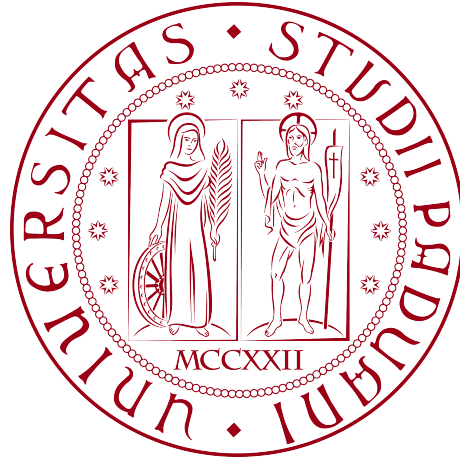


Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



QueryWhiz assistente virtuale per  
interrogazioni SQL

*Tesi di Laurea Triennale*

*Relatore*

Prof. Marco Zanella

*Laureando*

Vullnet Vogli

Matricola 2042379

---

ANNO ACCADEMICO 2023-2024



# Ringraziamenti

Desidero esprimere la mia gratitudine al professor Marco Zanella, mio relatore, per l'aiuto e il sostegno che mi ha dato durante la stesura dell'elaborato.

Vorrei anche ringraziare, con affetto, i miei genitori per il loro sostegno, il grande aiuto e la loro presenza in ogni momento durante gli anni di studio.

Desidero poi ringraziare i miei amici per i bellissimi anni trascorsi insieme e le mille avventure vissute.

Padova, Dicembre 2024

*Vullnet Vogli*

# Sommario

Il presente documento descrive lo sviluppo di un chatbot per potenziare le funzionalità dell'interfaccia aziendale esistente, denominata "Zoom", utilizzata per la gestione dei dati. Attualmente, Zoom consente operazioni di visualizzazione tabellare e semplici filtri.

L'obiettivo del progetto è migliorare l'esperienza utente attraverso un chatbot capace di interpretare richieste in linguaggio naturale, offrendo interrogazioni più avanzate e un accesso immediato a informazioni dettagliate e personalizzate.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	L'azienda . . . . .	1
1.2	Organizzazione del testo . . . . .	2
<b>2</b>	<b>Descrizione dello stage</b>	<b>3</b>
2.1	Introduzione al progetto . . . . .	3
2.2	Pubblico di riferimento . . . . .	3
2.3	Uso previsto . . . . .	4
2.4	Ambito del prodotto . . . . .	4
<b>3</b>	<b>Analisi dei requisiti</b>	<b>5</b>
3.1	Requisiti funzionali obbligatori . . . . .	6
3.2	Requisiti funzionali desiderabili . . . . .	6
3.3	Requisiti non funzionali obbligatori . . . . .	7
3.4	Requisiti non funzionali desiderabili . . . . .	8
<b>4</b>	<b>Architettura di Sistema</b>	<b>9</b>
4.1	Stack tecnologico . . . . .	10
4.1.1	Linguaggio . . . . .	10
4.1.2	Protocollo . . . . .	12
4.1.3	Framework API . . . . .	15
4.1.4	Reverse proxy . . . . .	16
4.1.5	Container . . . . .	17
4.1.6	Database . . . . .	17
4.2	Descrizione API . . . . .	19

4.2.1	Validazione e gestione degli errori . . . . .	19
4.2.2	Documentazione con Swagger . . . . .	19
4.3	Architettura della REST API . . . . .	20
4.3.1	Implementazioni di <code>BaseModel</code> . . . . .	20
4.3.2	Test e validazione tramite <code>TestModel</code> . . . . .	20
4.4	Backend . . . . .	22
4.4.1	Business logic . . . . .	23
4.4.2	Flusso generale . . . . .	24
<b>5</b>	<b>Verifica e validazione</b>	<b>26</b>
5.1	Test di Unità . . . . .	26
5.2	Test di Integrazione . . . . .	28
5.3	Test di Validazione . . . . .	28
5.4	Test Parametrizzati . . . . .	29
5.5	Test sui Logger . . . . .	30
<b>6</b>	<b>Test LLM</b>	<b>31</b>
6.1	Studio valutazione query generate . . . . .	32
6.2	Implementazione benchmark LLM . . . . .	33
6.3	Conclusioni . . . . .	33
<b>7</b>	<b>Requisiti soddisfatti</b>	<b>35</b>
7.1	Requisiti funzionali obbligatori . . . . .	35
7.2	Requisiti funzionali desiderabili . . . . .	37
7.3	Requisiti non funzionali obbligatori . . . . .	38
7.4	Requisiti non funzionali desiderabili . . . . .	39
<b>8</b>	<b>Miglioramenti software</b>	<b>40</b>
<b>9</b>	<b>Conclusioni</b>	<b>41</b>
9.1	Conoscenze acquisite . . . . .	41
9.2	Valutazione personale . . . . .	i
	<b>Acronimi e abbreviazioni</b>	<b>ii</b>

## INDICE

---

Glossario	iii
Bibliografia	v
Sitografia	vi

# Elenco delle figure

4.1	Overview architettura . . . . .	10
4.2	UML API . . . . .	20
4.3	UML Backend . . . . .	23
4.4	Diagramma di sequenza . . . . .	24

# Elenco delle tabelle

3.1	Requisiti funzionali obbligatori . . . . .	6
3.2	Requisiti funzionali desiderabili . . . . .	6
3.3	Requisiti non funzionali obbligatori . . . . .	7
3.4	Requisiti non funzionali desiderabili . . . . .	8
4.1	REST vs. gRPC vs. GraphQL . . . . .	15
5.1	Tabella dei test di unità. . . . .	27
5.2	Tabella dei test di integrazione. . . . .	28
5.3	Tabella dei test di validazione. . . . .	28
5.4	Tabella dei test parametrizzati. . . . .	29
5.5	Tabella dei test sui logger. . . . .	30

## ELENCO DELLE TABELLE

---

7.1	Stato requisiti funzionali obbligatori . . . . .	36
7.2	Stato requisiti funzionali desiderabili . . . . .	37
7.3	Stato requisiti non funzionali obbligatori . . . . .	38
7.4	Stato requisiti non funzionali desiderabili . . . . .	39

# Elenco dei codici sorgenti

6.1	Esempio di equivalenza semantica . . . . .	32
6.2	Esempio di equivalenza sintattica . . . . .	32



# Capitolo 1

## Introduzione

### 1.1 L'azienda

L'esperienza di tirocinio ha avuto luogo negli uffici della sede padovana dell'azienda <sup>1</sup>Zucchetti S.p.A: la prima software house italiana. Fondata nel 1978, Zucchetti S.p.A. risponde alle esigenze dei clienti fornendo numerosi servizi ad aziende, professionisti e associazioni di categoria per la gestione del personale, la contabilità, il fisco e la gestione dei processi aziendali (vendite, logistica, magazzino, produzione ecc.). In questo senso, si configura come un significativo punto di riferimento in grado di fornire un valido supporto per la gestione delle più svariate esigenze di business. Tra le varie sedi sparse in tutto il mondo, quella di Padova si occupa di ricerca e sviluppo.

