella lista di riferimento. Questo risultato conferma l'efficacia del matching semantico delegato al modello: la corrispondenza non si basa sulla somiglianza carattere per carattere ma sulla comprensione del significato.

6.2.2 Gestione operativa delle consegne

I comandi di gestione delle consegne rappresentano uno dei casi d'uso più articolati, poiché richiedono l'estrazione simultanea di più parametri da una singola frase in linguaggio naturale. Il comando "segna la consegna degli alimentari come effettuata, ha ritirato Marco"

è stato scomposto correttamente: il cliente è stato mappato a "Alimentari Freschi" dalla lista tramite corrispondenza semantica su "gli alimentari", l'esito è stato identificato come "Consegna Effettuata" e il referente estratto come "Marco".

Particolarmente rilevante è stato il test con formulazione minima "segna consegna alimentari effettuata", ovvero tre parole chiave senza struttura grammaticale completa. Il modello ha correttamente identificato il cliente tramite fuzzy matching e ha lasciato vuoti i campi referente e annotazioni, senza inventare dati non forniti, comportamento coerente con le istruzioni del prompt.

6.2.3 Gestione amministrativa

La registrazione degli incassi ha rappresentato il banco di prova più impegnativo per il sistema, sia per la complessità strutturale del file JSON da produrre sia per la varietà di formulazioni possibili. Il comando "aggiungi un incasso di 150 euro in contanti e 50 euro con assegno" ha prodotto correttamente un array di due pagamenti con i metodi `CONT` e `ASSE` e gli importi rispettivi. Il comando nella forma informale "metti 50 euro contanti e 150 assegno", con l'ordine invertito rispetto alla formulazione standard, ha prodotto la medesima struttura con gli importi corretti nella posizione giusta, confermando che il modello non esegue un matching posizionale ma comprende la semantica della frase.

I resi di cauzione sono stati testati con il comando "reso di tre pedane": il modello ha convertito il numero espresso in lettere nel valore numerico 3, ha mappato "pedane" a "Pedane" dalla lista di riferimento e ha prodotto l'array corretto in 1373 millisecondi. Un test rilevante ha riguardato il comando "incassa tutto in cash": il modello ha correttamente mappato "cash" al codice `CONT`, ma non disponendo dell'importo esatto ha prodotto un'azione `errore_comprensione` spiegando che l'importo è un dato necessario. Il modello non ha inventato valori: ha segnalato la mancanza del dato.

6.2.4 Reportistica e impostazioni

I comandi di reportistica hanno mostrato risultati uniformemente positivi. Il comando "stampa ricevuta" in forma ridottissima ha prodotto l'azione `stampa_ricevuta_consegna` in 986 millisecondi, il tempo più basso registrato durante i test, coerentemente con la semplicità strutturale del file JSON restituito (nessun parametro).

6.2.5 Gestione degli errori

I test sulla gestione degli errori hanno confermato il comportamento robusto del sistema in condizioni anomale. Il comando "registra incasso per la ditta Acme" (dove "Acme" non è presente in nessuna lista) ha prodotto un'azione `errore_comprensione` con una risposta vocale che spiega chiaramente il problema. Comandi completamente fuori contesto, come "che tempo fa oggi", sono stati gestiti con risposte che spiegano le funzionalità disponibili e invitano l'utente a formulare un comando pertinente. In nessun caso il modello ha tentato di rispondere a domande estranee al dominio operativo dell'applicazione.

6.3 Risultati dei test su dispositivo mobile

I test condotti sull'applicazione Flutter di esempio installata sul dispositivo Android fisico, descritti nel dettaglio nella Sezione 5.7, hanno confermato il corretto funzionamento dell'intero flusso end-to-end. I log della dashboard di ngrok hanno registrato le richieste provenienti dal dispositivo mobile con codice di risposta HTTP 200, confermando che il

middleware ha elaborato correttamente i comandi vocali pronunciati dall'utente e ha restituito file JSON strutturati validi. Il motore di sintesi vocale del dispositivo ha letto ad alta voce le risposte vocali contenute nel campo `risposta_vocale`, e il dispatcher dell'applicazione di esempio ha visualizzato sullo schermo l'azione identificata e i relativi parametri.

La fase di test su dispositivo mobile ha evidenziato alcune problematiche operative che non erano emerse durante i test sulla macchina server. La prima ha riguardato la raggiungibilità del server dall'esterno, risolta tramite il tunneling con ngrok, come descritto nel capitolo precedente. La seconda ha riguardato la scelta dello strumento di tunneling: il primo tentativo con Localtunnel, pur avendo dimostrato che il tunnel funzionava correttamente, si è rivelato inadatto per l'uso con l'applicazione Flutter a causa della schermata di sicurezza che intercettava le chiamate HTTP automatiche. La terza ha riguardato la compatibilità del sistema di build: l'utilizzo della sintassi Groovy nel file Kotlin DSL ha prodotto un errore di compilazione, risolto sostituendo la riga con la sintassi corretta `minSdk = 21`.

6.4 Analisi delle prestazioni

I metadati contenuti nel campo `_meta` delle risposte hanno fornito informazioni quantitative per una valutazione delle prestazioni del sistema basata sui dati reali delle sessioni di test.

6.4.1 Tempi di risposta

I tempi di risposta misurati durante i test si sono collocati in un intervallo compreso tra 986 e 2929 millisecondi, con la maggior parte delle richieste completate tra i 1100 e i 2000 millisecondi. La Tabella 6.1 riepiloga i valori osservati per le principali tipologie di comando.

