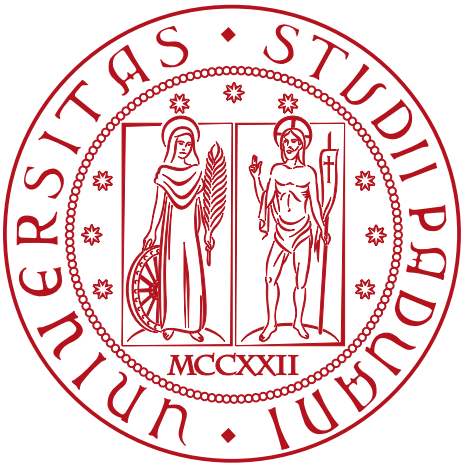
Università degli studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Integrazione di Large Language Models nei sistemi ERP: il caso AI LiveBot su Odoo

Tesi di laurea

Relatore
Prof. Zanella Marco

Laureando
Egidi Marco
Matricola 2082851

ANNO ACCADEMICO 2025-2026

«I don't care what my place in the world is. I just want to know what I am capable of.»

— Eiichirō Oda

Ringraziamenti

Innanzitutto desidero esprimere la mia più sincera gratitudine al relatore della mia tesi Prof. Zanella Marco, per la disponibilità, l'aiuto e il costante supporto offertomi durante la stesura del lavoro.
Rivolgo poi un affettuoso ringraziamento ai miei genitori e ai miei familiari, per il sostegno e la vicinanza dimostratimi nel corso degli anni di studio.
Desidero inoltre ringraziare i miei amici per i preziosi momenti condivisi e per tutti i bellissimi anni trascorsi insieme.
Vorrei infine ringraziare i colleghi di SyncLab SRL e il tutor aziendale Pallaro Fabio, per avermi dato l'opportunità di lavorare ed imparare all'interno di questo progetto e per il supporto ricevuto.

Padova, Aprile 2026 *Egidi Marco*

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage curricolare, della durata di circa trecentoventi ore, dal laureando Egidi Marco presso l'azienda SyncLab SRL. Lo stage è stato condotto sotto la supervisione del tutor aziendale Pallaro Fabio, mentre il prof. Prof. Zanella Marco ha ricoperto il ruolo di tutor accademico.

Questa tesi tratta la progettazione e lo sviluppo di **AI LiveBot**, un modulo per la piattaforma Odoo 18 che integra funzionalità di intelligenza artificiale per la gestione assistita del magazzino e degli ordini di vendita. L'obiettivo è quello di semplificare l'interazione con il sistema gestionale attraverso l'uso del linguaggio naturale, riducendo i tempi operativi e migliorando l'accessibilità per gli utenti.

Organizzazione del testo

Il primo capitolo introduce l'azienda, il progetto e le motivazioni che mi hanno portato a sceglierlo;

Il secondo capitolo descrive l'organizzazione dello stage, gli obiettivi formativi e la pianificazione delle attività;

Il terzo capitolo illustra l'analisi dei requisiti funzionali e qualitativi del sistema;

Il quarto capitolo presenta le tecnologie e gli strumenti utilizzati, con un focus su Odoo e LLM;

Il quinto capitolo descrive l'architettura della soluzione, le problematiche affrontate e l'implementazione del modulo;

Il sesto capitolo trae le conclusioni sull'esperienza di stage e analizza i risultati ottenuti.

Convenzioni tipografiche

Durante la stesura del testo ho scelto di adottare le seguenti convenzioni tipografiche:

- Gli acronimi, le abbreviazioni e i termini di uso non comune menzionati vengono definiti nel glossario, situato alla fine del documento (p. 50);

- Per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *termine*$_G$;
- I termini in lingua straniera non di uso comune o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*;
- I nomi di funzioni o variabili appartenenti ad un linguaggio di programmazione vengono scritte con un carattere `monospaziato`;
- Le citazioni ad un libro o ad una risorsa presente nella bibliografia (p. 54) saranno affiancate dal rispettivo numero identificativo, es. [1];
- I blocchi di codice sono rappresentati nel seguente modo

```c
float Q_rsqrt( float number ){
  long i;
  float x2, y;
  const float threehalfs = 1.5F;
  x2 = number * 0.5F;
  y  = number;
  i  = * (long * ) &y;
  i  = 0x5f3759df - (i>>1);
  y  = * (float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );
//y  = y * ( threehalfs - ( x2 * y * y ) );
  return y;
}
```

Codice 1: Codice d'esempio.

Indice

Elenco delle Figure

Elenco delle Tabelle

Elenco dei Codici Sorgente

Capitolo 1

Introduzione

L'evoluzione dei sistemi Enterprise Resource Planning (ERP) (*Enterprise Resource Planning*) ha trasformato radicalmente il modo in cui le aziende gestiscono le proprie risorse, centralizzando dati e processi in un'unica piattaforma integrata. Tuttavia, la complessità di tali sistemi rappresenta spesso una barriera all'ingresso per gli utenti meno esperti, richiedendo una conoscenza approfondita dei menu e delle procedure operative.
In questo contesto, l'Intelligenza Artificiale Generativa e i *Large Language Models* (Large Language Model (LLM)) offrono nuove opportunità per semplificare l'interazione uomo-macchina. L'integrazione di assistenti virtuali intelligenti all'interno dei software gestionali permette di trasformare comandi complessi in conversazioni naturali, riducendo i tempi di operatività e minimizzando gli errori umani.
Il presente lavoro di tesi descrive l'attività di stage svolta presso l'azienda Sync Lab, focalizzata sulla progettazione e sviluppo di «AI LiveBot», un modulo per la piattaforma Odoo 18 che integra funzionalità di intelligenza artificiale per la gestione assistita del magazzino e degli ordini di vendita.

1.1 L'azienda

Lo stage si è svolto presso **Sync Lab**, un'azienda italiana nata nel 2002 con sede principale a Napoli, che si è affermata rapidamente nel mercato dell'*Information and Communication Technology* (Information and Communication Technology (ICT)). Nata inizialmente come *software house*, Sync Lab ha vissuto un processo di maturazione delle competenze tecnologiche e metodologiche che l'ha portata a evolversi in un *System Integrator* completo. L'azienda collabora con diversi clienti per offrire servizi di progettazione, realizzazione e manutenzione di soluzioni Information Technology (IT), con l'obiettivo di supportare il cliente sia dal punto di vista tecnologico sia nel governo del cambiamento organizzativo.

Figura 1: Logo di SyncLab S.r.l.

Sync Lab ha puntato molto sull'innovazione, sviluppando prodotti proprietari nei propri laboratori di ricerca e sviluppo e conquistando fette di mercato in settori strategici quali la *cybersecurity*, la videosorveglianza, il mobile e la sicurezza delle infrastrutture informatiche. Attualmente l'azienda conta oltre 250 dipendenti dislocati in cinque sedi sul territorio italiano: Roma, Napoli, Milano, Verona e Padova. Tutte le sedi collaborano in sinergia, avvalendosi di specialisti per supportare partner e clienti di rilievo, tra cui figurano realtà come Enel, Sky, Vodafone, Tim, Fastweb, Trenitalia, Poste Italiane, UniCredit e Intesa Sanpaolo.
I principali clienti citati sono illustrati in Figura 2.

Figura 2: Clienti di SyncLab S.r.l.

1.2 L'idea e lo scopo dello stage

L'idea alla base del progetto di stage nasce dalla necessità di rendere più accessibili e immediate le operazioni di gestione del magazzino all'interno dell'ERP Odoo. L'obiettivo principale è stato lo sviluppo di **AI LiveBot**, un assistente virtuale integrato nel sistema di chat nativo di Odoo 18.

Figura 3: Logo di Odoo

Il modulo realizzato permette agli operatori di interagire con il sistema gestionale utilizzando il linguaggio naturale, eliminando la necessità di navigare attraverso menu complessi per operazioni di routine. Attraverso l'integrazione con modelli LLM come Google Gemini [1], l'assistente è in grado di:

- Interpretare richieste dell'utente (es. «Controlla la disponibilità di Cestino Rosso»);
- Eseguire ricerche intelligenti (*fuzzy search*) nel catalogo prodotti;
- Verificare giacenze e disponibilità in tempo reale;
- Creare ordini di vendita e validare consegne direttamente dalla chat;
- Supportare la dettatura vocale (Speech-to-Text (STT)) per l'inserimento dei comandi.

Figura 4: Logo di Gemini

1.3 Motivazioni

La scelta di intraprendere questo percorso di stage è stata dettata dal forte interesse verso l'applicazione pratica dell'Intelligenza Artificiale in contesti aziendali reali. In particolare, la possibilità di lavorare su Odoo, uno dei framework ERP open source più diffusi al mondo, combinandolo con le più recenti tecnologie di *Generative AI*, ha rappresentato una sfida stimolante sia dal punto di vista tecnico che progettuale. L'opportunità di approfondire il linguaggio Python, l'architettura modulare di Odoo e le API dei moderni LLM ha costituito un'occasione fondamentale per consolidare le conoscenze acquisite durante il percorso di studi e applicarle allo sviluppo di una soluzione innovativa e concreta.

1.4 Organizzazione del testo

Il presente documento è organizzato nei seguenti capitoli:

Il secondo capitolo descrive l'organizzazione dello stage, gli obiettivi formativi e la pianificazione delle attività.

Il terzo capitolo illustra l'analisi dei requisiti funzionali e qualitativi del sistema.

Il quarto capitolo presenta le tecnologie e gli strumenti utilizzati, con un focus su Odoo e LLM.

Il quinto capitolo descrive l'architettura della soluzione, le problematiche affrontate e l'implementazione del modulo.

Il sesto capitolo trae le conclusioni sull'esperienza di stage e analizza i risultati ottenuti.

Riguardo la stesura del testo, sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini tecnici sono definiti nel glossario;
- i termini in lingua straniera o facenti parte del gergo tecnico sono evidenziati in *corsivo*.

Capitolo 2

Descrizione stage

In questo capitolo viene descritta l'organizzazione dello stage, definendo gli obiettivi formativi, la pianificazione temporale delle attività e la metodologia di lavoro adottata in collaborazione con l'azienda ospitante.

2.1 Competenze da apprendere

Durante il percorso di stage, l'obiettivo formativo principale è stato l'acquisizione di competenze specifiche relative allo sviluppo di moduli per sistemi ERP e all'integrazione di tecnologie di Intelligenza Artificiale. In particolare:

- **Linguaggi e tecnologie web**: approfondimento di Python, eXtensible Markup Language (XML), XML Path Language (XPath),

HyperText Markup Language (HTML5) (HTML5), Cascading Style Sheets (CSS3) (CSS3) e JavaScript (JS);
- **Database**: gestione di database relazionali PostgreSQL;
- **Framework Odoo**: comprensione dell'architettura modulare di Odoo 18, del sistema ORM, delle Viste e dei meccanismi di estensione;
- **Intelligenza Artificiale**: studio delle API di modelli LLM (come Google Gemini e OpenAI), tecniche di **Prompt Engineering** e gestione di output strutturati (JSON Schema);
- **Strumenti di sviluppo**: utilizzo avanzato di Git per il versionamento del codice.

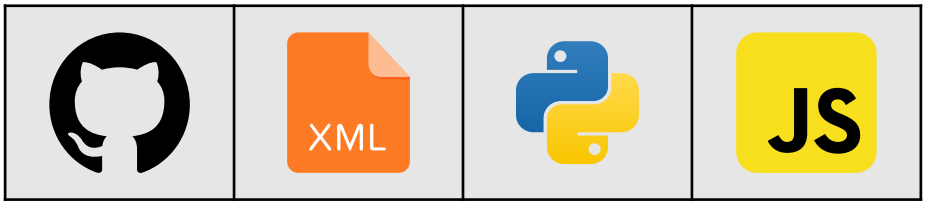


Tabella 1: Loghi: GitHub, XML, Python, JavaScript

2.2 Vincoli

Lo svolgimento del progetto è stato soggetto ai seguenti vincoli:
- **Temporali**: il progetto doveva essere completato entro un monte ore totale di 320 ore;
- **Tecnologici**: utilizzo obbligatorio della piattaforma Odoo (versione 18) e del linguaggio Python. L'integrazione doveva avvenire sfruttando il sistema di chat nativo di Odoo;
- **Funzionali**: il sistema doveva essere in grado di interpretare il linguaggio naturale, interrogare il magazzino e creare ordini di vendita previa validazione.