---

<sup>1</sup>[www.zucchetti.it](http://www.zucchetti.it)

## 1.2 Organizzazione del testo

**Il secondo capitolo** esplicita l'obiettivo del software e i vari vincoli

**Il terzo capitolo** approfondisce i requisiti che il software deve soddisfare

**Il quarto capitolo** riguarda le varie scelte tecnologiche e architetturali del software

**Il quinto capitolo** riguarda i test effettuati sul software

**Il sesto capitolo** approfondisce le tecniche di benchmark dei modelli

**Il settimo capitolo** esplicita i requisiti soddisfatti

**L'ottavo capitolo** propone diverse migliorie e nuove funzionalità

**Il nono capitolo** presenta una sintesi finale

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

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: *Application Program Interface (API)<sub>G</sub>*;

# Capitolo 2

## Descrizione dello stage

### 2.1 Introduzione al progetto

Il progetto mira ad ampliare le capacità dell'interfaccia aziendale esistente, denominata "Zoom", per la gestione dei dati. Attualmente, l'interfaccia consente la visualizzazione dei dati in formato tabellare e offre funzionalità limitate a semplici filtri di ordinamento e selezione.

L'obiettivo è sviluppare un chatbot che potenzi queste funzionalità, consentendo agli utenti di effettuare interrogazioni più complesse.

Il chatbot sarà in grado di interpretare richieste formulate in linguaggio naturale, migliorando così l'interazione con i dati e facilitando l'accesso a informazioni più dettagliate e personalizzate.

### 2.2 Pubblico di riferimento

Questa sezione descrive le persone o i gruppi che utilizzeranno il software, specificando le loro competenze e il livello di interazione con il sistema. L'obiettivo è delineare chi beneficerà dell'applicazione e a chi è destinata.

Il chatbot è progettato per soddisfare le esigenze di diverse categorie di utenti, inclusi utenti occasionali, amministratori e sviluppatori. L'obiettivo è rendere il prodotto accessibile e utilizzabile anche da chi ha una conoscenza tecnica limitata, permettendo a tutti gli utenti di ottenere i risultati desiderati

senza la necessità di una preparazione tecnica approfondita. La progettazione dell'interfaccia e delle funzionalità è tale da non richiedere competenze tecniche avanzate.

L'applicazione è stata pensata come prodotto interno aziendale.

### 2.3 Uso previsto

Il software è progettato per

- Fornire strumenti e funzionalità per l'interpretazione e la manipolazione dei dati.
- Offrire un'interfaccia per semplificare le query e le operazioni sui dati.

La frequenza d'uso è a discrezione dell'utente.

### 2.4 Ambito del prodotto

L'ambito del prodotto stabilisce il perimetro delle funzionalità che il sistema offrirà e le aree che rimarranno fuori dal suo scopo.

- Funzionalità incluse: il sistema permetterà la generazione di query SQL.
- Funzionalità escluse: non sono previste modifiche alle strutture o ai dati del database.
- Integrazioni con altri sistemi: per la generazione delle query SQL, il chatbot si interfacerà con una *RESTful API* dedicata.

# Capitolo 3

## Analisi dei requisiti

In questa sezione vengono definiti i requisiti individuati in seguito ad un accurato studio ed interazione tra le parti interessate.

Ogni requisito è classificabile in:

- **Funzionale:** i requisiti che l'utente finale richiede esplicitamente e che il sistema deve offrire. Descrivono le funzionalità del sistema, le azioni che il sistema può compiere e le informazioni che il sistema può fornire.
- **Non funzionale:** più impliciti, definiscono le qualità e i vincoli che il sistema deve soddisfare come prestazioni, sicurezza e usabilità. Descrivono quindi come deve comportarsi o quali standard deve soddisfare.

Ogni requisito può essere obbligatorio oppure desiderabile.

Inoltre, ogni requisito è identificato da un codice univoco, che gli consente di essere facilmente riconosciuto e referenziato.

### 3.1 Requisiti funzionali obbligatori

La tabella 3.1 specifica i requisiti funzionali obbligatori:

ID	Requisito
FO-001	Essere attivato tramite un'azione dell'utente (ad esempio un elemento grafico o combinazione di tasti)
FO-002	Avere un'interfaccia utente chiara e intuitiva, con elementi grafici e pulsanti per facilitare l'interazione
FO-003	Feedback visivo all'utente durante la fase di generazione risposta
FO-004	Essere in grado di comprendere ed interpretare il linguaggio naturale per gestire una varietà di domande e richieste dell'utente
FO-005	La visualizzazione dei risultati deve avvenire tramite gli strumenti messi a disposizione da Zoom
FO-006	Deve essere in grado di effettuare operazioni di selezioni sui dati
FO-007	Deve essere in grado di gestire operazioni che includono raggruppamenti
FO-008	Deve essere in grado di gestire operazioni che includono condizioni

**Tabella 3.1:** Requisiti funzionali obbligatori

### 3.2 Requisiti funzionali desiderabili

I requisiti funzionali desiderabili sono specificati in tabella 3.2:

ID	Requisito
FD-001	Essere accessibile a tutti gli utenti, inclusi quelli con disabilità, seguendo le linee guida di accessibilità web
FD-002	Permettere all'utente di personalizzare la visualizzazione dei risultati

**Tabella 3.2:** Requisiti funzionali desiderabili

### 3.3 Requisiti non funzionali obbligatori

La tabella 3.3 specifica i requisiti funzionali obbligatori:

<b>ID</b>	<b>Requisito</b>
NFO-001	Validare gli input utente ed assicurare rispettino criteri ben definiti come tipo di dato, lunghezza, o formato
NFO-002	Fornire chiari messaggi di errore in caso di richieste non realizzabili

**Tabella 3.3:** Requisiti non funzionali obbligatori

### 3.4 Requisiti non funzionali desiderabili

I requisiti non funzionali desiderabili sono riportati in tabella 3.4:

<b>ID</b>	<b>Requisito</b>
NFD-001	Autenticazione utenza: solo gli utenti con l'accesso a Zoom devono essere in grado di conversare con il chatbot
NFD-002	Essere utilizzabile da mobile

**Tabella 3.4:** Requisiti non funzionali desiderabili

# Capitolo 4

## Architettura di Sistema

QueryWhiz è una RESTful API progettata per generare *Query SQL<sub>G</sub>* partendo da uno *Schema<sub>G</sub>* della tabella.

Per la generazione della query è stato deciso di utilizzare dei modelli di *Machine Learning (ML)<sub>G</sub>*, questo perché sono in grado di:

1. Comprendere il linguaggio naturale
2. Comprendere la richiesta in base al contesto fornitogli

Durante la fase di progettazione si è deciso di adottare una architettura a microservizi per i seguenti motivi:

1. Eseguire un modello di ML è molto esigente in termini di risorse.
2. Le dimensioni di tali modelli sono spesso elevate.

Utilizzando questo approccio possiamo garantire manutenibilità e scalabilità maggiori.

L'applicazione è pensata per essere eseguita in locale su un singolo server ma, grazie all'utilizzo di un API gateway, è possibile incaricarne diversi così da avere un sistema sempre reattivo ed efficiente.