Comando	Durata (ms)
filtra giri (“mostrami i giri da fare”)	1164
apri giro (“aprimi il giro di firenze”)	1133
richiesta elenco giri con caching attivo	2131
apertura giro con caching attivo	2262
registrazione incasso con caching attivo	2482
incasso multiplo (“150 contanti e 50 assegno”)	1431
incasso informale (“metti 50 contanti e 150 assegno”)	1858
gestisci consegna con referente	2929
gestisci consegna senza referente	2013
reso cauzioni (“tre pedane”)	1373
stampa ricevuta	986

Tabella 6.1: Tempi di risposta misurati durante i test, espressi in millisecondi.

I comandi più semplici, come la stampa della ricevuta o l'apertura di un giro, producono risposte più rapide poiché richiedono un numero ridotto di token in output. I comandi più complessi, come la gestione di una consegna con tutti i campi popolati, richiedono tempi superiori coerentemente con la maggiore quantità di testo da generare. In un contesto operativo

reale, a questi tempi si aggiungerebbe la latenza di rete tra il dispositivo mobile e il server middleware, stimabile in 100–500 millisecondi a seconda della qualità della connessione. Il tempo complessivo dall'emissione del comando vocale alla lettura della risposta si è collocato nell'ordine dei due-quattro secondi, un intervallo accettabile per un'interazione vocale che non richiede risposta in tempo reale.

È opportuno chiarire un aspetto che la Tabella 6.1 potrebbe far interpretare in modo scorretto: le righe relative ai comandi eseguiti con il caching attivo (2131, 2262 e 2482 millisecondi) presentano durate non inferiori, e in alcuni casi superiori, a quelle dei comandi analoghi eseguiti senza caching (1133 e 1164 millisecondi). Questo non costituisce un'anomalia di misurazione, ma riflette una distinzione concettuale importante: il *prompt caching* riduce il numero di token *in ingresso* effettivamente riconteggiati e fatturati dall'API, non necessariamente il tempo di latenza percepito dall'utente. La durata complessiva di una richiesta dipende, infatti, anche da altri fattori non influenzati dal caching, quali il numero di token generati in *output* (la risposta vocale e il payload operativo), il carico istantaneo dell'infrastruttura di Anthropic al momento della chiamata e, potenzialmente, un overhead di elaborazione interno legato al recupero della porzione di prompt dalla cache rispetto alla sua rielaborazione diretta. I dati raccolti in questa tesi non consentono di isolare statisticamente queste componenti, poiché ogni comando è stato misurato in un numero limitato di esecuzioni; il caching va quindi valutato correttamente come un meccanismo di riduzione del costo per token fatturato, non come una garanzia automatica di riduzione della latenza end-to-end. Un'indagine futura con un numero maggiore di campionamenti per ciascun comando sarebbe necessaria per quantificare con rigore statistico l'effettivo impatto del caching sui tempi di risposta.

6.4.2 Consumo di token

Il consumo di token varia significativamente in funzione dell'attivazione del meccanismo di prompt caching introdotto nella versione definitiva del server. Nelle sessioni di test condotte con il caching disabilitato, il consumo di token in input si è mantenuto stabile tra le 3424 e le 3453 unità per ogni richiesta: questa stabilità deriva dal fatto che il prompt di sistema costituisce la parte dominante dell'input e ha una lunghezza fissa, mentre le variazioni dipendono dalla lunghezza della trascrizione e delle liste di riferimento.

Con il caching abilitato, il comportamento cambia radicalmente a partire dal secondo comando della sessione. I dati misurati durante la sessione di test riportata nei Listati 5.29, 5.30 e 5.31 mostrano con precisione l'entità della riduzione: il primo comando della sessione ha consumato 3424 token in ingresso, mentre il secondo ne ha richiesti 59 e il terzo 68. A partire dal secondo comando, questi valori corrispondono esclusivamente alla lunghezza del blocco dinamico — stato di sessione e trascrizione vocale — poiché la porzione statica del prompt, congelata nella cache di Anthropic, non viene riconteggiata tra i token fatturati nelle richieste successive. In questa specifica sessione di tre comandi consecutivi, condotti sullo stesso giro e senza cambio di città o di cliente, il risparmio rispetto all'approccio senza caching supera il 98% per i token in ingresso del secondo e del terzo comando, riducendo il costo effettivo di ogni turno a una frazione trascurabile di quello del primo invio. È importante sottolineare che questo valore percentuale è stato misurato su un'unica sessione di esempio, non su una media statistica condotta su un campione ampio di sessioni operative reali; il beneficio effettivo dipende fortemente dal numero di comandi impartiti consecutivamente all'interno della stessa sessione e dalla frequenza con cui il contesto della sessione cambia, come discusso nella Sezione 6.6.

Il consumo di token in output è rimasto, invece, comparabile tra le due modalità, variando tra le 73 (per azioni senza parametri come la stampa della ricevuta) e le 202 unità (per risposte elaborate come l'elenco dei giri con dettaglio dei clienti), proporzionalmente alla complessità del file JSON e della frase vocale generati.

La Tabella 6.2 confronta il consumo di token osservato nelle due modalità operative per i tre comandi della sessione di test documentata.

Comando	Token input (senza caching)	Token input (con caching)
mostrami i giri di consegna	∼3430	3430
apri il primo giro	∼3440	59
segna l'incasso per il primo cliente	∼3450	68

Tabella 6.2: Confronto del consumo di token in ingresso con e senza prompt caching.

