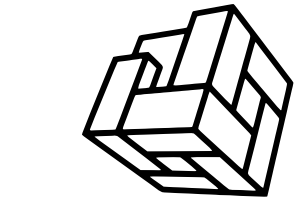
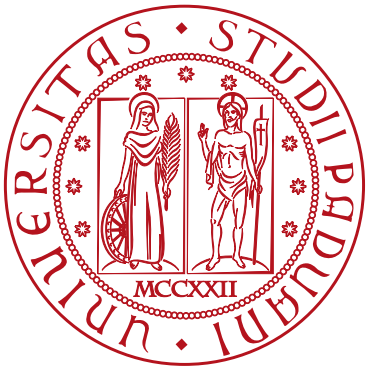


PEBKAC
Gruppo: 11
Email: pebkacswe@gmail.com
Docs: https://pebkac-swe-group-11.github.io
GitHub: https://github.com/PEBKAC-SWE-Group-11



Università degli Studi di Padova
Corso di Laurea: Informatica
Corso: Ingegneria del Software
Anno Accademico: 2024/2025

Specifica Tecnica

Informazioni sul documento:

Verificatori	Tommaso Zocche
	Derek Gusatto
	Matteo Piron
Redattori	Derek Gusatto
	Tommaso Zocche
	Ion Bourosu
	Alessandro Benin
Uso	Esterno
Destinatari	Tullio Vardanega
	Riccardo Cardin

Registro delle versioni

Versione	Data	Autore	Ruolo	Descrizione
1.0.0	2025-04-05	Derek Gusatto	Responsabile	Approvazione e rilascio
0.9.0	2025-04-05	Tommaso Zocche	Verificatore	Verifica finale
0.8.0	2025-04-05	Derek Gusatto	Verificatore	Verifica §3.3.1 Frontend, diagramma delle classi
0.7.1	2025-04-04	Ion Bourosu	Amministratore	Aggiornamento §3.3.1 Frontend, aggiunto diagramma delle classi
0.7.0	2025-04-04	Tommaso Zocche	Verificatore	Verifica ultime aggiunte
0.6.1	2025-04-04	Derek Gusatto	Programmatore	Aggiornata struttura documento, §3.3.3 Database, immagini ER ed Esagonale
0.6.0	2025-04-04	Derek Gusatto	Verificatore	Verifica §Backend e struttura documento
0.5.1	2025-04-03	Tommaso Zocche	Programmatore	Aggiunte in §Backend
0.5.0	2025-04-01	Derek Gusatto	Verificatore	Verifica §Frontend
0.4.1	2025-03-12	Ion Bourosu	Progettista	§Frontend
0.4.0	2025-04-01	Derek Gusatto	Verificatore	Verifica precendenti
0.3.3	2025-03-30	Alessandro Benin	Progettista	Descrizione Container
0.3.2	2025-03-27	Alessandro Benin	Progettista	Tracciamento Requisiti
0.3.1	2025-03-24	Alessandro Benin	Progettista	Backend
0.3.0	2025-03-09	Matteo Piron	Verificatore	Verifica §Frontend
0.2.1	2025-03-08	Ion Bourosu	Responsabile	Frontend §Architettura di sistema
0.2.0	2025-02-28	Matteo Piron	Verificatore	Verifica §Architettura di sistema
0.1.1	2025-03-03	Derek Gusatto	Progettista	§Architettura di sistema
0.1.0	2025-02-28	Matteo Piron	Verificatore	Verifica §Introduzione, §Tecnologie
0.0.5	2025-02-28	Ion Bourosu	Progettista	Descrizione Python e revisione Flask, Angular e Scrapy
0.0.4	2025-02-27	Derek Gusatto	Progettista	Docker, PostgreSQL, SQL
0.0.3	2025-02-26	Alessandro Benin	Progettista	Descrizione di Ollama, JSON, YAML

Versione	Data	Autore	Ruolo	Descrizione
0.0.2	2025-02-26	Ion Bourosu	Progettista	Descrizione di Flask, Angular e Scrapy
0.0.1	2025-02-25	Derek Gusatto	Responsabile	Prima impostazione, Introduzione, Tecnologie (inizio)

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Scopo del documento

Il presente documento ha l'obiettivo di definire in maniera esaustiva e comprensibile gli aspetti tecnici chiave del progetto "Vimar GENIALE". Il fine principale di questa risorsa è fornire una descrizione dettagliata e approfondita dell'architettura$_G$ implementativa del sistema$_G$.
Nel contesto dell'architettura$_G$ è prevista un'analisi approfondita che si estenda anche al livello di design più basso, includendo la definizione e la spiegazione dei design pattern$_G$ e dei linguaggi usati nel progetto. Gli obiettivi del presente documento sono tre: motivare le scelte di sviluppo adottate, fornire al gruppo una guida fondamentale per l'attività di codifica e monitorare la copertura dei requisiti identificati nel documento Analisi dei Requisiti V2.0.0.

1.2 Scopo del prodotto

Il progetto "Vimar GENIALE" mira a sviluppare un'applicazione intelligente che supporti installatori elettrici nell'uso di dispositivi Vimar$_G$, facilitando l'accesso alle informazioni tecniche sui prodotti, rispondendo a domande poste in linguaggio naturale. La tecnologia alla base prevede l'uso di modelli di LLM$_G$ e di tecniche RAG$_G$, con una struttura di gestione basata su container$_G$ e integrata in un ambiente cloud$_G$. Il sistema include tre componenti principali: un applicativo web responsive$_G$, un applicativo server$_G$ e un'infrastruttura cloud-ready$_G$.