Il panoramica architetturale è mostrata in figura [4.1](#)

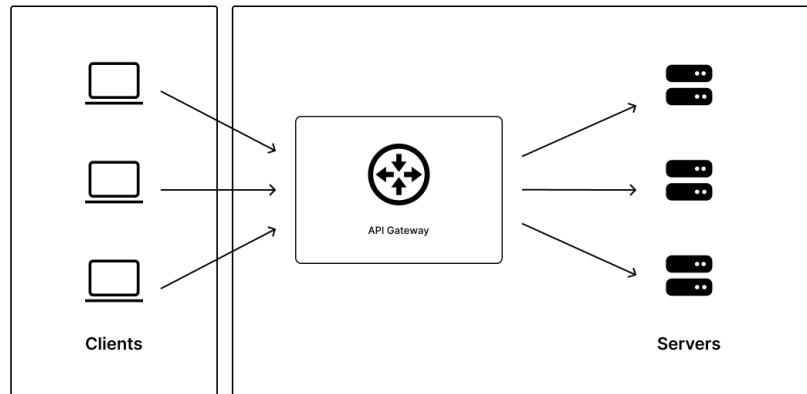


Figura 4.1: Overview architettura

## 4.1 Stack tecnologico

In questa sezione vengono riportati i motivi che hanno portato all'utilizzo di specifici linguaggi, *Librerie<sub>G</sub>* e *Framework<sub>G</sub>*.

Il progetto prevede l'utilizzo di tecnologie open source; si è quindi scelto di utilizzare i modelli disponibili su [Huggingface](#), una piattaforma che offre una ampia gamma di modelli pre-addestrati ed open source per il *Natural Language Processing (NLP)<sub>G</sub>*. Huggingface rappresenta una soluzione ideale in quanto garantisce la flessibilità necessaria per sfruttare modelli di ultima generazione senza limitazioni di licenza proprietaria.

### 4.1.1 Linguaggio

Nel contesto di una architettura a microservizi, la scelta del linguaggio di programmazione per la creazione della REST API si restringeva a tre opzioni principali: Java, C# e Python. Ogni linguaggio presentava vantaggi distinti e la decisione finale è stata presa considerando vari fattori come le esigenze del progetto, le competenze del team e la tecnologia già in uso.

Esaminiamo ciascuno:

- **Java:**

- Framework: esistono diversi framework solidi e ben supportati come Spring Boot e Microaunt.
  - Scalabilità: Java è noto per le sue prestazioni e per la sua capacità di scalare facilmente, rendendolo una scelta adatta per ambienti ad alta intensità di richieste.
  - Compatibilità con ONNX Runtime: l’inferenza dei modelli di machine learning tramite ONNX Runtime può essere integrata, consentendo al linguaggio di gestire processi di inferenza con performance ottimizzate
- **C#:**
    - Integrazione con l’ecosistema .NET: C# è profondamente integrato con la piattaforma .NET, la quale fornisce un’infrastruttura potente e ben documentata per la creazione di microservizi come ASP.NET Core.
    - Cross-platform: con .NET Core, C# ha guadagnato il vantaggio di essere cross-platform, permettendo il deployment di applicazioni su Linux, macOS e Windows, ampliando la sua flessibilità operativa.
    - Compatibilità con ONNX Runtime: ONNX Runtime è facilmente integrabile anche con C#, permettendo di eseguire inferenze sui modelli di machine learning senza problemi di compatibilità o prestazioni.
  - **Python:**
    - Familiarità con il team: Python è già ben conosciuto dal team, riducendo così i tempi di sviluppo e migliorando la velocità con cui l’API poteva essere implementata e testata.
    - Ampio ecosistema ML: essendo Python il linguaggio predominante nel campo del machine learning e del data science, molti modelli open-source (inclusi quelli presenti su Huggingface) sono già ottimizzati per questo linguaggio, rendendo l’integrazione con il backend particolarmente semplice.

- Tempi ridotti: dato che il progetto non presentava requisiti altamente complessi o necessità di performance estreme, Python rappresentava la scelta più pragmatica, bilanciando facilità d'uso e tempo di sviluppo.

La scelta di Python si è rivelata la più adatta per il contesto specifico del progetto, data la sua semplicità, la familiarità con il team e l'ottima integrazione con i modelli di machine learning open-source.

Tuttavia, per progetti futuri con requisiti di scalabilità, prestazioni o integrazioni più avanzate, soluzioni basate su Java o C# potrebbero essere più indicate, offrendo una maggiore robustezza e flessibilità, specialmente in contesti enterprise o distribuiti su larga scala.

### 4.1.2 Protocollo

Per quanto riguarda il protocollo di comunicazione della API la scelta ricadeva tra: gRPC, REST, GraphQL. Iniziamo analizzando i pro e i contro di ciascuno:

- **gRPC**: è un framework ad alte prestazioni ed open source sviluppato da Google.

Pro:

- Si basa su HTTP/2 per il trasporto il quale supporta *Multiplexing<sub>G</sub>*.
- Streaming bidirezionale: permette una comunicazione *Full duplex<sub>G</sub>* rendendolo ideale per applicazioni che hanno bisogno di una comunicazione a due vie (chat apps, real-time data pipelines).
- Strong Typing: i dati sono di un tipo ben definito.

Contro:

- Complessità: richiede un setup più avanzato, con particolare attenzione ai *Protobufs<sub>G</sub>*. In aggiunta, effettuare il debugging di dati in formato binario risulta più impegnativo rispetto a testo in chiaro.
- Supporto limitato: non è nativamente supportato nei browser.

- **REST**: Representational State Transfer

Pro:

- Semplicità e ubiquità: molto diffuso tra la comunità di sviluppatori.
- Browser friendly: basato su HTTP è quindi supportato da tutti i browser.
- Flessibile e leggibile: utilizza *JavaScript Object Notation (JSON)*<sub>G</sub> o *eXtensible Markup Language (XML)*<sub>G</sub>, formati semplici e leggibili sia per macchine che sviluppatori.
- Caching: supporto al caching grazie agli header HTTP, utile in caso di ottimizzazioni necessarie.
- Statelessness: ogni richiesta contiene tutti i dati necessari, rendendo il sistema più scalabile.

Contro:

- Over-fetching/Under-fetching: le chiamate possono risultare inefficienti in casi in cui al client servono più o meno dati di quanti siano ritornati dagli endpoint definiti. Ciò può essere causa di over-fetching (più dati del necessario) o di under-fetching (meno dati del necessario).
- Performance: l'utilizzo di JSON o XML rende più lento il processo di serializzazione/de-serializzazione rispetto al formato binario di gRPC. Va inoltre tenuto in considerazione che il formato binario ha dimensione più ridotta.

- **GraphQL**: linguaggio di query open source.

Pro:

- Query specifiche: i client richiedono esattamente i dati necessari andando così ad evitare over-fetching e under-fetching.
- Endpoint singolo: al contrario di REST, il quale solitamente ha molteplici endpoint, GraphQL tipicamente utilizza un singolo endpoint per tutte le query semplificando la gestione.

- Recupero efficiente dei dati: i client possono recuperare dati nidificati o correlati in un'unica richiesta, evitando la necessità di più round trip tipici di REST.
- Strong Typing: i dati sono di un tipo ben definito.