Ipotizzando una flotta di 50 autisti che effettuano ciascuno 30 comandi vocali al giorno, il volume giornaliero ammonterebbe a circa 1500 richieste. Senza caching, il consumo totale si collocherebbe nell'ordine di 5–6 milioni di token in ingresso; con il caching attivo il costo si riduce, ma non nella stessa misura osservata nella singola sessione di esempio discussa sopra. Nella pratica operativa, infatti, un autista cambia frequentemente giro e cliente nel corso della giornata — ad esempio ogni volta che completa una consegna e passa alla successiva — e ciascuno di questi cambiamenti comporta una variazione del contenuto del blocco statico iniettato nel prompt (Sezione 6.6), che invalida la cache e fa tornare il costo del comando successivo al valore pieno di circa 3400 token. Una stima più prudente colloca quindi il risparmio reale a un valore intermedio tra lo scenario senza caching e il caso ideale di sessione interamente cacheable, con un beneficio comunque significativo nei tratti di lavoro in cui l'autista permane sullo stesso giro per più comandi consecutivi. Ai prezzi correnti delle API di Claude Haiku 4.5, anche in questo scenario più cauto il costo complessivo resta compatibile con un modello di business sostenibile, sebbene una quantificazione precisa del risparmio medio richieda dati di utilizzo raccolti in produzione su un periodo prolungato, e non sia possibile dedurla con certezza dai soli test condotti durante questo tirocinio.

6.5 L'esperimento con il modello locale

La sperimentazione con l'architettura locale basata su Ollama e sul modello Qwen2.5-1.5B, introdotta a livello di motivazione e di scelta del modello nel Capitolo 4 e implementata nel Capitolo 5, ha prodotto un corpus di dati quantitativi che meritano una trattazione specifica accanto a quelli del sistema principale. In questa sezione vengono presentati i risultati delle iterazioni di ottimizzazione del prompt, il confronto diretto con Claude Haiku 4.5 e la valutazione delle prospettive evolutive, integrando l'analisi dei log di esecuzione riportati nel Capitolo 5 con una visione sintetica orientata alla valutazione complessiva del sistema.

6.5.1 I risultati dei test e le iterazioni di ottimizzazione

I test sono stati condotti direttamente sulla macchina server Ubuntu già utilizzata per il middleware. La prima criticità emersa ha riguardato la lunghezza del prompt. Con il template completo da circa 3400 token — lo stesso utilizzato con Claude — il tempo di elaborazione del primo comando è risultato del tutto incompatibile con l'uso operativo:

```
[2026-04-10T08:57:09.783Z] Richiesta: "mostra giro
   consegne"
```

```
[2026-04-10T08:58:24.575Z] Azione: apri_giro_consegna
   (74792ms)
```

Listato 6.2: Log del primo test con prompt completo e Qwen2.5-1.5B.

Il problema non era il modello in sé, bensì il numero di token da elaborare prima di poter generare la prima parola di risposta: su CPU, ogni token in input richiede un tempo di elaborazione che si somma linearmente. Con 3400 token in input, l'elaborazione richiedeva 74 secondi. La soluzione adottata è stata la riduzione progressiva del prompt attraverso quattro iterazioni successive, ciascuna testata su un insieme di casi che coprivano tutte le quattordici azioni operative. I log completi di ciascuna iterazione, con il dettaglio dei comandi di test e dei parametri misurati, sono riportati nel Capitolo 5; di seguito ne vengono sintetizzati gli esiti qualitativi e quantitativi.

Prima iterazione — prompt compresso (600 token, zero esempi). La prima versione ridotta ha eliminato tutti gli esempi few-shot, conservando soltanto l'elenco delle azioni con i relativi parametri e le regole di mapping dei sinonimi. I tempi sono scesi drasticamente, con la maggior parte dei comandi elaborata in tre-dodici secondi; tuttavia, l'accuratezza si è attestata al 50% su sei comandi di riferimento. Il modello senza esempi confondeva il referente con il cliente nell'azione `gestisci_consegna`, inventava nomi di azioni inesistenti, come `visualizza_ricevuta` invece di `stampa_ricevuta_consegna`, e produceva output doppi malformati nel caso dei resi di cauzione.

Seconda iterazione — prompt medio (1100 token, esempi selettivi). La seconda versione ha reintrodotto esempi few-shot limitati alle azioni che il modello gestiva peggio senza dimostrazione esplicita:

- `gestisci_consegna`, con la distinzione tra referente e cliente;
- `registra_resi_cauzioni`, con la conversione dei numeri in lettere;
- `stampa_ricevuta_consegna`, con il nome esatto dell'azione.

L'accuratezza è salita al 70% su dieci casi, con i tempi stabilizzati tra i 2.6 e i 4.7 secondi a partire dal secondo comando. Il primo comando rimaneva, però, sui 22 secondi per il riscaldamento iniziale della cache del modello. Tre problemi persistevano: il fuzzy matching incompleto su nomi parziali come "firenze" che produceva "Firenze" invece di "Firenze Centro"; il mancato riconoscimento dell'assenza dell'importo come condizione di errore per il comando "incassa tutto in contanti"; e la gestione dei comandi fuori contesto, che venivano mappati su azioni esistenti, invece di produrre un `errore_comprensione`.

Terza iterazione — prompt finale (1368 token, esempi mirati). La terza versione ha aggiunto tre regole critiche esplicite e due esempi diretti per i casi problematici:

- la regola di fuzzy matching, obbligatorio con l'istruzione di scrivere sempre la stringa esatta presente nella lista, corredata da un esempio che mostrava la mappatura di "firenze" a "Firenze Centro";
- la regola dell'importo obbligatorio per gli incassi, con un esempio diretto del comando "incassa tutto in contanti", che produce `errore_comprensione`;
- la regola di fuori contesto, con l'esempio di "che tempo fa" che deve ugualmente produrre `errore_comprensione`.