2.3 Pianificazione

Le attività sono state pianificate per una durata complessiva di 8 settimane (320 ore), suddivise come segue:
- **Prima settimana (40h)**: Studio e ripasso delle tecnologie di base (Structured Query Language (SQL), HTML5/CSS3/JS, Python) e sviluppo di script di interazione con PostgreSQL.
- **Seconda settimana (40h)**: Studio approfondito della piattaforma Odoo 18 (moduli Magazzino, Ordini, ChatBot) e implementazione di un meccanismo di intercettazione dei messaggi nella chat.

- **Terza settimana (40h)**: Studio dei modelli LLM (Gemini, ChatGPT) e realizzazione di un prototipo **standalone** in Python per testare le API tramite librerie come LangChain.
- **Quarta settimana (40h)**: Progettazione architetturale della soluzione integrata, definizione dei casi di test e studio delle strutture dati di Odoo per giacenze e ordini.
- **Quinta settimana (40h)**: Implementazione dell'integrazione tra Odoo e l'LLM scelto, con realizzazione del primo prototipo funzionante in chat.
- **Sesta settimana (40h)**: Sviluppo del **System Prompt** per la gestione degli ordini e implementazione del flusso completo: richiesta utente -> recupero dati magazzino -> generazione JSON -> creazione ordine.
- **Settima settimana (40h)**: Proseguimento degli sviluppi, raffinamento del codice e gestione dei casi d'errore.
- **Ottava settimana (40h)**: Test finali, collaudo del sistema, verifica dei requisiti e stesura della documentazione tecnica.

2.4 Organizzazione del lavoro

Il lavoro è stato svolto in stretta collaborazione con il tutor aziendale, **Fabio Pallaro**. La metodologia adottata ha previsto:

- **Incontri periodici**: check-point settimanali per verificare lo stato di avanzamento, chiarire dubbi e affinare gli obiettivi;
- **Versionamento**: utilizzo di un repository Git aziendale per il mantenimento del codice sorgente;
- **Documentazione**: stesura parallela di un documento tecnico descrittivo delle scelte progettuali.

Gli obiettivi del progetto sono stati classificati in:

- **Obbligatori (O)**: Acquisizione conoscenze, implementazione interazione LLM-Odoo, gestione ordini tramite linguaggio naturale.
- **Desiderabili (D)**: Autonomia nelle customizzazioni Odoo.
- **Facoltativi (F)**: Implementazione funzionalità **Speech-to-Text**.

Nel corso delle settimane il piano iniziale è stato affinato in base ai risultati intermedi. Nella fase centrale sono stati introdotti in modo incrementale sia il **linguaggio di funzioni** utilizzato dall'assistente per richiamare le operazioni Odoo, sia il modello astratto di servizio `warehouse.operations`, che ha permesso di concentrare in un unico punto la logica di dominio (ricerca prodotti, gestione ordini, delivery e statistiche). Nelle ultime settimane è stata posta particolare attenzione alla definizione degli schemi JSON e dei flussi di conferma (creazione, modifica e annullamento ordini), oltre che alla redazione del manuale utente e alla validazione tramite scenari d'uso realistici.

2.5 Analisi dei rischi

Durante la fase di pianificazione sono stati individuati i rischi potenziali riportati nella Tabella 2.

Rischio	Descrizione	Mitigazione
R1 - Tecnologico	Difficoltà nell'integrazione tra l'ambiente Odoo e le API esterne degli LLM.	Realizzazione di un prototipo **standalone** nella terza settimana per isolare la complessità.
R2 - Apprendimento	Curva di apprendimento elevata per il framework Odoo 18.	Dedicate 40 ore iniziali allo studio specifico della piattaforma e dei moduli standard.
R3 - Affidabilità LLM	Possibilità che il modello generi output non conformi al JSON Schema richiesto (allucinazioni).	Utilizzo di tecniche di **Function Calling** e validazione rigida del JSON ricevuto prima dell'elaborazione.

Tabella 2: Tabella dei rischi preventivati.

Capitolo 3

Analisi dei requisiti

In questo capitolo viene presentata l'analisi dei requisiti del sistema **AI LiveBot**. L'analisi è stata condotta partendo dagli obiettivi definiti nel piano di lavoro e raffinata attraverso gli incontri periodici con il tutor aziendale. Lo scopo è definire in modo chiaro e non ambiguo le funzionalità che il modulo deve offrire e i vincoli che deve rispettare.

3.1 Analisi degli stakeholder

Per una corretta definizione dei requisiti, è fondamentale identificare gli attori coinvolti nel progetto:

- **Committente (Sync Lab)**: L'azienda ospitante, interessata a valutare la fattibilità e l'efficacia dell'integrazione di LLM all'interno dell'ecosistema Odoo per proporre soluzioni innovative ai propri clienti.
- **Utenti finali (Operatori/Agenti)**: Il personale addetto alle vendite o alla gestione del magazzino che utilizzerà il sistema. Necessitano di uno strumento rapido e intuitivo per svolgere operazioni ripetitive senza dover navigare tra menu complessi.
- **Sviluppatore (Stagista)**: Responsabile dell'analisi, progettazione e implementazione del modulo, garantendo il rispetto dei vincoli tecnici e temporali.

3.2 User stories

Per descrivere le funzionalità dal punto di vista dell'utente, è stata adottata la metodologia delle **User Stories**. Di seguito sono riportate le storie principali che hanno guidato lo sviluppo.

Epic 1. Interazione e Assistenza

US1 Interazione via Chat

Un esempio di inizio interazione è mostrato in Figura 5.

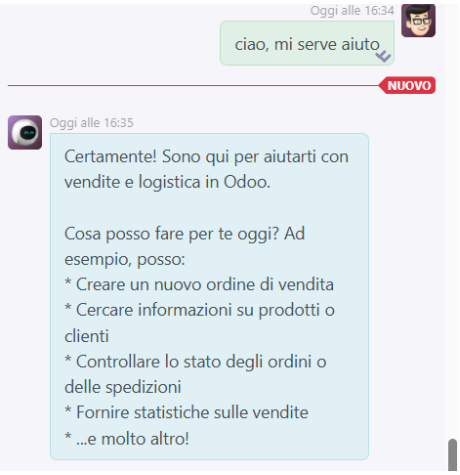


Figura 5: Esempio di inizio interazione tramite chat.

Descrizione: Come operatore, voglio poter inviare richieste in linguaggio naturale tramite la chat di Odoo, per evitare di dover cercare manualmente le informazioni nei menu.

Acceptance criteria:
1. L'utente scrive un messaggio nella chat (es. «Ciao, mi serve aiuto»).
2. Il sistema riconosce che il messaggio è rivolto al bot.
3. Il sistema risponde in modo coerente e contestuale.

US2 Verifica disponibilità prodotti

Un esempio di verifica disponibilità è mostrato in Figura 6.

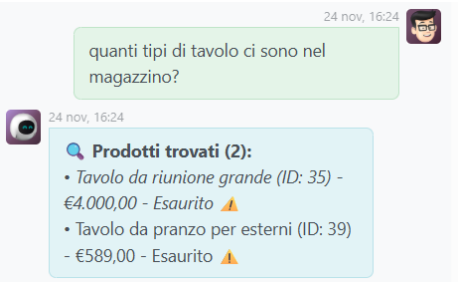


Figura 6: Esempio di verifica disponibilità prodotti tramite chat.

Descrizione: Come agente di vendita, voglio sapere immediatamente se un prodotto è disponibile in magazzino chiedendolo al bot, per poter dare risposte rapide ai clienti.

Acceptance criteria:
1. L'utente chiede la disponibilità di un prodotto (es. «Quanti Cestini Rossi abbiamo?»).

2. Il sistema identifica il prodotto nel database (anche con ricerca approssimata).
3. Il sistema restituisce la quantità esatta disponibile («Abbiamo 50 unità di Cestino Rosso»).
4. Se il prodotto non esiste, il sistema informa l'utente.

Epic 2. Gestione Ordini

US3 Creazione Ordine di Vendita

Un esempio di creazione ordine tramite chat è mostrato in Figura 7.

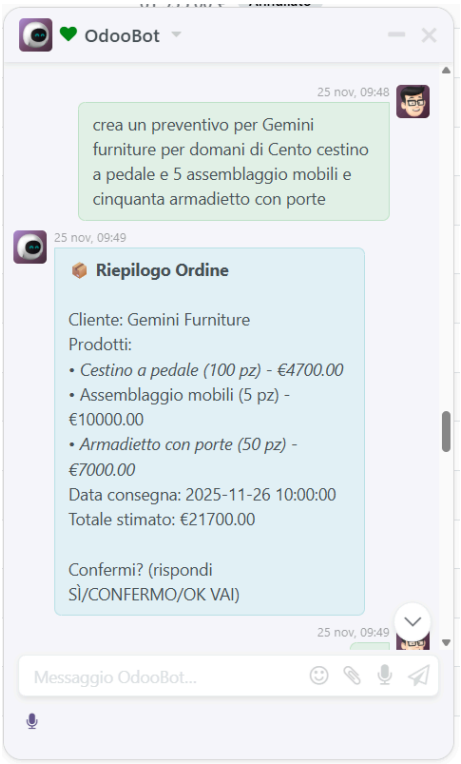


Figura 7: Esempio di creazione ordine tramite chat.

Descrizione: Come agente, voglio poter creare una bozza di ordine specificando cliente e prodotti in un unico messaggio, per velocizzare l'inserimento dati.

Acceptance criteria:
1. L'utente scrive una richiesta complessa (es. «Crea un ordine per Azure Interior per 5 Sedie da ufficio»).
2. Il sistema estrae il cliente e i prodotti dalla frase.
3. Il sistema verifica che i prodotti siano disponibili.

4. Il sistema prepara i dati per l'ordine.

US4 Conferma e Inserimento

Il flusso di conferma e inserimento è illustrato in Tabella 3.

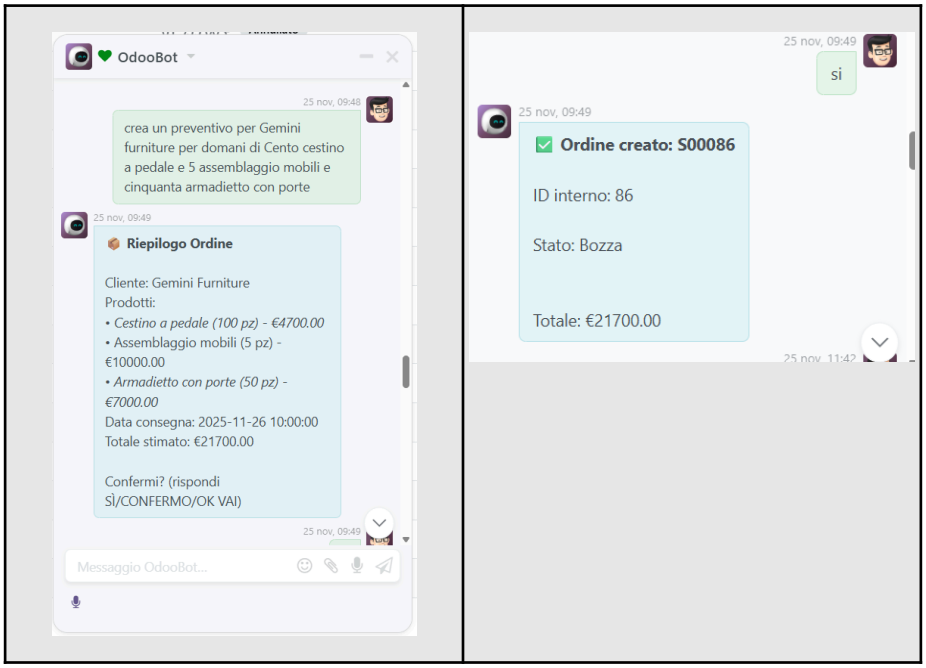


Tabella 3: Conferma e inserimento ordine tramite chat.