1.3 Glossario

Per evitare ambiguità relative al linguaggio utilizzato nei documenti, viene fornito il Glossario V2.0.0, nel quale si possono trovare tutte le definizioni di termini che hanno un significato specifico che vuole essere disambiguato. Tali termini sono marcati con una G a pedice.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- Regolamento del progetto didattico
  https://www.math.unipd.it/ tullio/IS-1/2024/Dispense/PD1.pdf
  (Ultimo accesso 2024-11-20)
- ISO/IEC 12207:1995 Information technology - Software life cycle processes
  https://www.math.unipd.it/ tullio/IS-1/2010/Approfondimenti/A03.pdf
  (Ultimo accesso 2025-02-22)

1.4.2 Riferimenti informativi

- Capitolato C2
  https://www.math.unipd.it/ tullio/IS-1/2024/Progetto/C2.pdf
  (Ultimo accesso 2025-03-26)

- Capitolato C2 - slides
  http://math.unipd.it/ tullio/IS-1/2024/Progetto/C2p.pdf
  (Ultimo accesso 2024-12-19)
- Documentazione$_G$ GitHub$_G$
  https://docs.github.com/en
  (Ultimo accesso 2024-11-14)

2 Tecnologie

In questa sezione si espone una panoramica delle tecnologie adottate. Si ha la descrizione delle tecnologie e dei linguaggi di programmazione utilizzati, delle librerie$_G$ e dei framework$_G$ necessari, oltre che delle infrastrutture realizzate.

2.1 Docker

Docker$_G$ è una piattaforma per la containerizzazione che consente di impacchettare, distribuire ed eseguire applicazioni in ambienti isolati e riproducibili. Il nostro progetto utilizza Docker per garantire una gestione efficiente delle dipendenze, facilitare il deployment e migliorare la portabilità del sistema su diverse infrastrutture. L'uso di Docker permette di standardizzare l'ambiente di sviluppo e produzione, riducendo problemi di compatibilità tra sistemi operativi e versioni di software. I container definiti nel progetto includono il backend, il frontend e il database PostgreSQL, ciascuno configurato per garantire modularità e scalabilità. Rispetto ad altre alternative (e.g. Vagrant) Docker è stato scelto per la sua efficienza e facilità d'uso.

2.1.1 Vantaggi

- **Isolamento e portabilità**: Ogni servizio è eseguito in un container indipendente, evitando conflitti tra dipendenze e facilitando il deployment su diverse piattaforme.
- **Scalabilità**: L'architettura a container permette di scalare orizzontalmente i servizi in base al carico di lavoro, migliorando le prestazioni del sistema.
- **Riproducibilità**: Grazie alla definizione di immagini Docker e file di configurazione YAML, l'ambiente di esecuzione può essere ricreato in modo identico su qualsiasi macchina.
- **Facilità di gestione e automazione**: L'integrazione con Docker Compose permette di avviare, arrestare e configurare l'intero sistema con un singolo comando, migliorando l'efficienza nella gestione dell'infrastruttura.

2.1.2 Svantaggi

- **Consumo di risorse**: L'esecuzione di più container può richiedere un utilizzo elevato di CPU e memoria, specialmente in ambienti con risorse limitate.
- **Complessità nella configurazione**: La gestione della rete tra container, la persistenza dei dati e la sicurezza richiedono una configurazione attenta per evitare problemi di isolamento e performance.
- **Overhead nella virtualizzazione**: Sebbene più leggero rispetto alle macchine virtuali, Docker introduce un livello di astrazione che può impattare leggermente le prestazioni rispetto all'esecuzione nativa.

2.2 Flask

Flask$_G$ è un micro-framework web per Python, progettato per facilitare lo sviluppo di applicazioni web in modo semplice e veloce. Non richiede strumenti o librerie particolari per funzionare e non impone una struttura rigida al progetto. Abbiamo deciso di utilizzarlo per implementare un API RESTful$_G$, che segue i principi REST$_G$, per gestire sessioni, conversazioni, messaggi e feedback, integrando anche funzionalità di intelligenza artificiale per generare risposte a domande attraverso un modello di linguaggio LLM$_G$ e un sistema di embedding$_G$. Nel nostro sistema, Flask è la parte del back-end$_G$ che permette la connessione con un front-end$_G$ per la gestione delle interazioni con l'utente, grazie a un sistema di routing$_G$ che definisce delle rotte per gestire le richieste HTTP$_G$ (GET, POST, PUT e DELETE). A differenza di framework più complessi come Django$_G$, Flask non impone una struttura rigida, permettendo di personalizzare l'architettura per ottimizzare le prestazioni delle operazioni di embedding e delle richieste HTTP.

2.2.1 Vantaggi

- Non impone una struttura rigida, permettendo al team di sviluppo di organizzare il codice senza troppi vincoli e di integrare librerie strettamente necessarie;
- È semplice, quindi adatto a che si interfaccia per la prima volta allo sviluppo web con Python$_G$;
- L'integrazione con tecnologie avanzate come LLM$_G$ e sistemi di embedding$_G$, dimostra la sua capacità di supportare soluzioni complesse.