I risultati su quindici casi hanno raggiunto il 93% di accuratezza, con l'unico errore causato dal troncamento del file JSON nell'incasso multiplo per insufficienza del parametro `num_predict`, impostato a 200. I tempi si sono attestati tra i 2.6 e i 5.2 secondi a regime.

Quarta iterazione — versione definitiva (1604 token, temperature zero). La quarta e ultima versione ha portato `num_predict` a 500 per risolvere i troncamenti e `temperature` a zero con `top_k: 1` per attivare il *greedy decoding* ed eliminare la variabilità tra run successivi. Sono stati aggiunti tre ulteriori esempi:

- "torna alla lista", che produce `cambia_vista_documenti` con `tipo_vista: lista`;
- "150 euro cash", che produce `registra_incasso_documento` correttamente;
- un esempio esplicito dell'incasso con pagamento multiplo come array.

I risultati su ventidue casi si sono attestati all'82%, con quattro errori residui:

- il comando "torna alla lista", che produceva `errore_comprensione` invece di `cambia_vista_documenti`;
- il comando "bar roma non consegnato era chiuso", che produceva un file JSON troncato;
- il comando "150 euro cash", che produceva `errore_comprensione`;
- il comando "registra incasso per ditta Acme", che generava un JSON invece dell'errore atteso.

I tempi medi a regime si sono collocati tra i 2.8 e i 5.8 secondi.
La Tabella 6.3 sintetizza i risultati delle quattro iterazioni di ottimizzazione.

Iterazione	Token input	Accuratezza	Latenza a regime	Primo comando
prompt completo	∼3400	n.d.	74 s	74 s
prompt compresso	∼600	50% (3/6)	2.6–4.8 s	∼12 s
prompt medio	∼1100	70% (7/10)	2.6–4.7 s	∼22 s
prompt finale v1	∼1368	93% (14/15)	2.6–5.2 s	∼26 s
prompt finale v2	∼1604	82% (18/22)	2.8–5.8 s	∼26 s

Tabella 6.3: Risultati delle quattro iterazioni di ottimizzazione del prompt per Qwen2.5-1.5B.

È necessario precisare un limite metodologico di questo confronto, prima di trarne conclusioni. Le quattro iterazioni non sono state valutate sulla stessa identica batteria di comandi: il numero di casi di test è cresciuto progressivamente da sei, a dieci, a quindici, fino a ventidue, con l'aggiunta a ogni iterazione di nuovi comandi pensati per coprire scenari via via più specifici (in particolare i casi limite individuati come problematici nell'iterazione precedente). Le percentuali di accuratezza riportate in tabella non sono, quindi, direttamente comparabili come se derivassero da un unico benchmark fisso: un miglioramento percentuale tra un'iterazione e la successiva può in parte riflettere una batteria di test più facile o più piccola, mentre un peggioramento può in parte riflettere l'introduzione di casi più esigenti, e non necessariamente un peggioramento assoluto delle capacità del modello sugli stessi identici comandi.

Detto questo, il calo di accuratezza dal 93% della terza iterazione all'82% della quarta non può essere interpretato semplicemente come un effetto della batteria di test ampliata: l'analisi degli errori residui mostra che almeno due dei quattro fallimenti della quarta iterazione — il comando "torna alla lista", mappato erroneamente su `errore_comprensione` invece di `cambia_vista_documenti`, e il comando "150 euro cash", anch'esso mappato erroneamente su `errore_comprensione` — riguardano categorie di comando (cambio di visualizzazione e registrazione di un incasso in forma sintetica) che le iterazioni precedenti gestivano correttamente o per le quali erano stati aggiunti esempi espliciti proprio in questa stessa iterazione. Si tratta, quindi, di una vera e propria regressione: l'aggiunta di nuovi esempi e regole nel prompt della quarta iterazione, unita al passaggio a `temperature: 0` con `top_k: 1`, ha alterato in modo non lineare il comportamento del modello anche su scenari che la versione precedente gestiva correttamente. Questo fenomeno è coerente con un limite noto dei modelli linguistici di piccole dimensioni instradati tramite *few-shot prompting*: l'aggiunta di nuovi esempi non sempre produce un miglioramento monotono delle prestazioni, poiché può alterare le associazioni implicite che il modello stabilisce tra il testo del prompt e l'output atteso, specialmente in assenza della capacità di generalizzazione più solida tipica dei modelli di dimensioni maggiori. La regressione osservata costituisce, di per sé, un ulteriore elemento a sfavore dell'adozione di un piccolo modello locale con prompt artigianale per questo caso d'uso, e rafforza la conclusione, discussa più avanti, secondo cui un percorso di fine-tuning supervisionato sarebbe preferibile rispetto a un'ulteriore ottimizzazione manuale e incrementale del prompt.

6.5.2 Il confronto con Claude Haiku 4.5

L'analisi comparativa tra le due architetture, basata sui dati misurati durante i test, evidenzia un divario consistente su tutte le dimensioni rilevanti per l'uso operativo.

Metrica	Claude Haiku 4.5	Qwen2.5-1.5B (Ollama)
accuratezza	∼97%	82–93% (instabile)
latenza media	1.1–2.9 s	2.8–5.8 s
primo comando	∼1.5 s	22–26 s (cold start)
stabilità	identica ad ogni run	variabile tra run diversi
JSON malformati	mai	troncamenti occasionali
dipendenza di rete	sì (API cloud)	sì (server middleware)
costo per chiamata	∼0.003 $	nessuno (ma server dedicato)

Tabella 6.4: Confronto tra Claude Haiku 4.5 e Qwen2.5-1.5B locale sui test condotti.