Descrizione: Come operatore, voglio che il sistema mi chieda conferma prima di salvare un ordine, per evitare errori dovuti a incomprensioni dell'AI.

Acceptance criteria:
1. Il sistema mostra un riepilogo dell'ordine che sta per creare (Cliente, Prodotti, Quantità, Totale).
2. L'utente deve rispondere affermativamente (es. «Sì, procedi»).
3. Solo dopo la conferma, il sistema crea effettivamente il record nel database e restituisce il link all'ordine creato.

3.3 Tracciamento dei requisiti

I requisiti sono stati classificati secondo la seguente notazione:

$$\mathbf{(F/Q/C)(M/D/O)R}$$

Dove:

- **Tipo**: F (Funzionale), Q (Qualitativo), C (Vincolo).
- **Priorità**: M (Mandatory - Obbligatorio), D (Desirable - Desiderabile), O (Optional - Facoltativo).

Nelle tabelle seguenti (Tabella 4, Tabella 5, Tabella 6) è riportato il tracciamento puntuale dei requisiti identificati.

Codice	**Descrizione**	**Fonti**
FMR1	Il sistema deve permettere all'utente di interagire tramite linguaggio naturale utilizzando la chat nativa di Odoo.	US1
FMR2	Il sistema deve essere in grado di interpretare richieste relative alla disponibilità di prodotti in magazzino.	US2
FMR3	Il sistema deve interrogare il database di Odoo per recuperare le giacenze aggiornate dei prodotti richiesti.	US2
FMR4	Il sistema deve permettere la creazione di una bozza di ordine di vendita a partire dalla richiesta dell'utente.	US3
FMR5	Il sistema deve richiedere conferma all'utente prima di procedere con l'inserimento effettivo dell'ordine.	US4
FMR6	Il sistema deve validare la disponibilità dei prodotti prima di creare l'ordine.	US3
FOR1	Il sistema deve supportare l'input vocale (Speech-to-Text) per l'inserimento dei comandi.	Piano di lavoro (F01)

Tabella 4: Tracciamento dei requisiti funzionali.

Codice	Descrizione	Fonti
QMR1	L'interfaccia utente deve essere integrata nativamente nel modulo Discuss di Odoo senza richiedere finestre esterne.	Piano di lavoro.
QMR2	Il sistema deve gestire eventuali «allucinazioni» dell'LLM validando sempre l'output strutturato (JSON) ricevuto.	Analisi dei rischi.
QDR1	Il tempo di risposta tra la richiesta dell'utente e la risposta del bot deve essere inferiore a 5 secondi in condizioni di rete ottimali.	Interno.

Tabella 5: Tracciamento dei requisiti qualitativi.

Codice	Descrizione	Fonti
CMR1	Il modulo deve essere sviluppato per la versione 18 della piattaforma Odoo.	Piano di lavoro.
CMR2	Il linguaggio di programmazione utilizzato per il backend deve essere Python.	Piano di lavoro.
CMR3	L'integrazione AI deve utilizzare API di Large Language Models (es. Google Gemini o OpenAI).	Piano di lavoro.

Tabella 6: Tracciamento dei requisiti di vincolo.

La Tabella 7 riassume la distribuzione dei requisiti per tipologia e priorità.

Tipo	Mandatory	Desirable	Optional	Somma
Functional	6	0	1	7
Qualitative	2	1	0	3
Constraint	3	0	0	3
Totale	**11**	**1**	**1**	**13**

Tabella 7: Riepilogo dei requisiti.

Capitolo 4

Tecnologie e strumenti

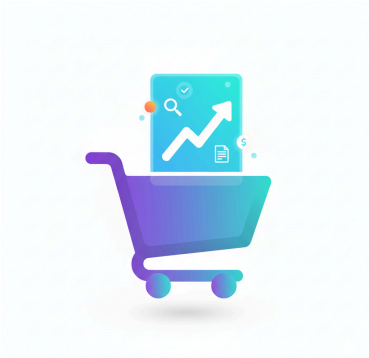


Figura 8: Logo di AI LiveBot

In questo capitolo vengono illustrate le tecnologie e gli strumenti utilizzati per la realizzazione del progetto **AI LiveBot**. La scelta di tali tecnologie è stata guidata dai vincoli di progetto e dalla necessità di

integrare funzionalità avanzate di Intelligenza Artificiale all'interno di un ecosistema ERP consolidato.

4.1 Odoo ERP

Odoo [2] è una suite di applicazioni aziendali open source che copre tutte le esigenze di un'azienda: Customer Relationship Management (CRM), e-commerce, contabilità, inventario, punto vendita, gestione progetti, ecc. La versione utilizzata per questo progetto è la **Odoo 18**.

4.1.1 Architettura

Odoo segue un'architettura **multitenant** a tre livelli (come mostrato in Figura 9):

- **Presentation Tier**: Interfaccia web basata su HTML5, JS e CSS3 (framework Odoo Web Library (OWL) [3]).
- **Logic Tier**: Server applicativo scritto in Python.
- **Data Tier**: Database relazionale PostgreSQL [4].

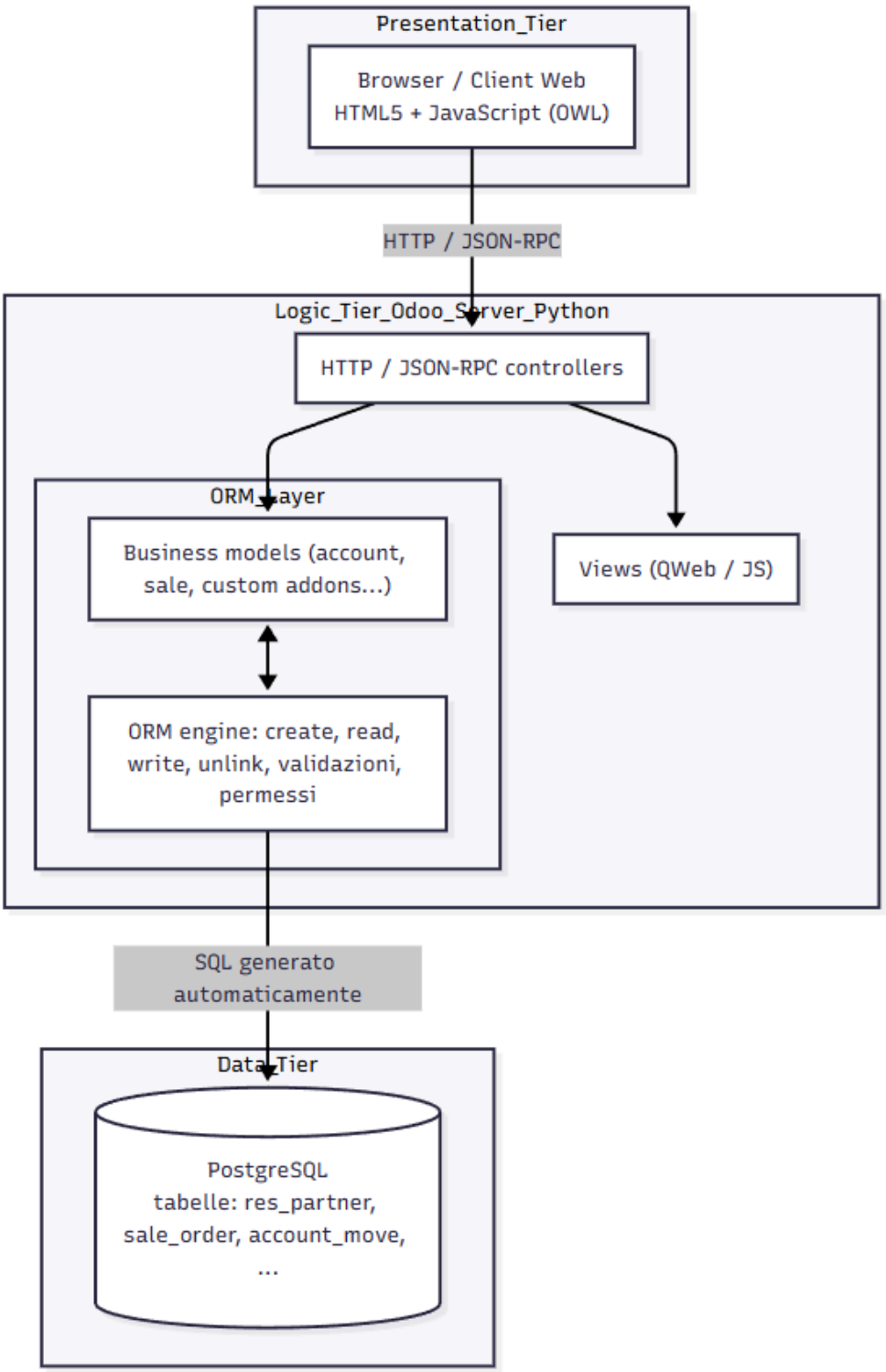


Figura 9: Schema Architettura Odoo

La modularità è il concetto chiave di Odoo: ogni funzionalità è incapsulata in un **modulo** (o **addon**) che può estendere o modificare il comportamento degli altri moduli esistenti senza alterarne il codice sorgente originale, grazie al meccanismo dell'ereditarietà. La Figura 10 mostra la struttura tipica di un addon Odoo.

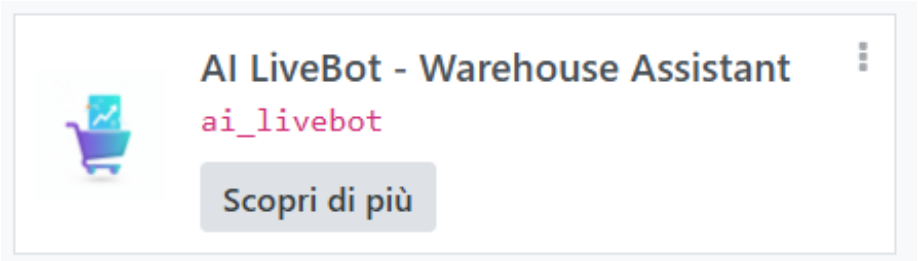


Figura 10: Struttura di un modulo Odoo

4.1.2 ORM (Object-Relational Mapping)

Odoo utilizza un proprio Object-Relational Mapping (ORM) che astrae le interazioni con il database PostgreSQL. Questo permette agli sviluppatori di interagire con i dati utilizzando oggetti Python, rendendo il codice più leggibile e sicuro rispetto all'uso diretto di query SQL.

4.1.3 Punti di forza e limitazioni

L'utilizzo di Odoo è stato un **vincolo di progetto**, in quanto l'azienda ospitante opera esclusivamente su questa piattaforma per i propri clienti. Ciò nonostante, la scelta si è rivelata coerente con gli obiettivi del progetto grazie ai numerosi punti di forza della piattaforma.
Tra i principali **punti di forza** di Odoo si annoverano:

- **Modularità**: ogni funzionalità è incapsulata in moduli indipendenti, facilmente estendibili senza modificare il codice sorgente originale;
- **Copertura funzionale**: un unico sistema copre CRM, vendite, magazzino, contabilità, e-commerce e molto altro;
- **Community attiva**: ampia documentazione, forum e marketplace di moduli di terze parti;
- **Licenza open source** (Community Edition): permette di ispezionare il codice e personalizzarlo liberamente.