2.2.2 Svantaggi

- La flessibilità d'altro canto può risultare un rallentamento, essendo che manca la standardizzazione nel codice, specialmente in un contesto in cui il team è inesperto.

2.3 Scrapy

Scrapy$_G$ è un framework$_G$ open-source per Python$_G$ progettato specificamente per il web scraping$_G$, ovvero l'estrazione di dati da siti web, nel nostro caso dal sito di Vimar. È basato su un'architettura asincrona$_G$, che lo rende adatto per il crawling$_G$ di grandi volumi di dati, riducendo i tempi di esecuzione rispetto a strumenti più semplici come BeautifulSoup$_G$. Altro motivo per cui è stato scelto di utilizzare Scrapy è dato dall'approccio che abbiamo deciso di adottare, che vede lo scraping come componente a sè stante. Dunque, ci da la possibilità di esportare i dati in formati strutturati come JSON$_G$ da dove vengono poi caricati nel database$_G$.

2.3.1 Vantaggi

- L'architettura asincrona ci permette di gestire un grande numero di richieste contemporaneamente;
- La separazione fra la gestione delle richieste, l'elaborazione dei dati e il salvataggio dei risultati facilitano lo sviluppo e la manutenzione del codice;
- Permette di salvare i dati in formati diversi (JSON$_G$, CSV$_G$ o XML$_G$) e di integrarli facilmente con database o altre applicazioni.

2.3.2 Svantaggi

- La scarsa esperienza nell'ambito del web scraping$_G$ può rallentare il processo, richiedendo più tempo per l'apprendimento di tale tecnologia;
- Si possono incontrare rallentamenti nel caso di politiche di crawling$_G$ specifiche o blocchi da parte dei siti web.

2.4 Ollama

Ollama$_G$ è una piattaforma leggera ed efficace per eseguire modelli di intelligenza artificiale in locale, con un'architettura ottimizzata che garantisce prestazioni elevate con un utilizzo ridotto di risorse. È più semplice e intuitiva rispetto ad LLM Studio, risultando ideale per il nostro progetto.

2.4.1 Vantaggi

- Più leggero ed efficiente, consumando meno risorse e offrendo migliori prestazioni su hardware$_G$ standard;
- Interfaccia$_G$ più semplice e intuitiva, facile da usare, anche senza configurazioni complesse.

2.4.2 Svantaggi

- Meno opzioni avanzate di personalizzazione per l'addestramento;
- Meno strumenti integrati per il monitoraggio e l'analisi delle prestazioni del modello.

2.5 PostgreSQL

PostgreSQL$_G$ è un sistema di gestione di database relazionale (RDBMS) open-source che garantisce elevate prestazioni, affidabilità e scalabilità. È stato scelto per il nostro progetto in quanto offre una gestione avanzata degli indici e la possibilità di estendere le sue funzionalità tramite moduli aggiuntivi. PostgreSQL supporta inoltre l'archiviazione e la gestione di dati non strutturati grazie a tipi di dati avanzati come JSONB e l'estensione VECTOR$_G$, fondamentale per l'elaborazione di dati ad alta dimensionalità, come gli embedding generati da modelli di intelligenza artificiale.

2.5.1 Vantaggi

- **Affidabilità e consistenza**: PostgreSQL garantisce la conformità agli standard ACID (Atomicità, Consistenza, Isolamento, Durabilità), assicurando l'integrità dei dati anche in presenza di errori o interruzioni del sistema;
- **Supporto per estensioni avanzate**: PostgreSQL può essere esteso con moduli come PostGIS (per dati geografici) e pgvector, che consente di memorizzare e gestire vettori ad alta dimensionalità, rendendolo adatto all'integrazione con sistemi di ricerca semantica e modelli di intelligenza artificiale;

- **Ottimizzazione delle query**: Il motore di PostgreSQL utilizza un planner sofisticato in grado di ottimizzare le query grazie all'uso di indici B-Tree, GIN, GiST e BRIN, migliorando le prestazioni nelle operazioni di ricerca e aggregazione;
- **Scalabilità e parallelismo**: Supporta replica sincrona e asincrona, partizionamento delle tabelle e parallelizzazione delle query, garantendo elevate prestazioni in ambienti distribuiti e con grandi volumi di dati.

2.5.2 Svantaggi

- **Maggiore complessità di gestione**: Rispetto ad altri RDBMS, PostgreSQL richiede una configurazione più avanzata per ottenere prestazioni ottimali, in particolare nella gestione degli indici e della cache;
- **Consumo di risorse**: Le operazioni di scrittura e indicizzazione possono risultare più costose in termini di memoria e CPU rispetto ad alternative come MySQL o SQLite, specialmente per applicazioni con carichi di lavoro elevati.

2.6 Angular

Angular$_G$ è un framework open-source per lo sviluppo di applicazioni web single-page (SPA$_G$). È basato su TypeScript$_G$, un superset di JavaScript$_G$. Nel nostro sistema è utilizzato per il front-end$_G$, permettendo una gestione efficiente delle interazioni utente e una comunicazione fluida con il back-end$_G$ tramite API RESTful$_G$. Si tratta di una comunicazione asincrona$_G$ (di default), grazie all'uso di RxJS$_G$ e degli Observables$_G$. Quest'approccio sfrutta il pattern reattivo$_G$ per gestire le oprazioni come le richieste HTTP$_G$. A differenza di framework più leggeri come React$_G$ o Vue.js$_G$, Angular ha una struttura più organizzata con strumenti come il routing$_G$ e la comunicazione con il back-end. Il che è molto utile per il progetto che si sta sviluppando, dove è richiesta una gestione avanzata delle interazioni utente e una perfetta integrazione con un API RESTful per il modello di linguaggio LLM$_G$ e il database vettoriale$_G$.