Il problema della variabilità tra run successivi merita una trattazione specifica. Con `temperature` superiore a zero, lo stesso comando poteva produrre il file JSON corretto in un'esecuzione e un errore di parsing nella successiva. Il comando "bar roma non consegnato era chiuso", ad esempio, produceva l'azione `gestisci_consegna` corretta nel test della terza iterazione e un file JSON troncato nel test della quarta iterazione, pur con un prompt sostanzialmente identico. Portare la temperatura a zero con `top_k: 1` ha risolto questa instabilità rendendo il comportamento del tutto deterministico, al costo, però, di eliminare qualsiasi forma di campionamento che potesse aiutare il modello nei casi ambigui.

Un aspetto che l'analisi ha chiarito definitivamente riguarda la natura della dipendenza dalla rete. Contrariamente all'ipotesi iniziale, l'esecuzione di Qwen su server non elimina il requisito di connettività; il dispositivo mobile deve comunque raggiungere il middleware tramite connessione, esattamente come avviene con Claude. L'unico scenario in cui l'approccio locale porterebbe un vantaggio reale rispetto all'architettura cloud è l'esecuzione completamente on-device, che però le misurazioni hanno dimostrato essere impraticabile nelle condizioni attuali: se il primo comando richiede 22-26 secondi sulla macchina server dedicata con il prompt ottimizzato, è ragionevole attendersi un peggioramento significativo su un dispositivo mobile, dove le risorse computazionali disponibili sono generalmente inferiori a quelle di un server dedicato. Si tratta, però, di una stima puramente teorica e qualitativa: nessun test è stato condotto direttamente su hardware mobile in questa tesi, e non si dispone di un benchmark comparativo rigoroso tra la CPU della macchina server utilizzata e quella di uno smartphone di fascia media. Senza una misurazione diretta, ci si limita a osservare che, se anche il rapporto tra le due piattaforme fosse contenuto, i tempi del primo comando si tradurrebbero plausibilmente in un ordine di grandezza superiore al minuto, del tutto incompatibile con l'interazione vocale in tempo reale; una quantificazione precisa di questo scostamento richiederebbe test dedicati su dispositivi mobili reali, che restano un'attività futura necessaria prima di poter trarre conclusioni definitive sulla fattibilità on-device.

6.5.3 Valutazione e prospettive

L'analisi ha portato a una conclusione chiara riguardo alla soluzione da adottare nel breve termine: per il caso d'uso specifico di MyDelivery, Claude Haiku 4.5 rappresenta la scelta più solida sotto ogni dimensione misurabile. L'accuratezza del 97% stabile contro l'82-93% instabile, la latenza media inferiore nonostante la natura cloud, e l'assenza di troncamenti JSON rendono il modello di Anthropic superiore per il contesto operativo attuale.

Il percorso verso un sistema on-device realistico, tuttavia, esiste ed è tecnicamente percorribile a medio termine attraverso il fine-tuning del modello su un dataset specifico per il dominio. Un SLM addestrato su migliaia di varianti reali dei comandi degli autisti — coprendo slang regionali, distorsioni dello speech-to-text, formulazioni telegrafiche e abbreviazioni dialettali — non avrebbe più bisogno di un prompt lungo e ricco di esempi: avrebbe già interiorizzato le quattordici azioni e i loro pattern durante il training, rendendo sufficiente un prompt da duecento-trecento token. Questa riduzione porterebbe la latenza stimata a uno-due secondi su server CPU; una stima ancora più incerta e puramente indicativa, in assenza di test specifici, collocherebbe la latenza su dispositivi mobili dotati di Neural Engine dedicato (iPhone 12 o superiore, Snapdragon 8 Gen 1 o equivalente) in una fascia teoricamente più favorevole rispetto all'esecuzione su CPU generica, ma tale stima andrebbe verificata sperimentalmente prima di essere assunta come dato di progetto. Anche in questo scenario con fine-tuning, tuttavia, Claude Haiku rimarrebbe più veloce e più preciso quando la connessione è disponibile; la configurazione ibrida — modello locale quando offline, Claude quando online — rappresenta l'architettura evolutiva più razionale per MyDelivery.

Il dataset di training potrebbe essere generato in due fasi: una prima fase sintetica, in cui Claude stesso genera varianti dei comandi operativi a partire dagli esempi del prompt attuale, con un costo stimato di due-tre euro per mille esempi; e una seconda fase basata sui dati reali raccolti durante il funzionamento in produzione del sistema, che contengono le vere distorsioni dello speech-to-text sui dispositivi degli autisti e i pattern linguistici effettivi emersi dall'uso sul campo. Questi dati di produzione rappresentano un patrimonio di qualità superiore rispetto a qualsiasi generazione sintetica, poiché riflettono il comportamento reale del sistema in condizioni operative autentiche.

La roadmap evolutiva identificata si articola su tre orizzonti temporali: nel breve termine, il sistema basato su Claude Haiku viene messo in produzione così come descritto in questa

tesi, garantendo accuratezza e stabilità; nel medio termine (tre-sei mesi), si raccolgono i dati di interazione reale degli autisti e si costruisce un dataset di fine-tuning; nel lungo termine (sei-dodici mesi), si esegue il fine-tuning su Qwen2.5-1.5B o un modello equivalente e si integra il risultato on-device nell'app Flutter, con Claude Haiku come fallback automatico quando il dispositivo è connesso.

6.6 Criticità emerse e limitazioni

Nonostante i risultati complessivamente positivi, durante la fase di test sono emerse alcune criticità che meritano di essere documentate.