Tra le **limitazioni** principali:

- **Curva di apprendimento**: l'architettura a ereditarietà multipla e il framework OWL richiedono tempo per essere padroneggiati;
- **Prestazioni**: su installazioni molto grandi o con moduli mal ottimizzati, le performance possono degradare;

- **Dipendenza da PostgreSQL**: non è possibile utilizzare altri Database Management System (DBMS).

4.2 Large Language Models (LLM)

I **Large Language Models** (LLM) sono modelli di deep learning addestrati su immense quantità di testo, capaci di comprendere e generare linguaggio naturale.

4.2.1 Google Gemini

Per questo progetto è stato scelto il modello **Google Gemini** [1] (nella sua versione Pro accessibile via Application Programming Interface (API)). Le motivazioni della scelta includono:
- **Finestra di contesto**: Ampia capacità di gestire input lunghi.
- **Multimodalità**: Capacità nativa di comprendere testo e (potenzialmente) immagini.
- **JSON Mode**: Supporto ottimizzato per generare output strutturati in formato JavaScript Object Notation (JSON), fondamentale per l'integrazione con il software gestionale.

4.2.1.1 Alternative considerate

Il panorama dei Large Language Models offre diverse opzioni:
- **OpenAI GPT-4 / GPT-4o** [5]: eccellente qualità di ragionamento e ampia adozione, ma costi per Token (Token) più elevati e, al momento della scelta, politiche di utilizzo più restrittive per alcuni mercati;
- **Anthropic Claude 3** [6]: ottimo per contesti lunghi e ragionamento step-by-step, ma disponibilità API inizialmente limitata in Europa;
- **Modelli open-source (LLaMA 3, Mistral, Qwen)**: eseguibili on-premise senza costi di API, ma richiedono infrastruttura Graphics Processing Unit (GPU) dedicata e tuning per raggiungere prestazioni comparabili ai modelli proprietari.

Gemini è stato preferito per il rapporto qualità/prezzo, la disponibilità di un piano gratuito per lo sviluppo, la buona documentazione e il supporto nativo al JSON Mode, che semplifica l'integrazione con il Domain Specific Language (DSL) di funzioni del modulo.

4.2.2 Prompt Engineering

Il **Prompt Engineering** [7] è la disciplina che si occupa di progettare e ottimizzare gli input (prompt) forniti ai modelli LLM per ottenere i risultati desiderati. Nel contesto di AI LiveBot, questa tecnica è stata

cruciale per istruire il modello su come interpretare le richieste degli utenti e mapparle sulle funzioni di Odoo.
Nel modulo sviluppato il prompt non contiene solo istruzioni generali sul ruolo dell'assistente (venditore esperto Odoo), ma anche la descrizione del **linguaggio di funzioni** che il modello deve usare per interagire con il gestionale. Invece di generare direttamente codice o query SQL, l'LLM è invitato a produrre tag testuali del tipo `[FUNCTION:nome_funzione|parametro:valore|...]`, che vengono poi interpretati dal backend e mappati su metodi Python sicuri.

4.2.3 Allucinazioni negli LLM

Uno dei fenomeni più critici e studiati nell'ambito dei Large Language Models è quello delle *allucinazioni*: output che appaiono fluenti e coerenti ma sono fattualmente scorretti, non verificabili o logicamente inconsistenti [8]. Il termine è mutuato dalla psicologia, dove indica una percezione reale in assenza di uno stimolo esterno appropriato; in modo analogo, il modello genera testo che *sembra* fondato ma non lo è.
Formalmente, il fenomeno emerge quando il modello assegna probabilità più alta a una sequenza errata rispetto a quella corretta:

$$P_\theta(y_{\text{hallucinated}} \mid x) > P_\theta(y_{\text{grounded}} \mid x)$$

Questo riflette un conflitto strutturale tra l'ottimizzazione della fluenza obiettivo primario del training e l'ancoraggio fattuale, che non è mai ottimizzato direttamente [9].
Tassonomia. La letteratura distingue principalmente quattro tipologie [8]:

- **Intrinseche**: il testo generato contraddice direttamente l'input o il contesto fornito (es. riassumere un documento con fatti opposti a quelli presenti).
- **Estrinseche**: il modello aggiunge informazioni non presenti nella fonte e non immediatamente verificabili, spesso plausibili ma non dimostrabili.
- **Fattuali**: affermazioni false rispetto alla conoscenza del mondo reale (es. capitali sbagliate, date errate, prodotti inesistenti).
- **Logiche**: ragionamenti internamente incoerenti, nonostante la correttezza grammaticale.

Cause. Le origini delle allucinazioni si collocano a due livelli [8]. A livello di *dati*, la causa principale è la divergenza source-reference nei dataset di addestramento: quando il corpus contiene coppie input-output in cui il target include informazioni non supportate dalla fonte, il modello apprende a generare «fuori contesto». A livello di *training e inferenza*, contribuiscono l'*exposure bias* durante l'inferenza il modello si condiziona sulle proprie predizioni precedenti anziché sul ground truth

e il *parametric knowledge bias*, per cui i modelli più grandi tendono a privilegiare la conoscenza memorizzata durante il pretraining rispetto al contesto fornito nel prompt.
Un framework di analisi sistematica è proposto da [9], che distingue due dimensioni ortogonali: la **Prompt Sensitivity** (PS), che misura quanto il tasso di allucinazione varia al variare della formulazione del prompt a modello fisso, e la **Model Variability** (MV), che misura la variazione tra modelli diversi a prompt fisso. Dalla combinazione delle due metriche emergono quattro categorie: allucinazioni *prompt-dominant* (alta PS, bassa MV), *model-dominant* (bassa PS, alta MV), *mixed-origin* e *unclassified*. Questa distinzione è rilevante perché le strategie di mitigazione più efficaci differiscono a seconda della categoria: le allucinazioni prompt-dominant sono in principio correggibili senza modificare il modello, mentre quelle model-dominant richiedono interventi architetturali o di fine-tuning.
Strategie di mitigazione. Gli approcci più consolidati operano su tre livelli [8], [9]:

- **Prompt-based**: tecniche come il *Chain-of-Thought* (Chain-of-Thought (CoT)) che guida il modello a esplicitare il ragionamento passo per passo e la specifica esplicita dei vincoli nel prompt riducono significativamente le allucinazioni prompt-indotte. Il CoT è particolarmente efficace sui modelli con alta PS, ma può amplificare l'errore quando la conoscenza interna è già lacunosa, producendo ragionamenti elaborati ma comunque errati.
- **Architetturale**: il *Retrieval-Augmented Generation* (Retrieval-Augmented Generation (RAG)) introduce una fonte esterna aggiornata durante l'inferenza, riducendo la dipendenza dalla memoria parametrica. Il *Function Calling* o DSL di funzioni vincola le azioni del modello a un insieme predefinito di operazioni, impedendo risposte non ancorate a dati verificati.
- **Validazione esterna**: la verifica lato backend dei parametri generati dal modello consente di intercettare e correggere inconsistenze prima che producano effetti concreti sul sistema.

Vale la pena osservare che nessun approccio elimina completamente il problema: i modelli più recenti mostrano tassi di allucinazione significativamente ridotti rispetto alle generazioni precedenti, ma spostano il fenomeno verso forme più sottili e difficili da rilevare automaticamente. Le implicazioni pratiche di queste strategie per il sistema sviluppato in questo elaborato sono discusse in Sezione 5.3.1.

4.2.3.1 Impatto dei token su costi e prestazioni

L'utilizzo di modelli LLM tramite API introduce un vincolo pratico legato al numero di Token scambiati per ogni chiamata. Come illustrato

in Figura 11, ogni richiesta include sia il **system prompt** sia lo storico della conversazione che si decide di mantenere come contesto; a sua volta, la risposta del modello contribuisce al conteggio complessivo. All'aumentare dei Token utilizzati cresce sia il costo economico della chiamata sia, in generale, la latenza di risposta.

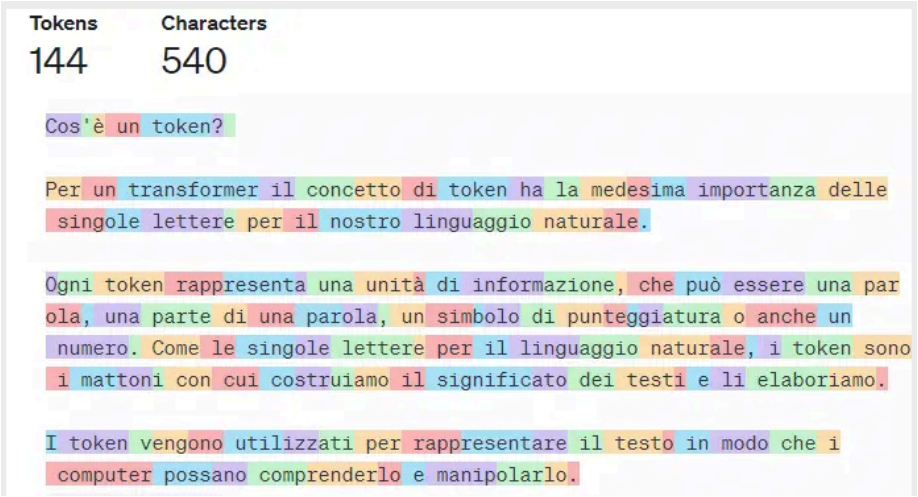


Figura 11: Rappresentazione dei token nei modelli LLM

Nel caso di **AI LiveBot** il **system prompt** svolge anche il ruolo di «wrapper» intorno al modello grezzo. A differenza delle interfacce web dei principali provider, dove l'utente interagisce con un chatbot già incapsulato in istruzioni di alto livello gestite dal fornitore, qui l'applicazione chiama direttamente l'endpoint del modello e deve quindi esplicitare nel prompt tutte le regole comportamentali (ruolo dell'assistente, formato delle risposte, uso del DSL di funzioni, vincoli operativi). Questo porta inevitabilmente a un prompt più verboso, che però è necessario per rendere l'assistente realmente utile e affidabile nel contesto ERP.
Per contenere l'impatto di questo overhead si è cercato comunque di:

- mantenere il **system prompt** il più compatto possibile, spostando nel codice backend la logica che può essere implementata deterministicamente (ad esempio, la normalizzazione dei termini di ricerca prodotti o la classificazione dell'intento dell'utente);
- limitare lo storico della conversazione inviato all'LLM alle sole parti davvero rilevanti per la richiesta corrente, evitando di passare sempre l'intera cronologia della chat;
- utilizzare, quando possibile, chiamate «task-specifiche» (brevi prompt dedicati a un singolo compito) al posto di un unico prompt generalista molto lungo.

In assenza di tali accorgimenti, l'accumulo progressivo dello storico e di istruzioni ripetitive nel prompt porterebbe a richieste via via più pesanti (maggiore numero di token), con un aumento sensibile dei costi di utilizzo del servizio e il rischio di avvicinarsi ai limiti massimi di contesto supportati dal modello.
Durante una sessione di test intensivo di circa sei ore, utilizzando il modello **Gemini 2.5 Flash** in chiamata diretta via API, sono state effettuate 307 richieste con un consumo complessivo di circa 5,6 milioni di token di input. Questo corrisponde a una media di circa 18–20k token di input per singola chiamata. Solo una piccola parte di questi token (nell'ordine di 100–200) è costituita dalla domanda dell'utente; la parte rimanente è dovuta al **system prompt** e allo storico conversazionale che vengono ripassati al modello ad ogni invocazione. Le risposte generate dal modello, essendo per lo più brevi messaggi informativi o riepiloghi di ordini, hanno una lunghezza significativamente inferiore (in genere alcune centinaia di token, raramente oltre 1.000). Possiamo quindi stimare un costo medio per richiesta nell'ordine di 20.000 token complessivi (input + output), pari a circa 6 milioni di token per una sessione di questo tipo. Questo valore è sensibilmente superiore a quello di un uso puramente informazionale del modello (domande e risposte puntuali), ma è coerente con uno scenario in cui l'LLM mantiene un contesto ricco e orchestra workflow ERP completi lungo più turni di conversazione. In un contesto produttivo, tali numeri rendono importante monitorare e ottimizzare il consumo di token, ad esempio riducendo lo storico, comprimendo il system prompt e riservando le funzionalità più costose ai casi d'uso che ne traggono il maggiore beneficio.