2.6.1 Vantaggi

- Utilizza un sistema di moduli e componenti che favorisce il riciclo del codice e una struttura organizzata;
- Si basa sul two-way of data binding$_G$, ovvero la sincronizzazione automatica dei dati tra modello$_G$ e vista$_G$;
- Grazie all'utilizzo di Angular CLI$_G$ alcune attività come la creazione del progetto e la generazione delle componenti, vengono automatizzate;
- Si integra facilmente con il back-end tramite sevizi HTTP, redendolo un'ottima scelta per applicazioni che richiedono una comunicazione costante con il server.

2.6.2 Svantaggi

- Richiede molto tempo per l'apprendimento a causa della sua complessità e della necessità di imparare concetti come moduli, servizi e dependency injection$_G$;

- Le app sviluppate con Angular possono richiedere molto spazio;
- Il ciclo di aggiornamente è molto frequente perciò è possibile che ci sia la necessità di intervenire per mantenrlo compatibile alle nuove versioni.

2.7 Linguaggi e formato dati

2.7.1 Python

Python$_G$ è un linguaggio di programmazione ad alto livello. Supporta la programmazione orientata agli oggetti, procedurale e funzionale. È adatto per l'utilizzo in ambiti come lo sviluppo web e l'intelligenza artificile. Nel nostro progetto la scelta di Python come linguaggio per il back-end$_G$ è stata esplicitamente richiesta nel capitolato, dunque abbiamo escluso linguaggi diversi per lo sviluppo del back-end.

2.7.1.1 Vantaggi

- Offre una sintassi intuitiva e vicina al linguaggio naturale, che lo rende ottimale per un team di sviluppo intesperto;
- Si adatta perfettamente con le altre tecnologie scelte, in particolare Flask$_G$;
- Grazie alle molteplici librerie offerte semplifica lo sviluppo, riducendo i tempi per l'implementazione di funzionalità complesse.

2.7.1.2 Svantaggi

- Rischia di diventare un collo di bottiglia in un'applicazione come quella che si sta sviluppando per Vimar$_G$, dove sono richieste operazioni complesse e veloci.

2.7.2 SQL

SQL (Structured Query Language)$_G$ è il linguaggio utilizzato per la gestione e manipolazione dei dati in PostgreSQL. Il nostro progetto sfrutta SQL per la definizione dello schema del database, la gestione delle relazioni tra le entità e l'esecuzione di operazioni di lettura, scrittura, aggiornamento ed eliminazione dei dati. Abbiamo scelto SQL$_G$ rispetto a NoSQL$_G$ perché garantisce affidabilità, coerenza e query avanzate, fondamentali per il nostro progetto.

2.7.2.1 Vantaggi

- Strutturato e organizzato, Ideale per dati relazionali con schemi ben definiti;
- Query$_G$ potenti e precise, grazie a JOIN$_G$, filtri e funzioni avanzate;
- Ampio supporto, Usato in molti database$_G$.
- L'integrazione con pgvector permette di eseguire operazioni di similarità su dati rappresentati come vettori, abilitando funzionalità di nearest neighbor search per applicazioni di machine learning e retrieval aumentato.

2.7.2.2 Svantaggi

- Meno flessibile con dati non strutturati, NoSQL_G è più adatto per dati dinamici e documentali;
- Scalabilità orizzontale complessa, NoSQL_G può essere più efficiente per sistemi distribuiti su larga scala;
- Rigidità nella modifica dello schema, cambiare la struttura del database_G può essere complicato in ambienti SQL_G.

2.7.3 YAML

YAML (YAML Ain't Markup Language)$_\text{G}$ è un formato di serializzazione dati leggibile dall'uomo, utilizzato principalmente per configurazioni e automazione, come nei file Docker Compose_G. La sua sintassi basata sull'indentazione lo rende più leggibile rispetto ad altri formati, ma richiede attenzione alla formattazione.

2.7.3.1 Vantaggi

- Maggiore leggibilità grazie alla sintassi senza parentesi e virgolette;
- Supporta commenti, facilitando la documentazione delle configurazioni;
- Permette riferimenti e ancoraggi per riutilizzare parti del file_G.

2.7.3.2 Svantaggi

- Sensibile all'indentazione, aumentando il rischio di errori difficili da individuare;
- Parsing_G più lento rispetto ad altri formati come JSON_G;
- Meno diffuso per l'interscambio dati rispetto ad altri standard.

2.7.4 JSON

JSON_G è un formato leggero per lo scambio di dati tra applicazioni, ampiamente utilizzato nelle API_G e nella comunicazione tra servizi. La sua sintassi basata su coppie chiave-valore e array_G lo rende facile da elaborare e compatibile con la maggior parte dei linguaggi di programmazione.

2.7.4.1 Vantaggi

- Struttura semplice e facilmente interpretabile da macchine e sviluppatori;
- Parsing_G più lento rispetto ad altri formati come JSON;
- Standard per lo scambio di dati tra sistemi diversi.