La prima riguarda la correttezza dell'identificativo del modello. Come documentato nel capitolo precedente, il nome `claude-4-5-haiku-20251001` impostato inizialmente ha prodotto un errore 404 da parte delle API di Anthropic. Il formato corretto `claude-haiku-4-5-20251001` antepone il nome della variante al numero di versione. Questo tipo di errore silenzioso — il server rispondeva con un messaggio generico invece di propagare il dettaglio dell'errore API all'utente — ha reso necessaria la lettura dei log del primo terminale per identificare la causa. In uno scenario di produzione con versioning automatico del modello sarebbe opportuno implementare un controllo esplicito all'avvio che verifichi la raggiungibilità del modello configurato.

La seconda criticità, presente nella versione iniziale del server e risolta dall'introduzione del prompt caching, riguardava il costo del prompt di sistema. Il template di istruzioni, contenente tutti i quattordici esempi di few-shot prompting, occupa circa 3400 token ad ogni richiesta nella modalità senza caching. Con il caching abilitato questo costo viene sostenuto integralmente soltanto al primo invio della sessione; i comandi successivi trasmettono all'API esclusivamente il blocco dinamico, come documentato nella Sezione 6.4. Rimane, comunque, aperta la possibilità di una ulteriore ottimizzazione futura che preveda la suddivisione del prompt in versioni ridotte per le operazioni più frequenti, da adottare qualora il costo del primo invio risultasse significativo in scenari con sessioni molto brevi.

La terza criticità è legata all'assenza, in questa fase, di un meccanismo di conferma esplicita prima dell'esecuzione delle azioni. Per operazioni delicate, come la registrazione di un incasso o la chiusura di una consegna, potrebbe essere opportuno implementare un passaggio di conferma vocale prima dell'esecuzione effettiva. Il prompt di sistema già prevede che il modello proponga i valori recuperati dal contesto — come dimostrato nel terzo esempio della sessione di test, dove l'importo di 150 euro viene suggerito e la risposta vocale si chiude con una richiesta implicita di conferma — ma la gestione della risposta dell'autista è demandata al dispatcher dell'applicazione Flutter, la cui implementazione è a carico del team di Infoservice.

La quarta criticità, emersa esclusivamente durante i test su dispositivo mobile, riguarda la raggiungibilità del server middleware dall'esterno della rete del provider. La macchina server, dotata di indirizzo IP privato e posizionata dietro un NAT, non risulta accessibile sulla porta 3000 dall'esterno. Per la fase di test questa limitazione è stata superata tramite il servizio di tunneling ngrok, ma per un ambiente di produzione sarà necessario che l'azienda configuri l'infrastruttura di rete in modo da garantire la raggiungibilità stabile del middleware.

La quinta criticità riguarda la natura non incondizionata del beneficio del prompt caching, già anticipata nella Sezione 6.4. Il meccanismo di caching adottato congela in cache l'intero blocco statico del prompt, comprendente sia le istruzioni di sistema sia il contesto dei dati appiattiti (città, giri, clienti e relativi dettagli) prodotto da `appiattisciStruttura()`. Poiché questo blocco viene rigenerato e iniettato nel messaggio a ogni richiesta sulla base dei dati ricevuti dall'applicazione Flutter, qualunque variazione nel suo contenuto — ad esempio il cambio del giro di consegna attivo, il passaggio a una nuova città o, più in generale, qualsiasi

aggiornamento delle liste di riferimento nel corso della giornata — produce un blocco statico testualmente diverso da quello precedente e, di conseguenza, invalida la cache esistente. Il comando successivo a un cambiamento di questo tipo torna quindi a costare l'intero prompt di circa 3400 token, esattamente come il primo comando di una nuova sessione. Il risparmio del 98% documentato nella Sezione 6.4 si applica dunque correttamente solo alle sequenze di comandi impartiti senza variazioni del contesto dati, e non a una giornata operativa intera, durante la quale un autista cambia necessariamente giro e cliente più volte. Questo limite non invalida l'utilità del meccanismo, che resta vantaggioso nei tratti di lavoro più frequenti in cui l'autista opera su un singolo giro per diversi comandi consecutivi, ma va tenuto presente nella stima dei costi complessivi di esercizio e meriterebbe, in un possibile sviluppo futuro, un meccanismo di caching più granulare in cui le istruzioni di sistema (realmente invarianti) siano separate dal contesto dati (variabile) in due blocchi distinti, in modo che il cambio di giro o cliente invalidi solo la porzione di cache relativa ai dati e non quella, ben più consistente in termini di token, relativa alle istruzioni di sistema.

6.6.1 I limiti dell'approccio con modello locale

L'analisi sperimentale condotta con Qwen2.5-1.5B tramite Ollama, presentata nel dettaglio nella Sezione 6.5, ha evidenziato una sesta criticità di natura più strutturale, che riguarda i limiti fondamentali dell'approccio locale rispetto a quello basato su API cloud per il caso d'uso specifico di MyDelivery. Si sintetizzano qui le implicazioni rilevanti per la valutazione complessiva del sistema, rimandando alla sezione citata per i dati quantitativi completi e per la discussione delle iterazioni di ottimizzazione.

Sul piano dell'accuratezza, il modello locale ha raggiunto valori variabili tra l'82 e il 93% a seconda della versione del prompt, a fronte del 97% stabile di Claude Haiku. La variabilità tra run successivi sullo stesso input, osservata con valori di temperatura superiori a zero, ha rappresentato il problema più grave per un sistema destinato all'uso operativo: un autista che pronuncia lo stesso comando in momenti diversi della giornata non può ricevere risposte diverse. La soluzione adottata, ovvero portare `temperature` a zero con `top_k: 1`, ha eliminato questa instabilità al costo di rendere il modello completamente deterministico anche nei casi ambigui, dove una forma di campionamento avrebbe potuto aiutare la comprensione. I troncamenti occasionali del file JSON nelle strutture più complesse, inoltre, hanno reso necessario portare `num_predict` a 500, un valore che introduce un overhead di elaborazione non trascurabile per i comandi più semplici, che avrebbero bisogno di soli 40-50 token.