4.3 Linguaggi di programmazione

4.3.1 Python

Python è il linguaggio principale per lo sviluppo backend in Odoo. La sua sintassi chiara e la vasta disponibilità di librerie (come `requests` per le chiamate API o `json` per il parsing) lo rendono ideale per l'integrazione di servizi esterni.
Punti di forza: leggibilità, ecosistema di librerie vastissimo (AI/ML, web, data processing), community attiva e ampia documentazione.
Limitazioni: performance inferiori rispetto a linguaggi compilati; il Global Interpreter Lock (Global Interpreter Lock (GIL)) può limitare il parallelismo CPU-bound.
Alternative: nel contesto Odoo non esistono alternative reali, poiché l'intero framework è scritto in Python. Per integrazioni esterne si sarebbero potuti usare microservizi in Go o Rust per le parti ad alta intensità

computazionale, ma la complessità aggiuntiva non era giustificata per questo progetto.

4.3.2 XML e XPath

In Odoo, l'interfaccia utente (Viste) e la dichiarazione dei dati sono definite tramite file XML. XPath viene utilizzato per localizzare elementi specifici all'interno delle viste esistenti ed estenderle (es. aggiungere un bottone in una form view standard).
Nel caso di AI LiveBot, le viste XML sono state utilizzate per estendere la chat di Discuss ed aggiungere elementi di interfaccia dedicati, come il pulsante per la dettatura vocale che sfrutta le API del browser.

4.4 Strumenti di sviluppo

4.4.1 Visual Studio Code

L'IDE (Integrated Development Environment (IDE)) scelto per lo sviluppo è Visual Studio Code, grazie alla sua leggerezza e all'ecosistema di estensioni per Python e Odoo.

4.4.2 Git

Per il controllo di versione è stato utilizzato **Git**. Il flusso di lavoro ha previsto l'uso di branch separati per le diverse funzionalità (feature branch workflow) per mantenere pulito il ramo principale del progetto.

4.4.3 Web Speech API e supporto Speech-to-Text

Per abilitare l'inserimento vocale dei comandi nella chat di Odoo è stato sfruttato il supporto alle **Web Speech API** [10] offerto dai browser moderni (in particolare Google Chrome e Microsoft Edge). L'idea è quella di mantenere la logica di riconoscimento vocale (STT) interamente sul lato client:

- un piccolo script JavaScript rileva il clic sull'icona del microfono nella barra di input della chat;
- il browser avvia il riconoscimento vocale e trascrive in tempo reale il parlato in testo;
- al termine della dettatura, il testo viene semplicemente inserito nel campo di input, che può essere inviato all'assistente come un normale messaggio.

Questa scelta architetturale ha due vantaggi principali: non richiede di integrare servizi esterni di Speech-to-Text lato server e consente di sfrut-

tare direttamente le capacità native del browser, a patto di rispettare alcuni vincoli (contesto sicuro HTTP Secure (HTTPS) o `localhost` e utilizzo di browser compatibili).
Punti di forza: nessun costo aggiuntivo, bassa latenza, nessuna dipendenza da servizi cloud esterni, privacy (l'audio non lascia il dispositivo nel caso di riconoscimento on-device).
Limitazioni: supporto non uniforme tra browser (Firefox ha un'implementazione parziale, Safari limitata); richiede contesto sicuro (HTTPS); qualità del riconoscimento dipendente dall'implementazione del browser.
Alternative considerate:

- **Google Cloud Speech-to-Text** e **Azure Speech Services**: alta qualità e supporto multilingua, ma introducono costi per utilizzo e latenza di rete;
- **Whisper (OpenAI)**: modello open-source eseguibile on-premise, ottima qualità, ma richiede risorse server (CPU/GPU) e aggiunge complessità infrastrutturale.

La Web Speech API è stata scelta per la semplicità di integrazione e l'assenza di costi, accettando il vincolo di compatibilità browser come compromesso ragionevole per un prototipo dimostrativo.

Capitolo 5

Progettazione e sviluppo

In questo capitolo viene descritta l'attività di progettazione e implementazione del modulo **AI LiveBot**. Vengono analizzate le scelte architetturali, le problematiche riscontrate durante lo sviluppo e le soluzioni adottate per superarle.

5.1 Architettura della soluzione

Il modulo **AI LiveBot** è stato progettato come un'estensione del modulo nativo `mail` (Discuss) di Odoo. L'obiettivo era intercettare i messaggi

inviati dagli utenti nella chat e, se necessario, inoltrarli all'LLM per l'elaborazione.
L'architettura logica prevede i seguenti componenti principali:

1. **Odoo Channel Listener**: un override del metodo `message_post` del modello `discuss.channel` che intercetta i messaggi in ingresso nella chat AI, filtra quelli provenienti dall'utente e attiva il flusso di elaborazione.
2. **AI Controller**: un servizio interno (definito nel modello `discuss.channel`) che incapsula la logica di chiamata ai provider LLM configurati (Google Gemini o OpenRouter), gestendo anche retry, errori temporanei e sicurezza.
3. **Function DSL & Prompt Manager**: un piccolo linguaggio testuale basato su tag, tramite cui l'LLM richiede l'esecuzione di funzioni applicative (es. `[FUNCTION:search_products|search_term:...|limit:...]`). Questa astrazione permette di separare la parte «ragionamento» del modello dalla parte di accesso ai dati Odoo.
4. **Warehouse Operations Service**: il modello astratto `warehouse.operations` che espone metodi di dominio ad alto livello (ricerca prodotti, gestione ordini, delivery, statistiche) invocati dal **Function DSL**.
5. **AI Configuration**: il modello `ai.config`, attraverso cui l'amministratore definisce profili attivi (provider, chiave API, modello, temperatura), garantendo che l'integrazione sia configurabile senza modifiche al codice.

La comunicazione avviene sempre nel seguente ordine:

- l'utente invia un messaggio (testo o dettatura vocale) nella chat di Discuss;
- il **Channel Listener** decide se si tratta di una nuova richiesta o della conferma di un'azione già proposta;
- in caso di nuova richiesta, costruisce i messaggi da inviare all'LLM e ne riceve una risposta testuale contenente, eventualmente, uno o più tag `[FUNCTION:...]`;
- il **Function DSL** estrae i tag, associa ogni funzione a un metodo di `warehouse.operations` e ne raccoglie i risultati;
- infine una seconda chiamata LLM (o una formattazione diretta) produce una risposta leggibile, che viene inviata all'utente in forma HTML formattato.

5.1.1 Pipeline completa di un messaggio

Il flusso completo di elaborazione di un messaggio può essere riassunto come segue:

1. **Input utente**: l'operatore scrive (o detta, tramite la funzionalità **Speech-to-Text**) una richiesta nella chat AI di Discuss.
2. **Intercettazione**: l'override di `message_post` su `discuss.channel` rimuove l'HTML, identifica il mittente e scarta i messaggi generati dal sistema o dal bot stesso.
3. **Gestione conferme**: se il testo corrisponde a una conferma esplicita (es. «Sì», «CONFERMO», «OK VAI»), il sistema verifica se esistono azioni pendenti salvate nel campo `pending_action_data` del canale (per esempio un ordine da creare o un ordine da cancellare) ed esegue direttamente il metodo corrispondente di `warehouse.operations` senza richiamare l'LLM.
4. **Richiesta AI**: se non ci sono azioni pendenti, viene costruito un messaggio arricchito con il contesto necessario (ad esempio il contenuto dell'ultimo ordine in bozza se l'utente chiede di «modificare il preventivo»), che viene passato a `_get_gemini_response`, il quale a sua volta delega a `_call_gemini` o `_call_openrouter` a seconda del provider configurato.
5. **Interpretazione funzioni**: la risposta dell'LLM può contenere uno o più tag `[FUNCTION:...]` che specificano quale funzione applicativa invocare e con quali parametri. Il parser `_parse_ai_function_calls` estrae queste chiamate e `_execute_function` le inoltra ai metodi di `warehouse.operations`.
6. **Sintesi della risposta**: i risultati delle funzioni (tipicamente in formato JSON) vengono sintetizzati in un testo comprensibile. Per i casi più complessi, viene effettuata una seconda chiamata LLM con i risultati in input e una richiesta del tipo «riassumi per l'utente in modo chiaro e professionale».
7. **Formattazione HTML**: prima di inviare la risposta in chat, la funzione `format_html_response` si occupa di trasformare il testo in HTML (gestendo liste, evidenziazioni, codici ordine, ecc.), in modo che l'utente riceva messaggi leggibili e coerenti con lo stile di Discuss.

5.2 Integrazione con l'LLM

L'integrazione con i modelli LLM avviene tramite chiamate Representational State Transfer (REST) API ai provider supportati (Google Gemini e OpenRouter). La configurazione è centralizzata nel modello `ai.config`, che consente di definire uno o più profili con:

- provider (`Gemini` o `OpenRouter`),
- modello da utilizzare (es. `gemini-2.5-flash`),
- chiavi API,
- parametri di generazione (temperatura, max Token).

Il metodo `_get_gemini_response` funge da **dispatcher** verso il provider scelto, gestendo anche casi tipici di errore (es. Hypertext Transfer Protocol (HTTP) 503, 429) con retry automatici e messaggi user-friendly verso l'utente.

5.2.1 Costruzione del prompt e DSL di funzioni

Una delle sfide principali è stata definire un **System Prompt** robusto, in grado di:

- istruire il modello ad agire come un assistente di vendita esperto in Odoo;
- limitare le **allucinazioni** [11] imponendo che le operazioni sul database passino sempre attraverso funzioni esplicite;
- separare la fase di comprensione del linguaggio naturale dalla fase di esecuzione effettiva.

Per questo è stato introdotto un piccolo **Domain Specific Language** (DSL) testuale basato su tag di funzione. L'LLM, invece di generare direttamente SQL o Python, produce frammenti come mostrato nel codice Codice 2.

```
1 [FUNCTION:search_products|search_term:
  "sedie ufficio"|limit: 5]
```

Plain Text

```
2 [FUNCTION:create_sales_order|partner_name:"Azure
  Interior"|order_lines:
  [{"product_id":38,"quantity":5}]|confirm:true]
```

Codice 2: Esempio di tag DSL `[FUNCTION:...]` generati dall'LLM.

Questi tag vengono estratti dal parser `_parse_ai_function_calls`, che ricostruisce il nome della funzione (`search_products`, `create_sales_order`, ecc.) e un dizionario di parametri. `_execute_function` si occupa poi di mappare ogni nome funzione su un metodo specifico di `warehouse.operations`.
In questo modo il modello si limita a decidere **cosa** fare (quali funzioni chiamare e con che parametri), mentre la logica di **come** farlo (domini Odoo, controlli di stato, transazioni) rimane completamente nel backend.