Contro:

- Complessità: setup e design di uno schema efficiente sono operazioni di complessità maggiore.
- Overhead per casi d'uso semplici: per semplici operazioni *CRUD<sub>G</sub>*, GraphQL può introdurre complessità inutile.
- Caching complesso: dato che viene fatto utilizzo di un singolo endpoint e query specifiche dipendendi dal client, tradizionali meccanismi di caching HTTP sono di difficile implementazione.
- Performance: in caso di query complesse il server può aumentare il carico e comportare un'elaborazione maggiore rispetto a una tipica richiesta REST o gRPC.

A seguito una tabella riassuntiva (4.1):

Caratteristica	gRPC	REST	GraphQL
<b>Protocollo di Trasporto</b>	HTTP/2	HTTP/1.1	HTTP/1.1 o HTTP/2
<b>Formato dei Dati</b>	Protobuf (binario)	JSON, XML	JSON
<b>Prestazioni</b>	Alte (binario, HTTP/2)	Moderate (testo, HTTP/1.1)	Moderate
<b>Complessità</b>	Alta (Protobuf, streaming, configurazione)	Bassa (semplice, conosciuto)	Media (struttura query, schema)
<b>Supporto per Streaming</b>	Sì (bidirezionale)	Limitato (HTTP/1.1, no streaming)	No
<b>Caching</b>	No (nativo)	Sì (caching HTTP)	Difficile (unico endpoint)
<b>Casi d'Uso Ideali</b>	Bassa latenza, throughput elevato, microservizi	CRUD, API semplici, applicazioni browser	Dati complessi, flessibilità lato frontend
<b>Supporto per Browser</b>	Limitato (gRPC-Web)	Sì	Sì

**Tabella 4.1:** REST vs. gRPC vs. GraphQL

Visti i trade off si è deciso di utilizzare **REST** come protocollo di comunicazione.

### 4.1.3 Framework API

I modelli di machine learning, in particolare i modelli di deep learning, tendono a essere vincolati alla CPU o alla GPU. Questi carichi di lavoro sono spesso intensivi dal punto di vista computazionale, il che li rende inadatti all'elaborazione asincrona. Se l'inferenza del modello è computazionalmente pesante (ad

esempio, un modello con miliardi di parametri), richieste asincrone non forniranno molti vantaggi poiché il ciclo di eventi verrà bloccato durante l'esecuzione del modello.

Possiamo scegliere quindi due framework sincroni: Flask e Django.

Django è un framework completo, dotato di tutti gli strumenti necessari per sviluppare applicazioni complesse, come ORM, pannello di amministrazione, autenticazione e autorizzazione, moduli e molto altro.

Poiché il progetto non richiede molte delle funzionalità avanzate di Django, la scelta è ricaduta su <sup>1</sup>**Flask**, un micro-framework leggero che fornisce il pieno controllo sull'architettura, ideale se si vuole configurazione minima e costruire solo i componenti specifici di cui si ha bisogno.

### 4.1.4 Reverse proxy

Come reverse proxy, si è optato per l'utilizzo di <sup>2</sup>**Nginx**, principalmente per la sua leggerezza in termini di risorse computazionali e la sua semplicità di configurazione. Nginx è particolarmente apprezzato per la sua capacità di gestire un alto volume di traffico, grazie alla sua architettura asincrona e non bloccante, che consente di servire un numero elevato di richieste contemporaneamente senza compromettere le prestazioni. Questo lo rende ideale per applicazioni web ad alte prestazioni e microservizi. Offre inoltre numerosi vantaggi in termini di scalabilità orizzontale, facilitando il bilanciamento del carico tra i server backend, migliorando la distribuzione delle richieste e ottimizzando l'uso delle risorse di sistema. Infine, la sua compatibilità con vari sistemi operativi e la possibilità di integrarsi facilmente con altre tecnologie e servizi, come Docker, Kubernetes e le soluzioni di monitoring, lo rendono una scelta ideale per l'architettura a microservizi, consentendo di gestire in modo efficace il traffico tra i vari componenti distribuiti.

---

<sup>1</sup><https://flask.palletsprojects.com/>

<sup>2</sup><https://nginx.org/>

### 4.1.5 Container

Per garantire una maggiore scalabilità e portabilità dell'intero sistema, si è scelto di adottare la containerizzazione utilizzando <sup>3</sup>**Docker**. Questo approccio consente di creare ambienti isolati, autonomi e replicabili, che possono essere facilmente distribuiti e configurati su diverse piattaforme, sia in ambienti di sviluppo che di produzione. I container Docker offrono numerosi vantaggi, tra cui la possibilità di eseguire il sistema in modo consistente su differenti macchine, riducendo i problemi legati alle differenze ambientali tra i vari stadi di sviluppo. L'adozione di Docker consente anche di isolare i diversi componenti del sistema in contenitori separati, facilitando la gestione delle dipendenze e riducendo i conflitti tra le librerie. Ogni container può essere configurato con esattamente le dipendenze necessarie per il corretto funzionamento dell'applicazione, senza che le modifiche a un componente influenzino negativamente gli altri. Questo approccio modularizza l'architettura, rendendo l'intero sistema più flessibile e facile da mantenere.

### 4.1.6 Database

<sup>4</sup>**AlaSQL** è una libreria JavaScript che permette l'esecuzione di query SQL direttamente nel browser, operando esclusivamente in memoria. Essa consente di interrogare strutture dati come array, file JSON, CSV o Excel senza la necessità di un database tradizionale.

A causa delle limitazioni del sistema, che non permettono un accesso diretto a un database, è stata scelta questa libreria per gestire le operazioni sui dati. In particolare, questa libreria viene utilizzata per visualizzare i risultati delle query, operando sui dati precedentemente raccolti durante la fase iniziale di caricamento della tabella.

L'approccio adottato consente di eseguire query personalizzate sui dati disponibili senza la necessità di un database fisico. Ciò offre vantaggi significativi,

---

<sup>3</sup><https://docs.docker.com/desktop/>

<sup>4</sup><https://github.com/AlaSQL/alasql>

come un ulteriore livello di astrazione e una maggiore sicurezza, poiché i dati vengono gestiti interamente in memoria, riducendo il rischio di esposizione a minacce esterne.

## 4.2 Descrizione API

La generazione della query SQL corrispondente alla domanda dell'utente avviene tramite richiamo all'endpoint `/api/v1/generate`.

Per utilizzare questo endpoint, è necessario inviare una richiesta POST con un oggetto JSON contenente due campi principali:

- **question**: la domanda in linguaggio naturale che descrive la query desiderata.
- **context**: lo schema della tabella o del database su cui la query sarà basata.

### 4.2.1 Validazione e gestione degli errori

Per garantire la correttezza dei dati inviati, viene utilizzata la libreria <sup>5</sup>`Pydantic`, che valida i tipi e i valori dei campi. Se i dati non rispettano i requisiti attesi, il sistema rifiuta la richiesta prima di tentare di generare la query SQL.

Nel caso in cui la query non possa essere generata in base ai dati forniti, viene restituito un messaggio di errore che informa l'utente con il testo: *"Non è possibile esaudire la richiesta"*. Questo messaggio segnala problemi come incoerenze nei dati o limiti nel modello di interpretazione della domanda.