2.7.4.2 Svantaggi

- Meno leggibile rispetto a YAML_G, specialmente in configurazioni complesse;
- Non supporta commenti, rendendo più difficile documentare il codice;
- Più rigido nella sintassi, con obbligo di virgolette e parentesi.

3 Architettura di sistema

3.1 Modello architetturale

Il sistema è stato progettato seguendo l'**architettura esagonale**, un modello che mira alla separazione tra business logic dell'applicazione e i servizi esterni, le fonti dati e le interfacce utente. Questa struttura pone il core_G al centro, circondato da ports_G che fungono da interfaccia tra il core_G e il mondo esterno.

Il **core_G** dell'applicazione è il fulcro del sistema, che contiene la logica di dominio e le regole di business. La sua progettazione punta ad evitare riferimenti diretti a dettagli tecnologici specifici, promuovendo l'indipendenza dal contesto esterno.

Le **ports_G** costituiscono il confine tra il core_G dell'applicazione e l'esterno, garantendo una comunicazione strutturata. Esistono due tipi di ports_G:

- Inbound Port (o Use Case): per le invocazioni del core_G da parte di componenti esterni, attraverso un'interfaccia definita. Sono in pratica dei punti di accesso al core_G e isolano la logica di dominio da implementazioni specifiche;
- Outbound Port: consentono al core_G di accedere a funzionalità esterne (e.g. interazione con librerie esterne_G o sistemi di persistenza). Mettono a disposizione un'astrazione che preserva l'indipendenza del core_G dai dettagli tecnici specifici.

I **services_G** costituiscono il livello più esterno dell'applicazione, fanno parte della business logic. L'implementazione dei services_G si concentra sulla logica di dominio, senza preoccuparsi degli aspetti tencologici specifici.

Gli **adapters_G** formano il livello più esterno dell'applicazione. Esistono due tipi di adapters_G:

- Input Adapters_G (o Controllers): invocano operazioni sulle Inbound Port, traducendo le azioni provenienti dall'esterno in chiamate alle ports_G in ingresso al core_G, rendendo le richieste esterne comprensibili per il core_G;
- Output Adapters_G: invocano operazioni sulle Outbound Port, traducendo le azioni del core_G in operazioni comprensibili per il mondo esterno.

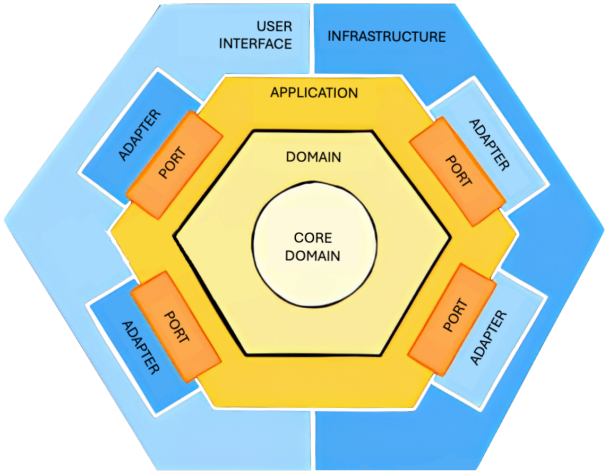


Figura 1: Rappresentazione dell'architettura esagonale

3.2 Componenti

L'architettura del sistema è suddivisa in:

- Frontend: si occupa di fornire un'interfaccia grafica all'utente per interagire col sistema. Inoltra le domande dell'utente al backend e visualizza i risultati;
- Backend: si occupa di elaborare le richieste degli utenti, interagendo con il sistema di persistenza e i servizi esterni, in particolare dialoga con il database vettoriale e con gli LLM;
- Database: è responsabile della memorizzazione della documentazione per la RAG e delle chat con relativi messaggi.

3.2.1 Assemblaggio delle componenti

Le componenti sono assemblate tramite Docker Compose, per facilitare l'esecuzione e la gestione di più container docker. In particolare sono stati prodotti i seguenti container:

- pgvector: espone l'istanza del database sulla porta 5432, permettendo al backend di accedere ai relativi dati;
- app: espone la componente di backend sulla porta 5001, dando al frontend la possibilità di accedere ai servizi offerti;
- frontend: espone l'applicazione web sulla porta 4200, dando la possibilità all'utente di connettersi e interagire col sistema;
- data processing: responsabile dello scraping, dell'indicizzazione e salvataggio nel DB.

3.3 Struttura del sistema

3.3.1 Frontend

- L'approccio adottato per lo sviluppo del front-end$_G$ si basa su una combinazione flessibile di pattern e pratiche tipiche del framework Angular$_G$, senza aderire rigidamente a un unico schema architetturale come MVC$_G$ o MVVM$_G$. Questa scelta consente di bilanciare modularità, scalabilità e semplicità nella gestione dello stato, adattandosi alle specifiche esigenze del progetto.
- La struttura dell'applicazione si articola in due aree funzionali ben distinte (utente e amministrativa), basate su componenti standalone$_G$ — come SidebarComponent, ChatboxComponent e AppComponent per l'area utente; AdminLoginComponent e AdminDashboardComponent per l'area amministrativa — supportati da servizi dedicati quali ApiService e ChatService. I componenti sono responsabili della presentazione, mentre i servizi gestiscono la comunicazione con il back-end$_G$ tramite API RESTful$_G$ (nel caso di ApiService) e centralizzano la logica di business e lo stato (come in ChatService).