La criticità più rilevante, tuttavia, è di natura architetturale: l'esecuzione locale su server, a differenza di quanto inizialmente ipotizzato, non elimina la dipendenza dalla rete. Il middleware rimane necessario e il dispositivo mobile deve comunque raggiungerlo tramite connessione, esattamente come avviene con Claude. Il vero vantaggio dell'approccio locale si realizzerebbe soltanto con l'esecuzione completamente on-device, che le misurazioni hanno però dimostrato impraticabile nelle condizioni attuali, con tempi del primo comando compresi tra 22 secondi e 74 secondi sulla macchina server dedicata. Si ribadisce, inoltre, che tali valori sono stati misurati esclusivamente sulla macchina server e non su un dispositivo mobile reale; una valutazione rigorosa della fattibilità on-device richiederebbe una campagna di test dedicata su hardware mobile, che resta un'attività futura necessaria prima di poter trarre conclusioni definitive sulla praticabilità di un'architettura completamente offline.

Conclusioni

Nel corso di questo lavoro è stato progettato e realizzato un assistente vocale basato su Large Language Model per l'applicazione mobile MyDelivery di Infoservice, con l'obiettivo di consentire agli autisti consegnatari di controllare l'intera applicazione tramite comandi espressi in linguaggio naturale, senza necessità di interagire manualmente con l'interfaccia touch. L'architettura adottata, completamente disaccoppiata su tre livelli, ha permesso di integrare il componente di Intelligenza Artificiale nell'ecosistema esistente senza modificare il codice sorgente dell'applicazione Flutter e senza esporre le credenziali API sul dispositivo mobile. Il server middleware Node.js, sviluppato nell'ambito di questo tirocinio, ha assunto il ruolo di intermediario sicuro tra il dispositivo dell'autista e il modello linguistico Claude Haiku di Anthropic, custodendo le chiavi API nell'ambiente server e assemblando dinamicamente il prompt di sistema a ogni richiesta. Il prompt incorpora tecniche di *few-shot prompting* e un meccanismo di *fuzzy matching* dinamico che consente al modello di risolvere nomi pronunciati in modo approssimativo o parziale, abbinandoli agli identificatori esatti presenti nelle liste di riferimento fornite dall'applicazione. Il sistema copre quattordici scenari operativi distinti e, nelle prove condotte con il modello Claude Haiku, ha raggiunto un'accuratezza del 97% con latenze medie comprese tra 1,1 e 2,9 secondi, un valore che soddisfa il vincolo non funzionale di cinque secondi complessivi pur non rientrando, per alcuni comandi più articolati, nel traguardo ideale di due secondi fissato in fase di progettazione. La valutazione sperimentale comparativa con il modello locale Qwen2.5-1.5B eseguito tramite Ollama ha chiarito i limiti dell'approccio completamente locale nelle condizioni attuali, evidenziando tempi di avvio e variabilità tra esecuzioni successive incompatibili con il contesto d'uso. La validazione del primo approccio ha misurato latenze comprese tra 986 e 2929 millisecondi e ha documentato, in una sessione di test condotta su comandi consecutivi senza variazioni del contesto operativo, un risparmio superiore al 98% nel consumo di token in ingresso grazie al meccanismo di prompt caching, che riduce il blocco da trasmettere da oltre tremila token a poche decine quando giro e cliente restano invariati tra un comando e il successivo; tale beneficio si riduce nella pratica operativa reale, in cui i cambi frequenti di giro e di cliente nel corso della giornata invalidano periodicamente la cache, e la sua quantificazione su scala di flotta richiederebbe dati raccolti in produzione su un periodo prolungato. Il lavoro ha al contempo identificato un percorso evolutivo realistico attraverso il fine-tuning su dati di produzione.

Gli sviluppi futuri più promettenti si collocano su tre orizzonti distinti. Nel breve termine, la messa in produzione del sistema basato su Claude Haiku — che continuerà a utilizzare il prompt di sistema completo da circa 3400 token con il meccanismo di prompt caching descritto nel Capitolo 5 — consentirà di raccogliere dati di interazione reale, ovvero le

trascrizioni effettive dei comandi degli autisti con le distorsioni dello *speech-to-text* specifiche dei dispositivi in uso e i pattern linguistici regionali che nessuna generazione sintetica può anticipare. Questi dati rappresentano il presupposto per il passo successivo: la costruzione di un dataset di *fine-tuning* su un modello leggero, come Qwen2.5-1.5B o uno equivalente, che abbia già interiorizzato le quattordici azioni operative durante il training e possa operare con un prompt da 200-300 token. Si tratta di un valore ulteriormente ridotto rispetto al prompt sperimentale da circa 1600 token utilizzato nei test descritti nel Capitolo 4 per l'esperimento di ricerca e sviluppo con Qwen2.5-1.5B su Ollama — prompt distinto e indipendente da quello di produzione usato con Claude, e ottenuto ottimizzando in quattro iterazioni successive un primo prompt sperimentale di partenza, anch'esso di circa 3400 token ma testato unicamente in quel contesto locale, non da confondere con il prompt di produzione di Claude che resta invariato. Questo ridurrebbe la latenza stimata a uno-due secondi su server CPU e renderebbe praticabile, nel lungo termine, una configurazione ibrida in cui il modello locale gestisce i comandi in assenza di connettività e Claude Haiku viene utilizzato come sistema di riferimento quando la rete è disponibile. Un secondo filone di sviluppo riguarda l'estensione della copertura funzionale del sistema: l'aggiunta di nuovi scenari operativi, come la gestione delle eccezioni di percorso o l'interazione con il backoffice web, amplierebbe ulteriormente il valore operativo dell'assistente vocale per le aziende clienti di Infoservice. Infine, l'approccio architetturale sviluppato in questa tesi, basato su un middleware stateless che assembla dinamicamente il contesto ad ogni richiesta senza richiedere addestramento specifico per ciascun cliente, si presta a essere esteso ad altri prodotti del portafoglio Infoservice e, più in generale, a qualsiasi applicazione gestionale in cui l'interazione vocale possa ridurre le frizioni operative degli utenti sul campo.