### 4.2.2 Documentazione con Swagger

Per documentare e testare le API, è stato scelto <sup>6</sup>`Swagger`. Oltre a fornire una chiara descrizione delle varie funzionalità dell'API, `Swagger` permette di effettuare richieste direttamente dalla sua interfaccia grafica (tramite browser all'indirizzo `"host/apidocs"`), semplificando il processo di testing e validazione da parte degli sviluppatori.

---

<sup>5</sup><https://docs.pydantic.dev/>

<sup>6</sup><https://swagger.io/>

### 4.3 Architettura della REST API

La API utilizza un'interfaccia denominata `BaseModel`, il cui scopo principale è la gestione della generazione delle query SQL in base alle richieste degli utenti. L'interfaccia definisce un metodo chiave, `generate`, che prende in input la domanda dell'utente e lo schema del database, producendo la query SQL corrispondente.

Il diagramma UML rappresentante tale gerarchia è mostrato in figura 4.2.

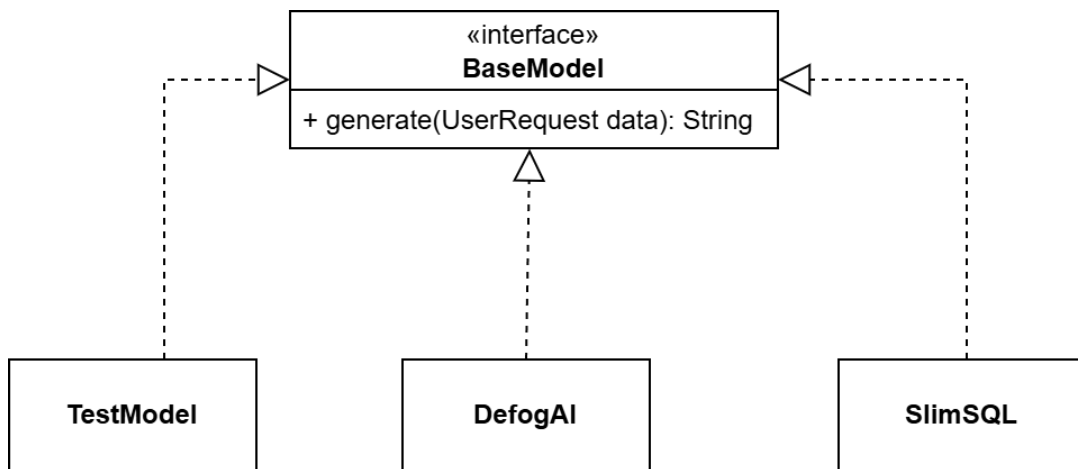


Figura 4.2: UML API

#### 4.3.1 Implementazioni di BaseModel

Ogni concretizzazione di `BaseModel` rappresenta un diverso modello di machine learning (ML) o una specifica regola di generazione delle query. Le classi che ereditano da `BaseModel` implementano il metodo `generate` con logiche e parametri personalizzati, adattando il processo di generazione della query al tipo di richiesta e di modello utilizzato.

#### 4.3.2 Test e validazione tramite TestModel

Una delle concretizzazioni di `BaseModel` è `TestModel`, progettata per eseguire test e validare il corretto funzionamento del sistema. In un ambiente di test controllato, `TestModel` simula la generazione delle query, permettendo

di verificare che il modello risponda correttamente alle richieste, fornendo un importante strumento per il debugging e la validazione.

## 4.4 Backend

Nel backend si è adottato il paradigma ad <sup>7</sup>eventi offerto da JavaScript, con l'obiettivo di garantire una chiara separazione tra la logica di business e gli elementi visivi.

Questo approccio offre diversi vantaggi, tra cui una riduzione dell'accoppiamento tra i moduli. Nella programmazione orientata agli eventi, i produttori e i consumatori di eventi operano in modo indipendente: i produttori generano e pubblicano eventi senza preoccuparsi dei consumatori, mentre questi ultimi si iscrivono agli eventi senza conoscere i dettagli della loro origine.

Questa separazione consente di apportare modifiche localizzate a parti del sistema senza impatti significativi sull'intera applicazione, facilitando così l'aggiornamento o la sostituzione di singoli componenti.

---

<sup>7</sup><https://stack.convex.dev/event-driven-programming>

### 4.4.1 Business logic

La business logic è articolata nelle classi mostrate in figura (4.3):

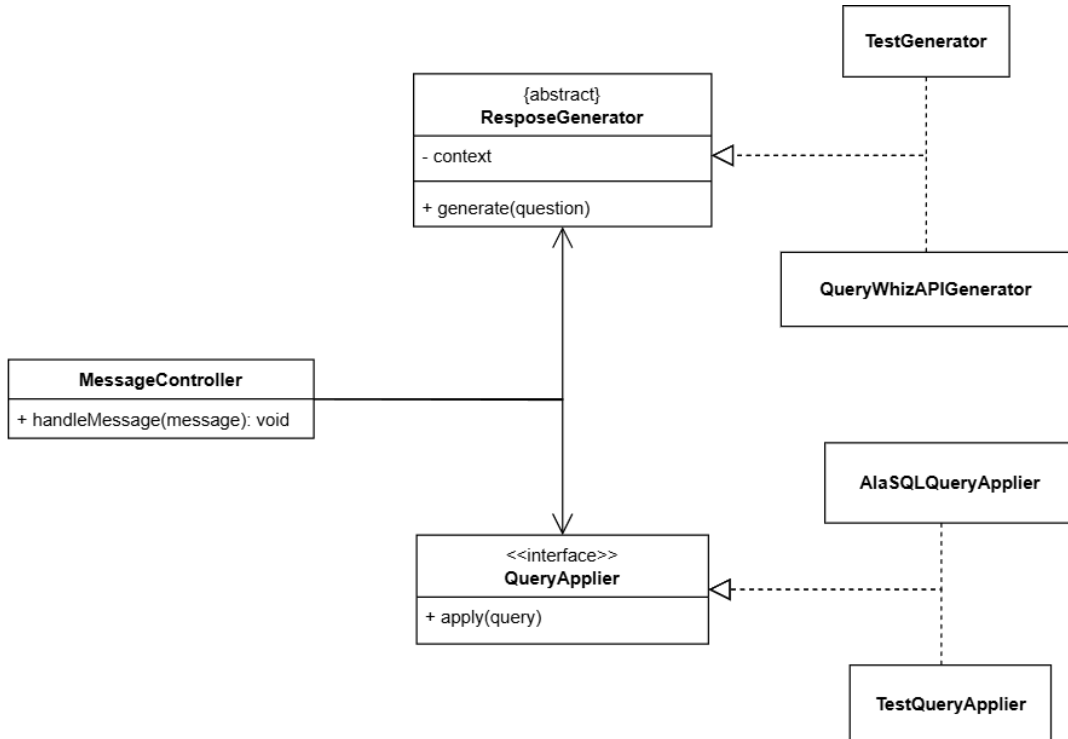


Figura 4.3: UML Backend

- **MessageController:**

- Classe principale responsabile della gestione dei messaggi. Ha un metodo pubblico `handleMessage(message)`, che coordina il flusso tra i vari componenti per trasformare il messaggio utente, e generare la query SQL.