5.2.2 Ottimizzazione del contesto e uso dei token

Un aspetto progettuale rilevante riguarda il modo in cui viene gestito il contesto passato all'LLM ad ogni invocazione. Un approccio ingenuo, che preveda di includere sempre l'intera cronologia della chat e un **system prompt** molto verboso, porta nel tempo ad un aumento lineare

del numero di token per richiesta: ogni nuovo messaggio include tutti i messaggi precedenti, più istruzioni ripetute e potenzialmente ridondanti. Nel caso specifico di **AI LiveBot** il **system prompt** non può essere ridotto a poche righe, perché deve sostituirsi alle istruzioni «nascoste» che normalmente incapsulano i modelli all'interno delle interfacce chat dei provider (ruolo dell'agente, limiti operativi, formato delle funzioni da usare, stile delle risposte, ecc.). Nonostante questo vincolo, sono state adottate alcune strategie per evitare sprechi di token:

- mantenere il **system prompt** concentrato sulle regole essenziali (ruolo dell'assistente, uso del DSL di funzioni, vincoli di sicurezza) evitando ripetizioni inutili;
- affiancarlo a prompt «mirati» per compiti specifici (ad esempio la normalizzazione dei termini di ricerca o la classificazione dell'intento), che possono essere chiamati separatamente e con uno storico minimo;
- inserire nel contesto solo le informazioni strettamente necessarie per la richiesta corrente (ad esempio l'ultimo ordine in bozza quando l'utente chiede di «modificare il preventivo»), anziché tutta la storia della conversazione.

Queste scelte riducono lo spreco di token e i costi economici associati alle chiamate API, mantenendo al tempo stesso la qualità delle risposte. Inoltre, il fatto di delegare al backend il salvataggio di uno stato strutturato (come `pending_action_data` per le azioni in attesa di conferma) evita di dover ripassare ogni volta al modello l'intera sequenza di decisioni pregresse, contribuendo ulteriormente a contenere il contesto testuale necessario.

5.3 Problematiche e Soluzioni

5.3.1 Gestione delle Allucinazioni

Come illustrato in Sezione 4.2.3, le allucinazioni rappresentano una criticità strutturale dei modelli linguistici di grandi dimensioni. In fase di sperimentazione, il modello tendeva talvolta a inventare nomi di prodotti o ad accettare ordini per quantità non disponibili in magazzino. Per mitigare questo problema è stata adottata una strategia su più livelli:

1. **Prompt Constraint**: nel prompt vengono fornite al modello solo le informazioni strettamente necessarie (ad esempio l'elenco dei prodotti più rilevanti per la richiesta), esplicitando che non deve creare prodotti inesistenti.
2. **Function DSL**: l'LLM non interagisce mai direttamente con il database, ma può solo richiedere l'esecuzione di funzioni predefinite (`search_products`, `get_stock_info`, `create_sales_order`, ecc.).

In questo modo si può controllare con precisione quali operazioni sono ammesse.

3. **Backend Validation**: prima di creare o modificare un ordine, il backend verifica sempre la coerenza dei parametri (esistenza dei prodotti, stato dell'ordine, disponibilità di magazzino) e, in caso di problemi, restituisce un errore strutturato che viene poi trasformato in un messaggio comprensibile per l'utente.

5.3.2 Formattazione dell'Output

Un altro problema emerso riguarda la mescolanza di testo discorsivo e blocchi strutturati (JSON) nelle risposte dell'LLM, che rendeva fragile il parsing lato backend. Per affrontarlo sono state adottate due strategie:

- utilizzo, dove possibile, delle modalità di generazione strutturata offerte dai provider (ad esempio il **JSON Mode** di Gemini);
- implementazione di un parser robusto in grado di estrarre il blocco JSON anche quando incapsulato in testo descrittivo, e di rimuovere eventuali tag `[FUNCTION:...]` o marker interni prima della visualizzazione.

Infine, la funzione `format_html_response` si occupa di mappare convenzioni testuali (bullet list, evidenziazione di codici ordine, simboli di stato) in HTML compatibile con la finestra di Discuss, migliorando la leggibilità senza modificare la struttura dei dati sottostante.

5.4 Implementazione delle funzionalità principali

5.4.1 Ricerca Prodotti

Quando l'utente chiede, ad esempio, «Avete sedie?», il sistema segue i seguenti passi:

1. Invia la query all'LLM, che riconosce l'intento di ricerca prodotti e genera un tag del tipo `[FUNCTION:search_products|search_term: "sedie"|limit: 50]`.
2. Il backend esegue la funzione `search_products`, che applica una ricerca **fuzzy** multi-pattern sul modello `product.product` (match esatto, match su tutte le parole chiave, match su almeno una parola), restituendo una lista di prodotti ordinati per rilevanza.
3. I risultati vengono sintetizzati in una risposta naturale (ad esempio elenchi puntati con nome prodotto, prezzo e giacenza) e mostrati in chat.
4. Se la richiesta dell'utente è ambigua (più prodotti possibili), il bot chiede un chiarimento, come descritto nel manuale utente.

5.4.2 Creazione e gestione degli ordini

Per la creazione di preventivi e ordini di vendita il flusso è il seguente:

1. L'utente scrive, ad esempio: «crea un preventivo per Azure Interior per 5 sedie ufficio e 2 tavoli».
2. L'LLM estrae le entità rilevanti (`partner_name`, prodotti, quantità) e, dopo eventuali ricerche prodotti, genera un tag `create_sales_order` con parametri strutturati (lista di `product_id` e quantità).
3. Il backend non esegue immediatamente la funzione: costruisce un riepilogo leggibile (cliente, righe, totale stimato, eventuale data di consegna) e memorizza i parametri nel campo `pending_action_data` del canale, apponendo un marker interno.
4. Solo se l'utente conferma esplicitamente (es. «SÌ», «CONFERMO»), il sistema recupera i parametri dal marker ed esegue effettivamente `create_sales_order` o `cancel_sales_order` tramite `warehouse.operations`.
5. La stessa logica viene applicata alle richieste di annullamento di preventivi/ordini, riducendo il rischio di cancellazioni accidentali dovute a malintesi dell'LLM.

Per le modifiche ad ordini in bozza viene utilizzata la funzione `update_sales_order`, che permette di cambiare quantità, aggiungere o rimuovere righe e aggiornare la data di consegna. Per ordini già confermati è stata prevista una funzione dedicata (`update_confirmed_sales_order`) che, in modo controllato, annulla gli eventuali delivery non ancora evasi, riporta l'ordine in bozza, applica le modifiche e lascia al magazzino la gestione dei nuovi trasferimenti.

5.5 Valutazione sperimentale

In questa sezione viene presentata una valutazione quantitativa del modulo **AI LiveBot**, con l'obiettivo di misurare in modo sistematico la qualità delle risposte dell'assistente e l'efficienza temporale delle chiamate ai modelli linguistici (LLM) integrati in Odoo 18. L'attenzione è rivolta in particolare a due aspetti: la capacità del modello di scegliere correttamente la funzione di backend da invocare (**tool choice**) e la qualità del ranking dei prodotti restituiti dalla funzione `search_products` in scenari tipici di gestione del magazzino.

Per la valutazione sono state adottate metriche standard di **Information Retrieval** (Information Retrieval (IR)) e di valutazione di sistemi conversazionali basati su LLM: **Tool Choice Accuracy** (Tool Choice Accuracy (TCA)), Hit@K (Hit@K (Hit@K)), Precision@K (Precision@K (Precision@K)), Recall@K (Recall@K (Recall@K)), **Mean Reciprocal**

Rank (Mean Reciprocal Rank (MRR)) e misure di latenza (percentili 50° percentile (P50) e 95° percentile (P95)). Ciò consente di analizzare in modo complementare (i) la correttezza della scelta della funzione applicativa, (ii) la qualità del ranking dei prodotti e (iii) l'esperienza d'uso dal punto di vista dei tempi di risposta percepiti.
I test sono stati condotti confrontando cinque modelli LLM, mantenendo invariato lo strato applicativo (prompt, DSL dei tag `[FUNCTION:...]`, logica Odoo) e variando unicamente il backend. In questo modo le differenze osservate possono essere attribuite ai modelli e non all'orchestratore. Il dataset di prova comprende 7 query rappresentative (ricerca prodotti e controllo stock), di cui 6 con un insieme noto di prodotti attesi (`expected_product_ids`). Il database contiene 43 prodotti complessivi, con un livello di ambiguità sufficiente a mettere in evidenza differenze tra i modelli.

5.5.1 Metriche adottate

La **Tool Choice Accuracy** misura la percentuale di volte in cui l'LLM seleziona la funzione corretta del backend (ad esempio `search_products` invece di `get_stock_info`) per una data richiesta utente. È definita come rapporto tra il numero di query in cui la funzione scelta coincide con quella annotata come corretta e il numero totale di query considerate. Questa metrica è cruciale perché un errore nella scelta dello strumento rende irrilevante anche un eventuale buon ranking successivo.
La metrica Hit@K indica la percentuale di query per cui almeno un prodotto atteso compare tra i primi K risultati restituiti dal sistema. Nel caso specifico è stato utilizzato Hit@5, coerente con uno scenario in cui l'operatore visualizza una lista breve di candidati e si aspetta che almeno un elemento corretto appaia nei primi risultati.
La Precision@K misura, per ciascuna query, la frazione di prodotti rilevanti presenti nei primi K risultati, mentre la Recall@K misura la frazione di prodotti rilevanti recuperati nei primi K rispetto al totale dei prodotti attesi. In questo studio è stata adottata Precision@5 e Recall@5, mediate sulle query di test, in linea con la letteratura IR per sistemi di raccomandazione e ricerca prodotti.
La metrica **Mean Reciprocal Rank (MRR)** valuta la posizione del primo risultato corretto nella lista ordinata: valori prossimi a 1 indicano che il primo prodotto atteso si trova quasi sempre al primo posto, mentre valori più bassi suggeriscono che l'utente deve scorrere più elementi prima di trovare ciò che cerca. Infine, i percentili P50 e P95 dei tempi di risposta (latenza, in millisecondi) forniscono una misura robusta dell'esperienza utente tipica e dei casi più lenti: P50 rappresenta la latenza mediana, P95 quella sotto la quale ricade il 95% delle richieste.

5.5.2 Risultati sperimentali

I test sono stati eseguiti su un'istanza di sviluppo di Odoo 18 con il modulo **AI LiveBot** configurato per utilizzare, in modo intercambiabile, cinque modelli LLM. La Tabella Tabella 8 riassume i risultati ottenuti.

Modello	**Tool Acc.**	**Hit@5**	**P@5**	**R@5**	**MRR**	**P50 (ms)**	**P95 (ms)**
Gemini 2.5 Flash	**85.71%**	**100%**	36.67%	81.41%	**1.00**	**2,518**	**5,910**
DeepSeek R1T2 Chimera	71.43%	**100%**	**40.00%**	**89.74%**	**1.00**	14,590	30,370
TNG R1T Chimera	71.43%	**100%**	**40.00%**	**89.74%**	**1.00**	18,089	28,433
GPT-OSS-20B	71.43%	66.67%	30.00%	56.41%	0.67	4,429	11,662
LLaMA 3.3 70B	71.43%	66.67%	30.00%	56.41%	0.67	6,870	17,778

Tabella 8: Confronto tra modelli LLM su 7 query di magazzino.

Tutti i modelli hanno registrato un tasso di errore pari a 0% (nessuna richiesta è terminata con errori infrastrutturali o di parsing dei tag), confermando la robustezza dell'integrazione applicativa. Dal punto di vista della **Tool Choice Accuracy**, **Gemini 2.5 Flash** ottiene il valore più alto (85,71%), mentre gli altri quattro modelli si attestano al 71,43%. In un contesto in cui la scelta della funzione corretta (ad esempio distinguere tra semplice ricerca prodotti e creazione di un ordine) è critica, questo vantaggio è rilevante.
Per quanto riguarda le metriche IR sulla funzione `search_products`, i modelli di tipo «reasoning» (DeepSeek R1T2 Chimera, TNG R1T Chimera) e **Gemini 2.5 Flash** raggiungono Hit@5 = 100%, ovvero almeno un prodotto atteso compare sempre tra i primi cinque risultati. I modelli **GPT-OSS-20B** e **LLaMA 3.3 70B** scendono a Hit@5 = 66,67%, suggerendo che in un terzo delle query nessun prodotto atteso compare nei primi cinque risultati.