Bibliografia

[1] Anthropic. The claude model family: Technical overview. Technical report, Anthropic, San Francisco, 2024. Disponibile all'indirizzo: `https://www.anthropic.com/claude`. (Cited at pages 5 e 6)

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dharival, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020. (Cited at pages 4 e 5)

[3] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6): 391–407, 1990. (Cited at page 3)

[4] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of NAACL-HLT 2019*, pages 4171–4186, Minneapolis, 2019. Association for Computational Linguistics. (Cited at page 5)

[5] Gemini Team, Google. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023. Disponibile all'indirizzo: `https://arxiv.org/abs/2312.11805`. (Cited at page 5)

[6] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997. (Cited at page 3)

[7] M. B. Hoy. Alexa, siri, cortana, and more: An introduction to voice assistants. *Medical Reference Services Quarterly*, 37(1):81–88, 2018. (Cited at page 6)

[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33 (1):31–88, 2001. (Cited at page 7)

[9] OpenAI. *OpenAI API Reference*. OpenAI, San Francisco, 2023. Disponibile all'indirizzo: `https://platform.openai.com/docs/api-reference`. (Cited at page 6)

[10] OpenAI. Gpt-4 technical report. Technical report, OpenAI, San Francisco, 2023. Disponibile all'indirizzo: `https://arxiv.org/abs/2303.08774`. (Cited at page 5)

[11] OpenAI. Openai o1 system card. Technical report, OpenAI, San Francisco, 2024. Disponibile all'indirizzo: `https://openai.com/index/openai-o1-system-card/`. (Cited at page 6)

[12] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744, 2022. (Cited at page 5)

[13] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, San Francisco, 2018. Disponibile all'indirizzo: `https://openai.com/research/language-unsupervised`. (Cited at page 5)

[14] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023. Disponibile all'indirizzo: `https://arxiv.org/abs/2302.04761`. (Cited at page 6)

[15] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, volume 27, pages 3104–3112, 2014. (Cited at page 3)

[16] H. Touvron, T. Lavril, G. Izaard, X. Martinet, A. Ashok, A. Bashlykov, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. Disponibile all'indirizzo: `https://arxiv.org/abs/2302.13971`. (Cited at page 5)

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017. (Cited at page 4)

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022. (Cited at page 5)

[19] J. Weizenbaum. Eliza — a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966. (Cited at page 3)

[20] J. White, Q. Fu, S. Hays, J. Sandborn, C. Overholt, H. Barringer, et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023. Disponibile all'indirizzo: `https://arxiv.org/abs/2302.11382`. (Cited at page 5)

Sitografia

- Anthropic, documentazione ufficiale delle API – https://docs.anthropic.com
- Dart, documentazione ufficiale del linguaggio Dart – https://dart.dev/guides
- Electric SQL, documentazione del sistema di sincronizzazione bidirezionale – https://electric-sql.com/docs
- Express.js, documentazione del framework Node.js – https://expressjs.com
- Flutter, documentazione ufficiale del framework Flutter – https://docs.flutter.dev
- Google Scholar, motore di ricerca per articoli scientifici – https://scholar.google.com
- Infoservice s.r.l., sito ufficiale dell'azienda – https://www.infoservice.it
- ngrok, documentazione del servizio di tunneling inverso – https://ngrok.com/docs
- Node.js, documentazione ufficiale del runtime JavaScript – https://nodejs.org/en/docs
- NodeSource, repository ufficiale per l'installazione di Node.js su sistemi Debian/Ubuntu – https://github.com/nodesource/distributions
- npm, documentazione del gestore di pacchetti Node.js – https://docs.npmjs.com
- PostgreSQL, documentazione ufficiale – https://www.postgresql.org/docs
- SQLite, documentazione ufficiale del database locale – https://www.sqlite.org/docs.html
- Ubuntu Server, documentazione ufficiale di Ubuntu – https://ubuntu.com/server/docs
- Zucchetti Group, sito ufficiale del gruppo di cui Infoservice è partner – https://www.zucchetti.it

Ringraziamenti

Il primo e più importante ringraziamento va alla mia famiglia, che mi ha sempre supportato, spronandomi a coltivare con costanza e dedizione le mie passioni, e aiutandomi a bilanciare le mie giornate tra studio, amicizie e altri interessi.

Un altro sentito ringraziamento va ai miei amici sparsi per l'Italia e per il mondo, la cui compagnia e il cui affetto hanno permesso a questi anni universitari di trascorrere serenamente, dandomi la giusta carica per affrontare ogni giornata di intenso studio.

Vorrei, inoltre, ringraziare il Prof. Domenico Ursino, che mi ha guidato nel lavoro di tesi seguendomi costantemente con estrema disponibilità, e il Dott. Gianluca Bonifazi, il quale ha fornito importantissimi spunti di riflessione e consigli preziosi per migliorare la gestione di questo progetto.