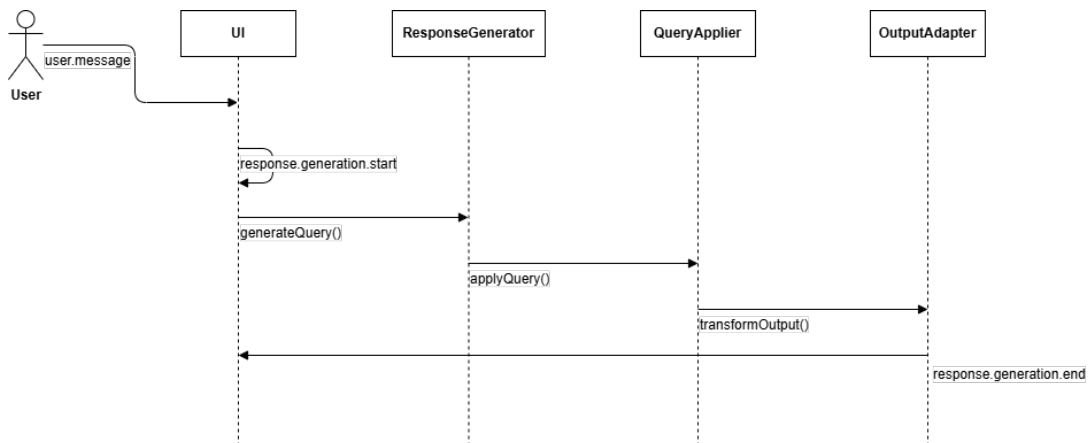
- **ResponseGenerator (classe astratta):**

- Una classe astratta che rappresenta la logica per generare risposte. Contiene un attributo `context`, rappresentante lo schema della tabella, e un metodo astratto `generate(question)`.
- Le implementazioni concrete includono:
  - \* **TestGenerator:** utilizzato per scopi di test, generando query di esempio.

- \* **QueryWhizAPIGenerator**: implementazione principale per la generazione delle query andando a richiamare la API.
- **QueryApplier (interfaccia)**:
  - Un’interfaccia che definisce il metodo `apply(query)` per applicare la query SQL generata ai dati.
  - Le implementazioni includono:
    - \* **TestQueryApplier**: utilizzato per testare le query.
    - \* **AlaSQLQueryApplier**: utilizzato per eseguire le query SQL su AlaSQL.

#### 4.4.2 Flusso generale

Il *diagramma di sequenza*  $\mathcal{G}$  è mostrato nell’immagine 4.4.



**Figura 4.4:** Diagramma di sequenza

1. Viene lanciato l’evento “user.message” per indicare che l’utente ha fatto una domanda e quindi aggiornare l’interfaccia.
2. Viene lanciato l’evento “response.generation.start” per indicare che il sistema sta processando la domanda e mostrare un indicatore all’utente.
3. Una volta formattata la domanda, viene richiamato **ResponseGenerator**, ossia la nostra API, per generare la query corrispondente alla domanda utente.

4. La query generata viene passata al **QueryApplier** per essere applicata ai dati.
5. Viene richiamato l'adattatore in output per trasformare i dati in output nel formato necessario per essere mostrato all'utente.
6. Infine, viene lanciato l'evento "response.generation.end" per indicare che il sistema ha terminato la generazione e mostrare i risultati all'utente.

# Capitolo 5

## Verifica e validazione

Di seguito sono riportati i test eseguiti per verificare il corretto funzionamento del software, che vengono eseguiti automaticamente durante il processo di *CI/CD<sub>G</sub>*.

### 5.1 Test di Unità

I test di unità verificano il corretto funzionamento delle singole componenti del sistema in isolamento. Lo scopo è assicurare che funzioni, metodi o classi producano il risultato atteso per input specifici, indipendentemente dall'ambiente esterno.

Per facilitare tali test, vengono utilizzati *Mock<sub>G</sub>* e *stub<sub>G</sub>*.

Tabella 5.1 specifica i test di unità effettuati.

<b>ID</b>	<b>Descrizione</b>	<b>Stato</b>
TU1	Verifica che l'endpoint <code>/api/v1/status</code> risponda con lo stato "ok".	Passato
TU2	Testa la generazione corretta di una query SQL utilizzando un modello mock.	Passato
TU3	Testa la gestione delle eccezioni lanciate durante la generazione della query da parte del modello.	Passato
TU4	Verifica la gestione degli errori di validazione in caso di input non valido.	Passato
TU5	Controlla che il logger venga chiamato correttamente durante una richiesta valida.	Passato
TU6	Testa la creazione di una risposta di successo da parte della classe <code>Response</code> .	Passato
TU7	Testa la creazione di una risposta di errore da parte della classe <code>Response</code> .	Passato

---

**Tabella 5.1:** Tabella dei test di unità.

## 5.2 Test di Integrazione

I test di integrazione verificano che le diverse componenti del sistema lavorino correttamente insieme. Lo scopo è assicurarsi che i moduli integrati producano risultati coerenti e rispettino i requisiti.

Tabella 5.2 specifica i test di integrazione effettuati.

ID	Descrizione	Stato
TI1	Verifica il flusso completo dall'input alla generazione della query SQL.	Passato

**Tabella 5.2:** Tabella dei test di integrazione.

## 5.3 Test di Validazione

I test di validazione si concentrano sulla gestione di input non validi. Lo scopo è garantire che il sistema reagisca in modo appropriato, restituendo errori significativi e dettagliati.

Tabella 5.3 specifica i test di validazione effettuati.

ID	Descrizione	Stato
TV1	Verifica il comportamento con un input privo del campo "question".	Passato
TV2	Verifica il comportamento con un input privo del campo "context".	Passato
TV3	Verifica il comportamento con il campo "context" di tipo errato.	Passato
TV4	Testa il sistema con una richiesta vuota per garantire errori significativi.	Passato

**Tabella 5.3:** Tabella dei test di validazione.

## 5.4 Test Parametrizzati

I test parametrizzati consentono di testare molteplici combinazioni di input con lo stesso caso di test. Lo scopo è migliorare la copertura del codice riducendo la ripetizione nei casi di test.

Tabella 5.4 specifica i test parametrizzati effettuati.

ID	Descrizione	Stato
TP1	Verifica il comportamento con input privo del campo "context".	Passato
TP2	Verifica il comportamento con input privo del campo "question".	Passato
TP3	Verifica la gestione di un tipo non valido per il campo "question".	Passato
TP4	Verifica il comportamento con una richiesta completamente vuota, garantendo un errore significativo.	Passato

**Tabella 5.4:** Tabella dei test parametrizzati.

## 5.5 Test sui Logger

I test sui logger verificano che i messaggi di log vengano registrati correttamente durante l'esecuzione del codice. Lo scopo è garantire che i dettagli rilevanti siano tracciati e disponibili per il debugging.

Tabella 5.5 specifica i test sui logger effettuati.

ID	Descrizione	Stato
TL1	Controlla che il logger venga chiamato correttamente durante una richiesta valida.	Passato

**Tabella 5.5:** Tabella dei test sui logger.

# Capitolo 6

## Test LLM

La selezione del modello più adatto per la generazione di query SQL dipende da vari fattori, tra cui la complessità delle richieste, la precisione dei risultati e la capacità di comprendere il contesto.