I modelli DeepSeek e TNG mostrano valori leggermente superiori a **Gemini** in termini di Precision@5 (40,00% contro 36,67%) e Recall@5 (89,74% contro 81,41%), indicando che, a parità di orchestrazione, tendono a posizionare una quota leggermente maggiore di prodotti rilevanti entro i primi cinque risultati. Tuttavia, la metrica **MRR** è pari a 1 per tutti e tre i modelli reasoning, a indicare che il primo prodotto corretto è sistematicamente in prima posizione.
Dal punto di vista temporale, **Gemini 2.5 Flash** presenta una latenza P50 di circa 2.5 secondi e una P95 inferiore ai 6 secondi, risultando nettamente più reattivo rispetto agli altri modelli. I modelli DeepSeek e TNG evidenziano invece P50 superiori ai 14 secondi e P95 oltre i 28 secondi, valori poco compatibili con un utilizzo interattivo in un ERP. I modelli **GPT-OSS-20B** e **LLaMA 3.3 70B** hanno prestazioni temporali intermedie, ma non offrono vantaggi sufficienti per compensare la qualità inferiore nelle metriche IR.

5.5.3 Discussione e limiti dello studio

Nel complesso, i risultati mettono in luce un chiaro compromesso qualità–velocità tra i modelli: i modelli reasoning offrono la migliore qualità di ranking ma con tempi di risposta molto elevati; **Gemini 2.5 Flash** rappresenta invece un punto di equilibrio favorevole, combinando una buona qualità IR, la percentuale più alta di tool choice corretta e una latenza compatibile con un utilizzo quotidiano del sistema.
Alla luce di questi dati, **Gemini 2.5 Flash** è stato scelto come modello raccomandato per l'integrazione di **AI LiveBot** in Odoo 18. Tale scelta è coerente con le esigenze di un assistente conversazionale di magazzino, in cui la reattività è un requisito chiave e la qualità delle risposte deve essere «sufficientemente buona» piuttosto che assoluta.
È importante tuttavia sottolineare alcune limitazioni dello studio: il dataset di test comprende solo 7 query (di cui 6 con `expected_product_ids`) e 43 prodotti, configurando una valutazione su scala ridotta, utile per un confronto preliminare ma non sufficiente per trarre conclusioni statisticamente robuste. Inoltre, i tempi di rete non sono stati rigidamente controllati, per cui le misure di latenza riflettono condizioni d'uso realistiche ma non completamente riproducibili. Infine, non sono state ancora introdotte metriche end-to-end (ad esempio il tasso di successo nella creazione completa di un ordine) né valutazioni qualitative tramite giudizi umani.
Come sviluppi futuri, si prevede di estendere il dataset di test con un numero maggiore di query e scenari (inclusi casi di modifica ordini, resi e stock parziali), di introdurre metriche end-to-end sui workflow completi e di ripetere il confronto su ambienti di esecuzione più controllati per affinare la stima delle latenze.

Capitolo 6

Conclusioni

In questo capitolo conclusivo vengono analizzati i risultati ottenuti al termine del periodo di stage, confrontandoli con gli obiettivi prefissati. Vengono inoltre discusse le conoscenze acquisite e i possibili sviluppi futuri del progetto.

6.1 Raggiungimento degli obiettivi

Il progetto **AI LiveBot** ha raggiunto con successo gli obiettivi principali definiti nel piano di lavoro. L'integrazione tra Odoo 18 e Google Gemini è stata completata, permettendo agli utenti di interagire con il sistema gestionale tramite linguaggio naturale.
In particolare:

- **Obiettivi Obbligatori**: Sono stati pienamente soddisfatti. Il modulo è in grado di interpretare richieste, interrogare il magazzino e creare ordini di vendita.
- **Obiettivi Desiderabili**: È stata acquisita una buona autonomia nello sviluppo di moduli Odoo custom.
- **Obiettivi Facoltativi**: L'integrazione Speech-to-Text è stata implementata con successo, permettendo l'inserimento vocale dei comandi e ricevendo feedback positivi durante le review interne.

Dal punto di vista della sostenibilità economica, l'analisi del consumo di token svolta nei capitoli tecnici mostra che l'integrazione di un LLM come orchestratore di workflow ERP è compatibile con un utilizzo produttivo, a patto di progettare con attenzione i prompt e il contesto. I test hanno evidenziato che una singola richiesta può richiedere in media nell'ordine di alcune decine di migliaia di token (in gran parte dovuti al system prompt e allo storico), ma che tali costi rimangono gestibili se riservati alle operazioni a maggior valore aggiunto (creazione ordini, verifiche di magazzino, riepiloghi complessi) e se accompagnati da strategie di ottimizzazione del contesto. In questo senso, la soluzione sviluppata dimostra non solo la fattibilità tecnica dell'integrazione, ma anche la possibilità di raggiungere un equilibrio ragionevole tra benefici operativi e costi di utilizzo del modello.

6.2 Requisiti soddisfatti

La Tabella Tabella 9 riassume lo stato di soddisfacimento dei requisiti identificati in fase di analisi.

Tipo	Mandatory	Desirable	Optional	Somma
Functional	6/6	0/0	1/1	7/7
Qualitative	2/2	1/1	0/0	3/3
Constraint	3/3	0/0	0/0	3/3
Totale	**11/11**	**1/1**	**1/1**	**13/13**

Tabella 9: Riepilogo dei requisiti soddisfatti.

6.3 Limitazioni del sistema

Nonostante i risultati positivi, il sistema presenta alcune limitazioni intrinseche che è importante riconoscere:

- **Dipendenza dal modello LLM**: Le prestazioni del sistema sono strettamente legate alla qualità del modello utilizzato. L'uscita di versioni più recenti di Gemini o di altri LLM potrebbe migliorare o peggiorare i risultati se il system prompt e il codice non venissero riadattati e ottimizzati di conseguenza.
- **Possibili allucinazioni**: Come tutti i modelli generativi, l'LLM può occasionalmente produrre risposte plausibili ma errate, specialmente per query ambigue o al di fuori del dominio previsto.
- **Latenza delle risposte**: L'interazione con API esterne introduce inevitabilmente una latenza maggiore rispetto a un'interfaccia tradizionale, che può impattare l'esperienza utente per operazioni frequenti.
- **Consumo di token**: Operazioni semplici possono comunque richiedere un numero significativo di token a causa del system prompt e dello storico conversazionale, rendendo il sistema meno conveniente per task ripetitivi e a basso valore aggiunto.
- **Dipendenza dalla connettività**: Il funzionamento del modulo richiede una connessione internet stabile per le chiamate API verso Google Gemini.

- **Limiti di contesto**: Conversazioni molto lunghe possono superare la finestra di contesto del modello, richiedendo strategie di troncamento che potrebbero far perdere informazioni rilevanti.

Queste limitazioni non compromettono la validità della soluzione sviluppata, ma evidenziano aree di attenzione per un eventuale deployment in produzione e per futuri interventi di ottimizzazione.

6.4 Conoscenze acquisite

Questa esperienza di stage ha rappresentato un'importante opportunità di crescita professionale. Le principali competenze acquisite riguardano:

- **Sviluppo ERP**: Comprensione profonda dell'architettura di Odoo, del suo ORM e del meccanismo di ereditarietà dei moduli.
- **Generative AI**: Esperienza pratica nell'utilizzo di API LLM, tecniche di Prompt Engineering e gestione di output strutturati (JSON).
- **Metodologie di lavoro**: Affinamento delle capacità di lavorare in team, gestione del versionamento con Git e rispetto delle scadenze.

6.5 Sviluppi futuri

Il progetto pone le basi per ulteriori evoluzioni, tra cui:

- **Supporto Multimodale**: Integrazione completa del riconoscimento vocale e analisi di immagini (es. foto di prodotti).
- **RAG (Retrieval-Augmented Generation)**: Implementazione di un sistema RAG per permettere al bot di rispondere a domande su documenti aziendali (manuali, policy) non strutturati.
- **Analisi Predittiva**: Utilizzo dell'AI non solo per eseguire comandi, ma per suggerire ordini basati sullo storico vendite.

6.6 Considerazioni personali

Lo stage presso Sync Lab è stato estremamente formativo. La possibilità di lavorare su tecnologie all'avanguardia come gli LLM, applicandole a un contesto concreto e complesso come un ERP aziendale, è stata una sfida stimolante. L'ambiente di lavoro collaborativo e il supporto costante del tutor mi hanno permesso di superare le difficoltà tecniche iniziali e di raggiungere un risultato di cui sono pienamente soddisfatto.

Appendice A

Esempi di codice

Tutti gli esempi di questo capitolo provengono direttamente dal modulo `ERP-ChatBot-LLM`*, così da mostrare frammenti reali dell'integrazione tra Odoo 18 e l'assistente AI.*

A.1 Override del canale Discuss

Il cuore dell'assistente è l'override di `discuss.channel` che intercetta ogni messaggio dell'utente, salva eventuali azioni in attesa di conferma e richiama l'LLM solo quando serve. Il codice Codice 3 riporta un estratto dell'override di `message_post`.

```python
class DiscussChannel(models.Model):
    _inherit = 'discuss.channel'

    def message_post(self, **kwargs):
        result = super(DiscussChannel, self).message_post(**kwargs)
        if not self.name or 'AI Assistant' not in self.name:
            return result

        body = kwargs.get('body', '')
        message_type = kwargs.get('message_type', 'comment')
        author_id = kwargs.get('author_id')

        if message_type == 'comment' and author_id:
            author = self.env['res.partner'].browse(author_id)
            if author.name in ['OdooBot', 'System', 'AI Assistant']:
                return result

            user_message = re.sub(r'<[^>]+>', '', body or '').strip()
            confirmation_handled = False

            if re.search(r'\b(SI|CONFERMO|OK VAI|PERFETTO)\b', user_message, re.I):
                confirmation_handled = self._execute_pending_action(user_mes

            if not confirmation_handled:
                self._generate_ai_response(body)

        return result
```

Codice 3: Override `message_post` in `models/ai_chatbot.py`

A.2 Marker e pending action

La riconciliazione tra prompt e backend avviene tramite marker testuali (`[PENDING_SO]`, `[PENDING_CANCEL]`). Il codice seguente salva nel database la funzione da eseguire dopo la conferma e ripulisce il testo che verrà mostrato all'utente.

```python
def _handle_pending_markers(self, text):
    if not text:
        return text

    marker_map = {
        '[PENDING_SO]': 'create_sales_order',
        '[PENDING_CANCEL]': 'cancel_sales_order'
    }

    for marker, function_name in marker_map.items():
        if marker in text:
            parts = text.split(marker)
            clean_text = parts[0].strip()
            json_part = parts[1] if len(parts) > 1 else ''
            json_str = _balanced_json_extract_simple(json_part)

            if json_str:
                params = json.loads(json_str)
                self.pending_action_data = json.dumps({
                    'function': function_name,
                    'params': params
                })

            if not any(x in clean_text.lower() for x in ['confermi', 'procedo', 'ok?']):
                clean_text += "\n\nConfermi? (rispondi SI/CONFERMO/OK VAI)"
            return clean_text

    return text
```

Codice 4: Gestione marker in `models/ai_chatbot.py`

A.3 Servizio `warehouse.operations`

Tutte le operazioni di dominio (ricerche, creazione ordini, evasione delivery) passano da un modello astratto dedicato. L'esempio seguente mostra la ricerca fuzzy multi-pattern dei prodotti.

```python
@api.model
def search_products(self, search_term=None, limit=50, product_type=None):
    Product = self.env['product.product']
    base_domain = []
    # Filtro per tipo prodotto (service, product, storable...)
    if product_type:
        type_map = {'service': 'service', 'product': 'product', ...}
        dt = type_map.get((product_type or '').strip().lower())
        if dt:
            base_domain.append(('product_tmpl_id.type', '=', dt))

    # 1) Match esatto
    domain_exact = base_domain + [('name', '=ilike', search_term)]
    products = Product.search(domain_exact, limit=limit)
    if products:
        return self._format_product_results(products)

    # 2) Match tutte le parole (AND)
    words = [w for w in search_term.split() if len(w) > 2]
    if len(words) > 1:
        for word in words:
            base_domain.append(('name', 'ilike', word))
        products = Product.search(base_domain, limit=limit)
        if products:
            return
return self._format_product_results(products)
domain_fallback = base_domain + [('name',
```