- In questo modo, operazioni quali la creazione di conversazioni, l'invio di messaggi o la gestione dei feedback vengono delegate ai servizi, rendendo i componenti più leggeri e riutilizzabili. La gestione dello stato è un elemento chiave dell'architettura: l'introduzione di un servizio come ChatService, che utilizza il pattern Observer$_G$ tramite BehaviorSubject$_G$ di RxJS$_G$ per centralizzare conversazioni, messaggi e conversazione attiva, consente ai componenti di reagire dinamicamente ai cambiamenti attraverso gli observable. Questo garantisce un'interfaccia utente$_G$ reattiva e fluida, resa possibile grazie a strumenti asincroni come async/await$_G$ e firstValueFrom$_G$.

- La sicurezza è gestita attraverso meccanismi appropriati al contesto: sessioni utente anonime persistenti tramite localStorage$_G$ per l'area chat, e autenticazione basata su hash SHA-256$_G$ con token JWT$_G$ per l'area amministrativa, implementata tramite interceptor HTTP$_G$.

3.3.1.1 Componenti principali

AppComponent

Si tratta del componente padre che gestisce la corretta comunicazioni tra i due componenti figli SidebarComponent e ChatboxComponent (i quali fanno parte dei Componenti Principali). Coordina l'interfaccia grafica utente principale e gestisce la visibilità della Sidebar nella situazione in cui l'applicativo viene utilizzato da mobile attraverso:

- `toggleSidebar()` - pulsante visibile solo in modalità mobile che permette la visualizzazione della lista di conversazioni;

- `closeSidebarOnMobile()` - funzione che si attiva in seguito alla selezione di una delle conversazioni presenti nella lista o in seguito alla creazione di una nuova conversazione, lasciando spazio allo spazio dedicato ai messaggi.

ChatboxComponent

Primo dei due componenti figli di AppComponent il cui ruolo principale è quello di fornire all'utente la corretta visualizzazione e invio dei messaggi e dei feedback. Questo è possibile grazie ai metodi:

- `sendMessage()` - responabile dell'invio di un messaggio valido da parte dell'utente, ovvero un messaggio che non superi i caratteri limite o che non sia vuoto;

- `sendPositiveFeedback()` - gestione dei feedback positivi;

- `sendNegativeFeedback()` - gestione dei feedback negativi;

- `submitFeedback()` - conferma del feedback che si intende mandare, il quale può essere positivo o negativo, vuoto o commentato.

SidebarComponent

Componente responsabile della gestione e visualizzazione della lista di conversazioni. I metodi che permettono ciò sono i seguenti:

- `createNewConversation()` - crea una nuova conversazione e può essere invocato in due situazioni: quando si vuole creare una conversazione nuovo tramite il pulsante dedicato oppure quando si cerca di cancellare una conversazione attiva, ovvero una conversazione che l'utente sta utilizzando o visualizzando nel momento della sua cancellazione. Dunque, al posto della conversazione eliminata viene creata di default una nuova conversazione;

- `selectConversation()` - selezione di una conversazione esistente nella lista, la quale dopo la sua selezione viene contrassegnata come 'attivia';

- `deleteConversation()` - cancellazione di una delle conversazioni esistenti (il comportamento della funzione quando si cerca di cancellare una conversazione attiva è già stato descritto sopra)

I metodi descritti in questo paragrafo sono tutti gestiti dai servizi implementati nel frontend.

3.3.1.2 Servizi

ChatService

Come anticipato precedentemente, i metodi utilizzati nei componenti figli sono implementati in ChatService, che ha il compito di gestire la comunicazione tra le richieste dei componenti del frontend e il backend. I metodi principali sono:

- `initializeFromStorage()` - metodo chiave grazie al quale è possibile l'inizializzazione dell'applicativo attraverso il controlla della presenza o meno di un ID di sessione nel localStorage. Se questo è già presente allora vengono recuperate tutte informazioni dipendenti dal ID conservato grazie a `readSession(sessionId)` e viene aggiornata la data di accesso con `updateSession(sessionId)`, altrimenti se ne genera una nuova conversazione con `createNewSession()`;

- le altre funzioni sono già state discusse precedentemente.

ApiService

Mentre ChatService contiene l'implementazione dei metodi utilizzati dai componenti figli, ApiService contiene le chiamate HTTP ai metodi CRUD presenti nel backend. Infatti, ApiService è iniettato in ChatService, e fa da tramite tra frontend e backend attraverso le API che mantengono una forma compattibile con le API scritte nel backend. Come si può intuire le chiamte principali gestiscono sessione, messaggi e feedback, e la maggior parte di queste sono già state discusse precedentemente. Tuttavia, ce ne sono alcune da analizzare:

- `askQuestion(conversationId, question)` - viene chiamata all'interno del metodo `sendMessage()` in ChatService. Il suo scopo è quello di collegarsi all'algoritmo che genera la risposta nel backend. Si resta in attesa e non appena viene ritornata, si può visualizzare nella sezione dei messaggi della conversazione interessata;

- Admin

  Le seguenti chiamate HTTP sono tutte necessarie per l'area riservata dell'amministratore

- `adminLogin(password)` - responsabile dell'autenticazione tramite password da parte dell'amministrazione per poter accedere alla Dashboard. Si specifica che si tratta di un'implementazione mock, accordata con il proponente. Tale argomento verrà trattato più approfonditamente successivamente;
- `getAdminStats()` - richiede dal backend i dati relativi al totale delle conversazioni e dei feedback creati dagli utenti. Si tratta quindi del recupero delle statistiche necessarie all'amministratore per comprendere l'andamento dell'applicativo;
- `getFeedbackWithComments` - funzionalità necessaria a recuperare i feedback dei messaggi che sono stati inviati con un commento annesso. Necessario agli svliuppatori in una fase iniziale in cui si prende atto delle osservazioni degli utenti con lo scopo di migliorare il prodotto in fasi successive. Feedback con messaggio e commento sono visibili nella Dashboard insieme alle statistice.