È fondamentale scegliere un modello che soddisfi i seguenti requisiti:

- **Comprensione della struttura SQL:** il modello deve generare una sintassi SQL corretta, gestendo con precisione operazioni come join, sottoquery e filtri.
- **Adattamento al contesto:** il modello deve adattare le query al database specifico, considerando tabelle, colonne e relazioni.
- **Gestione del linguaggio naturale:** deve interpretare correttamente richieste in linguaggio naturale e tradurle in query SQL efficienti.
- **Supporto linguistico:** deve essere in grado di gestire richieste in lingua italiana.
- **Scalabilità:** deve operare efficacemente su richieste di complessità variabile senza compromettere le prestazioni.

## 6.1 Studio valutazione query generate

Per valutare i modelli, è stato adottato un approccio basato sul confronto tra le query generate e quelle attese, ispirandosi allo studio <sup>1</sup>«ESM+: Modern Insights into Perspective on Text-to-SQL Evaluation in the Age of Large Language Models».

Questo metodo considera due tipi principali di equivalenza:

- **Equivalenza semantica:** valuta la similarità nel significato o nei risultati delle query.

```
SELECT MAX(weight) FROM dogs;
```

```
SELECT weight FROM dogs ORDER BY weight DESC LIMIT 1;
```

**Codice 6.1:** Esempio di equivalenza semantica

La metrica **Execution Accuracy (EXE)** verifica se il risultato dell'esecuzione della query generata corrisponde a quello della query attesa. Tuttavia, questa metrica può produrre falsi positivi, poiché query semanticamente diverse possono restituire lo stesso risultato (6.1).

- **Equivalenza sintattica:** misura la similarità nella struttura o forma delle query.

```
SELECT name, age FROM users WHERE age > 18;
```

```
SELECT name, age FROM users WHERE age > 18;
```

**Codice 6.2:** Esempio di equivalenza sintattica

La metrica **Exact Set Matching Accuracy (ESM)** confronta le parole chiave e gli argomenti delle query. Sebbene più rigorosa di EXE, può

---

<sup>1</sup><https://arxiv.org/pdf/2407.07313v2>

generare falsi negativi, poiché query semanticamente equivalenti possono differire nella struttura sintattica (6.2).

## 6.2 Implementazione benchmark LLM

Discutendo con il proponente si è deciso di basarsi su dei test funzionali, simili a test di unità. Questi test eseguono le query generate dal modello in un database di prova e confrontano i risultati ottenuti con quelli attesi.

Gli errori vengono identificati in quattro casi specifici:

- errori di sintassi nella query;
- selezione di un numero di colonne inferiore a quanto richiesto (non viene considerato errore se vengono selezionate più colonne del necessario);
- differenze nei valori restituiti nelle colonne;
- ordinamenti errati nei risultati.

Le query utilizzate nei test aumentano progressivamente di complessità, consentendo di verificare le capacità del modello in scenari sempre più articolati.

Sono stati effettuati diversi benchmark, e il modello più performante è risultato essere “lama-3-sqlcoder-8b” sviluppato da <sup>2</sup>[defog.ai](https://defog.ai). Questo modello si basa su “Meta-Llama-3-8B-Instruct”, opportunamente ottimizzato tramite *[fine-tuning<sub>G</sub>](#)* <sup>3</sup> per applicazioni di analisi dei dati.

## 6.3 Conclusioni

I modelli attuali presentano limiti significativi nella gestione di query matematiche che richiedono ragionamenti complessi. Queste difficoltà derivano dalle limitazioni intrinseche degli algoritmi di apprendimento automatico, che faticano a eseguire inferenze logiche avanzate.

Ad esempio, molti errori commessi riguardano:

---

<sup>2</sup><https://defog.ai>

<sup>3</sup><https://huggingface.co/defog/llama-3-sqlcoder-8b>

- l'utilizzo erroneo della mediana al posto della media;
- omissione la clausola DISTINCT;
- errori nella costruzione di formule matematiche, generando risultati incorretti.

# Capitolo 7

## Requisiti soddisfatti

A seguito tutti i requisiti soddisfatti.

### 7.1 Requisiti funzionali obbligatori

La tabella [7.1](#) indica lo stato di completamento dei requisiti funzionali obbligatori:

## CAPITOLO 7. REQUISITI SODDISFATTI

---

<b>ID</b>	<b>Requisito</b>	<b>Stato</b>
FO-001	Essere attivato tramite un'azione dell'utente (ad esempio un elemento grafico o combinazione di tasti)	Soddisfatto
FO-002	Avere un'interfaccia utente chiara e intuitiva, con elementi grafici e pulsanti per facilitare l'interazione	Soddisfatto
FO-003	Feedback visivo all'utente durante la fase di generazione risposta	Soddisfatto
FO-004	Essere in grado di comprendere ed interpretare il linguaggio naturale per gestire una varietà di domande e richieste dell'utente	Soddisfatto
FO-005	La visualizzazione dei risultati deve avvenire tramite gli strumenti messi a disposizione da Zoom	Soddisfatto
FO-006	Deve essere in grado di effettuare operazioni di selezioni sui dati	Soddisfatto
FO-007	Deve essere in grado di gestire operazioni che includono raggruppamenti	Soddisfatto
FO-008	Deve essere in grado di gestire operazioni che includono condizioni	Soddisfatto

**Tabella 7.1:** Stato requisiti funzionali obbligatori

## 7.2 Requisiti funzionali desiderabili

La tabella 7.2 indica lo stato di completamento dei requisiti funzionali desiderabili:

---

ID	Requisito	Stato
FD-001	Essere accessibile a tutti gli utenti, inclusi quelli con disabilità, seguendo le linee guida di accessibilità web	Soddisfatto
FD-002	Permettere all'utente di personalizzare la visualizzazione dei risultati	Soddisfatto

---

**Tabella 7.2:** Stato requisiti funzionali desiderabili

## 7.3 Requisiti non funzionali obbligatori

La tabella 7.3 indica lo stato di completamento dei requisiti non funzionali obbligatori:

---

ID	Requisito	Stato
NFO-001	Validare gli input utente ed assicurare rispettino criteri ben definiti come tipo di dato, lunghezza, o formato	Soddisfatto
NFO-002	Fornire chiari messaggi di errore in caso di richieste non realizzabili	Soddisfatto

---

**Tabella 7.3:** Stato requisiti non funzionali obbligatori

## 7.4 Requisiti non funzionali desiderabili

La tabella 7.4 indica lo stato di completamento dei requisiti non funzionali desiderabili:

---

ID	Requisito	Stato
NFD-001	Autenticazione utenza: solo gli utenti con l'accesso a Zoom devono essere in grado di conversare con il chatbot	Non soddisfatto
NFD-002	Essere utilizzabile da mobile	Non soddisfatto

---

**Tabella 7.4:** Stato requisiti non funzionali desiderabili

# Capitolo 8

## Miglioramenti software

Il miglioramento di un chatbot richiede interventi mirati su diversi aspetti del sistema, considerando sia l'efficienza tecnica che l'esperienza utente. Di seguito sono elencate alcune possibili aree di espansione:

- **API sicura:** aggiornare il protocollo di comunicazione ad *HyperText Transfer Protocol over Secure Socket Layer (HTTPS)*<sub>G</sub> per garantire la sicurezza dei dati trasmessi.
- **Intent detection:** implementare un sistema per identificare se la domanda dell'utente è pertinente allo scopo del bot, ovvero il supporto a query in linguaggio naturale. Questo approccio consente di:
  - Avvertire l'utente in caso di richieste non pertinenti.
  - Terminare automaticamente la conversazione dopo un numero definito di messaggi non attinenti.
- **Gestione del contesto esteso:** integrare un sistema per recuperare automaticamente lo schema delle tabelle necessarie, facilitando la generazione di query per richieste più complesse.
- **Feedback loop:** in caso di errori di sintassi nelle query generate, reinoltrare la richiesta al modello includendo il messaggio di errore. Questo permetterebbe al modello di correggere automaticamente la query e riprovare.