Codice 5: Ricerca prodotti in models/warehouse_operations.py

A.4 Controller JSON per Odoo

Le API che il frontend Odoo utilizza per parlare con l'assistente passano da un controller leggero che delega ogni chiamata al modello astratto oppure all'override del canale. Il codice Codice 6 mostra un esempio di controller minimale con route JSON per chat e magazzino.

```python
class AILiveBotController(http.Controller):

    @http.route('/ai_livebot/chat', type='json', auth='user', methods=['POST'])
    def chat(self, message, channel_id):
        channel = request.env['discuss.channel'].browse(channel
        if not channel.exists():
            return {'error': 'Channel not found'}
        channel.message_post(body=message, message_type='comment',
                             subtype_xmlid='mail.mt_c
        return {'success': True}

    @http.route('/ai_livebot/warehouse/stock', type='json', auth='user', methods=['GET'])
    def get_stock(self, product_name=None, product_id=None):
        warehouse_ops = request.env['warehouse.operations']
        return warehouse_ops.get_stock_info(product_name=prc
        product_id=product_id)

    @http.route('/ai_livebot/warehouse/validate', type='json', auth='user', methods=['POST'])
    def validate_order(self, picking_id):
        warehouse_ops = request.env['warehouse.operations']
        return warehouse_ops.validate_delivery(picking_id)
```

Codice 6: Controller `controllers/main.py`

A.5 Interfaccia Speech-to-Text

La parte client introduce la dettatura vocale direttamente nel composer di Discuss sfruttando il Web Speech API e un semplice toggle alimentato da Owl. Il codice Codice 7 mostra un estratto della patch JavaScript che abilita la dettatura vocale nel composer.

```javascript
/** @odoo-module **/
import { useState } from "@odoo/owl";

patch(Composer.prototype, {
    setup() {
        super.setup(...arguments);
        this.dictationState = useState({ isListening: false, error: null });
        if ("webkitSpeechRecognition" in window) {
            this.recognition = new webkitSpeechRecognition();
            this.recognition.continuous = true;
            this.recognition.lang = "it-IT";
            this.recognition.onresult = (event) => {
                // Estrae la trascrizione finale e la inserisce nella textarea
                let transcript = event.results[event.resultIndex][0].transcript;
                const textarea = document.querySelector(".o-mail-Composer-input");
                if (textarea) textarea.value += transcript;
            };
        }
    },
    toggleDictation() {
        if (!this.recognition) return alert("Browser non supportato.");
        this.dictationState.isListening ? this.recognition.stop()
        : this.recognition.start();
        this.dictationState.isListening = !this.dictationState.isListening;},});
```

Codice 7: Patch del composer in `static/src/js/composer_dictation.js`

A.6 XML del bottone microfono

Il template QWeb eredita il composer di Discuss e aggiunge un'azione coerente con lo stato della dettatura. Il codice Codice 8 mostra un estratto del template XML che aggiunge il bottone microfono.

```xml
<templates xml:space="preserve">
    <t t-inherit="mail.Composer" t-inherit-mode="extension">
        <xpath expr="//div[contains(@class, 'o-mail-Composer-footer')]" position="inside">
            <button class="btn btn-link o-mail-Composer-action"
                    t-att-class="{ 'text-danger': dictationState.isListening }"
                    t-on-click="toggleDictation"
                    title="Dettatura Vocale">
                <i class="fa fa-microphone" t-if="!dictationState.isListening"/>
                <i class="fa fa-stop-circle fa-spin" t-if="dictationState.isListening"/>
            </button>
        </xpath>
    </t>
</templates>
```

Codice 8: Template `static/src/xml/composer_dictation.xml`

Questi esempi sono sincronizzati con il repository `ERP-ChatBot-LLM` e fungono da riferimento diretto al codice sorgente del progetto.

Glossario

***API* – Application Programming Interface**: Insieme di protocolli e strumenti che permettono a diverse applicazioni software di comunicare tra loro. 18., 20., 26.

***CoT* – Chain-of-Thought**: Tecnica di prompting che guida un modello linguistico a esplicitare il ragionamento passo per passo prima di fornire la risposta finale, migliorando le prestazioni su compiti di ragionamento complesso. 20.

***CRM* – Customer Relationship Management**: Insieme di processi e strumenti per gestire le relazioni con i clienti (contatti, opportunità, pipeline di vendita, assistenza). 15.

***CSS3* – Cascading Style Sheets (CSS3)**: Linguaggio di stile per definire l'aspetto grafico di pagine web; CSS3 include moduli e funzionalità avanzate. 5., 15.

***DBMS* – Database Management System**: Software che consente di creare, gestire e interrogare database, garantendo persistenza, concorrenza e integrità dei dati. 18.

***DSL* – Domain Specific Language**: Linguaggio di programmazione o specifica progettato per un dominio applicativo particolare, in contrapposizione ai linguaggi general-purpose. 18., 20., 27.

***ERP* – Enterprise Resource Planning**: Sistema software integrato per la gestione dei processi aziendali: contabilità, vendite, magazzino, produzione, risorse umane, ecc. 1., 15.

***GIL* – Global Interpreter Lock**: Meccanismo di CPython che limita l'esecuzione simultanea di bytecode Python in più thread, influenzando il parallelismo CPU-bound. 22.

***GPU* – Graphics Processing Unit**: Processore specializzato per il calcolo parallelo, originariamente per la grafica, oggi ampiamente usato per l'addestramento e l'inferenza di modelli di machine learning. 18.

***Hit@K* – Hit@K**: Metrica che indica la percentuale di query per cui almeno un risultato rilevante compare tra i primi K risultati. 30.

***HTML5* – HyperText Markup Language (HTML5)**: Standard di markup per strutturare contenuti web; HTML5 introduce API e funzionalità moderne per applicazioni web. 5., 15.

***HTTP* – Hypertext Transfer Protocol**: Protocollo di comunicazione alla base del web; usato per lo scambio di richieste e risposte tra client e server. 27.

***HTTPS* – HTTP Secure**: Versione di HTTP che utilizza TLS per cifrare e autenticare la comunicazione tra client e server. 24.

***ICT* – Information and Communication Technology**: Insieme delle tecnologie e dei servizi legati all'informatica e alle telecomunicazioni (reti, software, sistemi informativi, ecc.). 1.

***IDE* – Integrated Development Environment**: Ambiente di sviluppo che integra editor, strumenti di build/debug e supporto al linguaggio (es. Visual Studio Code). 23.

***IR* – Information Retrieval**: Disciplina che si occupa del recupero di informazioni rilevanti da grandi collezioni di dati, tipicamente documenti testuali. 30.

***IT* – Information Technology**: Ambito che comprende infrastrutture, software e servizi informatici utilizzati per supportare processi e attività aziendali. 1.

***JS* – JavaScript**: Linguaggio di programmazione eseguito principalmente nel browser, usato per aggiungere interattività alle pagine e alle web app. 5., 15.

***JSON* – JavaScript Object Notation**: Formato di scambio dati leggero, basato su testo e leggibile dall'uomo, comunemente usato per la comunicazione tra client e server. 18.

***LLM* – Large Language Model**: Modello di deep learning addestrato su grandi quantità di testo, capace di comprendere e generare linguaggio naturale. Esempi: GPT-4, Gemini, Claude, LLaMA. 1., 3., 18., 20., 26., 27., 30.

***MRR* – Mean Reciprocal Rank**: Metrica di valutazione per sistemi di ranking che misura la posizione media del primo risultato rilevante. 31.

***ORM* – Object-Relational Mapping**: Tecnica di programmazione che permette di interagire con un database relazionale utilizzando oggetti del linguaggio di programmazione, astraendo le query SQL. 17.

***OWL* – Odoo Web Library**: Framework JavaScript reattivo sviluppato da Odoo per la costruzione di componenti dell'interfaccia utente web. 15.

***P50* – 50° percentile**: Percentile 50 (mediana): valore sotto il quale ricade il 50% delle osservazioni (es. latenza). 31.

***P95* – 95° percentile**: Percentile 95: valore sotto il quale ricade il 95% delle osservazioni (evidenzia i casi più lenti). 31.

***Precision@K* – Precision@K**: Metrica che misura la frazione di risultati rilevanti tra i primi K elementi restituiti. 30.

***RAG* – Retrieval-Augmented Generation**: Architettura che integra un modello generativo con un sistema di recupero di informazioni esterne, riducendo la dipendenza dalla sola memoria parametrica del modello e mitigando le allucinazioni. 20.

***Recall@K* – Recall@K**: Metrica che misura la frazione di risultati rilevanti recuperati nei primi K rispetto al totale dei rilevanti attesi. 30.

***REST* – Representational State Transfer**: Stile architetturale per sistemi distribuiti che utilizza il protocollo HTTP per la comunicazione tra client e server. 26.

***SQL* – Structured Query Language**: Linguaggio standard per interrogare e manipolare dati in database relazionali. 5., 17., 19.

***STT* – Speech-to-Text**: Tecnologia che converte il parlato in testo scritto, nota anche come riconoscimento vocale automatico (ASR). 3., 23.

***TCA* – Tool Choice Accuracy**: Metrica che misura la percentuale di volte in cui un modello seleziona correttamente lo strumento/funzione da invocare. 30.

***Token* – Token**: Unità base di testo elaborata dai modelli linguistici. Può corrispondere a una parola, parte di parola o carattere, a seconda del tokenizer utilizzato. 18., 20., 21., 26.

***XML* – eXtensible Markup Language**: Linguaggio di markup per rappresentare dati in forma strutturata; in Odoo è usato, ad esempio, per definire viste e dati. 4., 23.

***XPath* – XML Path Language**: Linguaggio per selezionare nodi all'interno di un documento XML, usato in Odoo per localizzare elementi nelle viste da estendere. 4., 23.

Bibliografia

[1] Google, «Google Gemini API Documentation». [Online]. Disponibile su: https://ai.google.dev/docs

[2] Odoo S.A., «Odoo 18 Developer Documentation». [Online]. Disponibile su: https://www.odoo.com/documentation/18.0/developer.html

[3] Odoo S.A., «Odoo OWL Framework Documentation». [Online]. Disponibile su: https://github.com/odoo/owl

[4] The PostgreSQL Global Development Group, «PostgreSQL Documentation». [Online]. Disponibile su: https://www.postgresql.org/docs/

[5] OpenAI, «GPT-4 Technical Report». [Online]. Disponibile su: https://openai.com/research/gpt-4

[6] Anthropic, «Claude 3 Model Card». [Online]. Disponibile su: https://www.anthropic.com/claude

[7] Q. Dong *et al.*, «A Survey on In-context Learning», 2023. [Online]. Disponibile su: https://arxiv.org/abs/2301.00234

[8] Z. Ji *et al.*, «Survey of Hallucination in Natural Language Generation», *ACM Computing Surveys*, vol. 55, 2023, doi: 10.1145/3571730.

[9] D. Anh-Hoang, V. Tran, e L.-M. Nguyen, «Survey and Analysis of Hallucinations in Large Language Models: Attribution to Prompting Strategies or Model Behavior», *Frontiers in Artificial Intelligence*, vol. 8, 2025, doi: 10.3389/frai.2025.1622292.

[10] W3C, «Web Speech API Specification». [Online]. Disponibile su: https://wicg.github.io/speech-api/

[11] Z. Ji *et al.*, «Survey of Hallucination in Natural Language Generation», 2023. [Online]. Disponibile su: https://arxiv.org/abs/2202.03629