3.3.1.3 Area amministratore

LoginComponent

Per raggiungere questa sezione dell'applciativo è necessario modificare l'URL aggiungendo `/admin/login` che porta l'amministratore all'interfaccia grafica dedicata all'autenticazione dove basta inserire la password per poter accedere in seguito alla Dashboard (`/admin/dashboard`). Le funzioni principali sono:

- `login()` - verifica la password inserita e autentica l'utente;
- `hashPassword` - calcolo dell'hash della password per la verifica.

Una volta autenticato, nel localStorage viene salvato il token generato durante l'accesso alla Dashboard.

DashboardComponent

Come già discusso, nelle Dashboard si possono visualizzare le statisctiche amministrative e i feedback commentati degli utenti. Ciò è possibili grazie a:

- `loadStats()` - carica le statistiche;
- `loadFeedbackComments` - carica i feedback commentati;
- `calcSatisfactionRate` - calcolo del tasso di soddisfazione degli utenti.

Dalla Dashboard si possono fare due azioni, ritornare all'area utilizzata da un utente generico, ovvero alle conversazioni, tramite pulsante specifico, ma così facendo il token generato durante l'autenticazione è ancora presente nel localStorage, quindi l'amministratore in momemnti successivi può accedere all Dashboard senza dover ripetere la fase di autenticazione. Altrimenti, tramite pulsante specifico può fare 'Logout', così rimuovendo il token dal localStorage.

AuthInterceptor

Intercetta le richieste HTTP per aggiungere token di autenticazione. Attraverso `intercept()` aggiunge l'header di autorizzazione alle richieste verso le rotte amministrative. È utilizzato globalmente per proteggere le rotte amministrative.

AdminGuard

Protegge le rotte amministrative verificando l'autenticazione. Controlla se l'utente è autenticato prima di permettere l'accesso a una rotta attraverso `canActivate()`. È utilizzato nelle definizioni delle rotte per proteggere le sezioni amministrative.

3.3.1.4 Design Pattern

L'applicazione front-end implementa specifici pattern di progettazione software che ne definiscono l'architettura complessiva. La scelta di questi pattern si allinea con le best practice di Angular e con le esigenze progettuali.

Pattern creazionali

- Singleton

  L'intero sistema di dependency injection$_G$ di Angular implementa implicitamente questo pattern. Tutti i servizi (ApiService e ChatService) sono dichiarati con @InjectableprovidedIn: 'root', garantendo un'unica istanza per l'intera applicazione. Grazie a questo sistema si riesce la centralizzazione dello stato dell'applicazione, la prevenzione di inconsistenze e l'ottimizzazione delle risorse attraverso il riuso delle istanze di servizio.

Pattern strutturali

- Decorator

  Si usa il sistema nativo di decoratori TypeScript/Angular$_G$ (@Component, @Injectable, @Input e @Output) per estendere dinamicamente il comportamento delle classi e di AuthInterceptor per decorare le richieste HTTP. Così facendo si separano le responsabilità e si riutilizza il codice.

- Façade

  ApiService astrae le complessità delle chiamate HTTP e ChatService nasconde i dettagli implementativi della gestione delle conversazioni. Il vantaggio è il disaccoppiamento tra logica di business e interfaccia utente.

Pattern comportamentali

- Observer

  Uso estensivo di BehaviorSubject e Observable di RxJS per gestire flussi di dati asincroni e aggiornamenti dell'interfaccia utente.

- Mediator

  AppComponent funge da mediatore per la comunicazione tra componenti figli come SidebarComponent e ChatboxComponent.