# Capitolo 9

## Conclusioni

### 9.1 Conoscenze acquisite

Lo stage ha avuto l'obiettivo di ampliare le conoscenze nella gestione della codebase di una REST API. Ho imparato a creare endpoint per gestire richieste HTTP (GET, POST, PUT, DELETE) e a rispondere in modo appropriato, rispettando le convenzioni REST, come l'utilizzo di URI chiari, metodi HTTP corretti e codici di stato adeguati per indicare il successo o il fallimento delle operazioni. La codebase risultante è modulare, ben organizzata e documentata.

Inoltre, ho acquisito esperienza nell'utilizzo di Docker, fondamentale in contesti di microservizi per l'isolamento dei singoli servizi. Ho configurato e gestito diversi container, garantendo la loro comunicazione attraverso reti Docker.

Infine, la configurazione di Nginx come reverse proxy mi ha permesso di ottimizzare la gestione del traffico in ingresso, bilanciando le richieste tra diversi server backend tramite tecniche di load balancing.

## 9.2 Valutazione personale

Il progetto si è rivelato meno complesso del previsto ed è stato tutto sommato soddisfacente. Tuttavia, avrei preferito lavorare su un progetto aziendale già in produzione, per osservare da vicino i processi e le pratiche che lo supportano. In particolare, mi sarebbe piaciuto comprendere come i professionisti affrontano le sfide architetturali e gestiscono i compromessi, soprattutto in presenza di legacy code; come sviluppano ed eseguono i test nelle loro pipeline CI/CD; e come si occupano del deployment, del roll-out di nuove versioni e della gestione della retrocompatibilità, quando necessaria.

Un aspetto trascurato in questo caso è la documentazione. Sebbene sia comprensibile che mantenerla aggiornata sia costoso e complesso, soprattutto in contesti soggetti a frequenti cambiamenti, un documento che descriva le principali funzionalità e l'architettura generale sarebbe estremamente utile sia per i nuovi arrivati, che per il lungo termine; riducendo la dipendenza dai colleghi più esperti.

Zoom, essendo un software con circa vent'anni di storia, rifletteva una fase evolutiva dell'ingegneria del software in cui molte pratiche attuali non erano ancora consolidate. È comprensibile che un refactor completo sarebbe stato proibitivo in termini di costi, tempi e necessità di garantire il supporto continuo agli utenti. In contesti come quelli di aziende consolidate, ad esempio Zucchetti, la cautela nelle modifiche ai sistemi critici è inevitabile e giustificata dalla grandezza e complessità dei progetti.

Nonostante queste considerazioni, ritengo questa esperienza preziosa, poiché mi ha offerto una prospettiva realistica sulla progettazione software nel nostro territorio. Il settore software, a mio avviso, è spesso sottovalutato: bilanciare le richieste degli investitori con la qualità e la velocità di sviluppo è una sfida, poiché non tutti comprendono appieno il lavoro che si svolge dietro le quinte. Investire tempo e risorse per sviluppare software ben progettato porterebbe vantaggi sia agli investitori, che otterrebbero nuove funzionalità più rapidamente, sia agli sviluppatori, che troverebbero più agevole implementarle.

# Acronimi e abbreviazioni

**API** Application Program Interface. [2](#)

**HTTPS** HyperText Transfer Protocol over Secure Socket Layer. [40](#)

**JSON** JavaScript Object Notation. [13](#)

**ML** Machine Learning. [9](#)

**NLP** Natural Language Processing. [10](#)

**XML** eXtensible Markup Language. [13](#)

# Glossario

**CI/CD** Queste pratiche, combinate, riducono i rischi di errore, aumentano la qualità e permettono un ciclo di sviluppo più rapido e continuo.. [26](#)

**CRUD** Acronimo per Create, Read, Update, Delete, le quattro operazioni basilari per la gestione dei dati.. [14](#)

**diagramma di sequenza** Rappresenta l'interazione tra oggetti o componenti di un sistema nel tempo, mostrando lo scambio di messaggi e l'ordine delle operazioni. [24](#)

**fine-tuning** Processo di adattamento di un modello di machine learning pre-addestrato a un nuovo compito specifico, continuando l'addestramento su un nuovo set di dati rilevante.. [33](#)

**Framework** Struttura di sviluppo predefinita che offre strumenti e linee guida per costruire applicazioni in modo più efficiente.. [10](#)

**Full duplex** Modalità di comunicazione in cui i dati possono essere trasmessi in entrambe le direzioni contemporaneamente.. [12](#)

**Librerie** Collezioni di funzioni, classi o metodi predefiniti che possono essere riutilizzati da un programma per eseguire compiti specifici.. [10](#)

**Mock** Oggetti che riproducono il comportamento di altri oggetti non ancora implementati.. [26](#)

**Multiplexing** Tecnica di trasmissione di più segnali o flussi di dati su un unico canale.. [12](#)

**Protobufs** Sistema di serializzazione sviluppato da Google per strutturare i dati in un formato binario compatto.. [12](#)

**Query SQL** Un'istruzione utilizzata per recuperare, inserire, aggiornare o cancellare dati da un database SQL. [9](#)

**RESTful API** Un'API che aderisce ai principi del REST (Representational State Transfer) per consentire la comunicazione tra sistemi tramite il protocollo HTTP. [4](#)

**Schema** La struttura che definisce l'organizzazione dei dati all'interno di un database, inclusi tabelle, campi e relazioni tra essi. [9](#)

**stub** Porzione di codice utilizzata per simulare il comportamento di funzionalità software (come una routine su un sistema remoto) e può fungere anche da temporaneo sostituto di codice ancora da sviluppare.. [26](#)

# Bibliografia

## Articoli

Ascoli, Benjamin G. «ESM+: Modern Insights into Perspective on Text-to-SQL Evaluation in the Age of Large Language Models». In: *none* (2024), pp. 1–2.  
URL: <https://arxiv.org/pdf/2407.07313v2>.

# Sitografia

*AlaSQL*. URL: <https://github.com/AlaSQL/alasql>.

*Docker*. URL: <https://docs.docker.com/desktop>.

*Flask*. URL: <https://flask.palletsprojects.com/>.

*Nginx*. URL: <https://nginx.org/en/>.

*Pydantic*. URL: <https://docs.pydantic.dev/>.

*Swagger*. URL: <https://swagger.io/>.

*Zucchetti*. URL: <https://www.zucchetti.it/it/cms/home.html>.