3.3.1.5 Diagramma delle classi

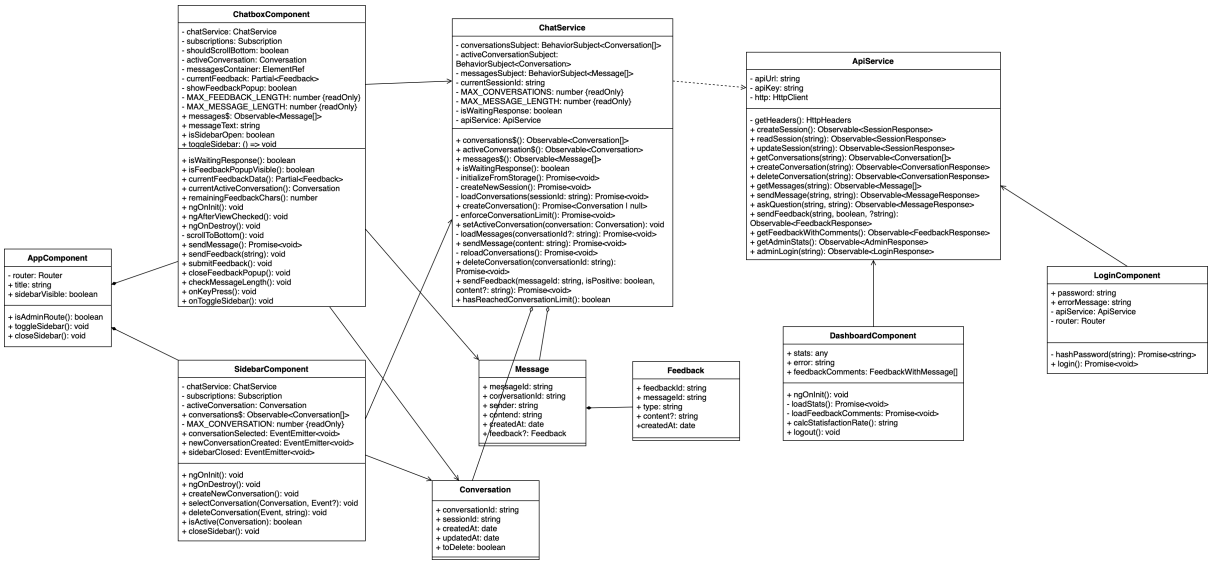


Figura 2: Schema delle classi utilizzate nel frontend

3.3.2 Backend

Il sistema è architettato secondo il pattern esagonale (Ports and Adapters), dove il Core costituisce il dominio centrale dell'applicazione. Questa componente rappresenta il nucleo del sistema, caratterizzato da un isolamento completo dalle dipendenze esterne. Al suo interno sono contenute le regole di business fondamentali e le entità che definiscono i concetti chiave del dominio: le conversazioni, i messaggi e le sessioni. Il Core opera come un'unità autonoma, implementando la logica di business in modo indipendente dalle modalità di input/output dei dati.
La comunicazione tra il Core e l'ambiente esterno è gestita attraverso le Porte, che definiscono le interfacce di interazione. Queste costituiscono i contratti formali attraverso i quali il Core stabilisce le modalità di comunicazione desiderate, astraendosi dalle implementazioni concrete. Le Porte rappresentano le specifiche tecniche che definiscono come il Core intende interagire con l'esterno, delegando la responsabilità dell'implementazione alle componenti periferiche.
Il sistema è completato dagli Adattatori, che formano un layer periferico attorno al Core. Questi componenti implementano le interfacce definite dalle Porte, fungendo da intermediari tra il Core e le risorse esterne. Gli Adattatori si occupano della traduzione dei dati e delle richieste tra il dominio e l'ambiente esterno, garantendo che il Core rimanga isolato dalle specifiche implementative delle interfacce esterne.

3.3.2.1 Componenti

3.3.2.1.1 Core - ConversationService

Nel core del sistema, la classe `ConversationService` si occupa principalmente della gestione delle sessioni, delle conversazioni, dei messaggi e dei feedback.

Gestione delle sessioni

Il servizio offre diversi metodi per la gestione delle sessioni:

- `createSession()` permette di creare una nuova sessione;
- `readSession(sessionId)` recupera una sessione esistente dato un identificativo specifico;
- `updateSession(sessionId)` consente di aggiornare il timestamp di una sessione.

Gestione delle conversazioni

Per quanto riguarda le conversazioni, il servizio gestisce:

- `createConversation(sessionId)` crea una nuova conversazione associata a una sessione esistente;
- `readConversations(sessionId)` restituisce tutte le conversazioni relative a una sessione specifica;
- `readConversationById(conversationId)` permette di recuperare una conversazione in base al suo ID;

- `deleteConversation(conversationId)` consente di eliminare una conversazione (o marcarla come eliminata);
- `updateConversationTimestamp(conversationId)` aggiorna il timestamp di una conversazione.

Gestione dei messaggi

Il sistema offre anche funzionalità per la gestione dei messaggi all'interno delle conversazioni:

- `addMessage(conversationId, sender, content)` consente di aggiungere un nuovo messaggio a una conversazione esistente;
- `readMessages(conversationId)` restituisce tutti i messaggi di una determinata conversazione.

Gestione dei feedback

Infine, la gestione dei feedback si articola in diversi metodi:

- `readFeedback(messageId)` recupera il feedback associato a un messaggio specifico;
- `addFeedback(messageId, feedback, content)` permette di aggiungere un feedback a un messaggio esistente;
- `readNumPositiveFeedback()` e `readNumNegativeFeedback()` forniscono il conteggio dei feedback positivi e negativi;
- `readFeedbackWithComments()` recupera tutti i feedback che contengono anche dei commenti.

3.3.2.1.2 Adapter - DBRepository

Gli adattatori sono responsabili della comunicazione tra il core del sistema e il database. In particolare, `DBRepository` si occupa dell'esecuzione delle query SQL e della gestione dei risultati:

- `executeQuery(query, params)` esegue una query SQL passando i parametri necessari;
- `fetchOne(query, params)` recupera un singolo risultato dalla query;
- `fetchAll(query, params)` restituisce tutti i risultati di una query;
- `close()` chiude la connessione al database.

3.3.2.1.3 Adapter - ContextExtractorService

Il `ContextExtractorService` ha il compito di individuare i prodotti e le porzioni di documentazione (chunk) presenti nel database che risultano più pertinenti e coerenti con il prompt fornito dall'utente.
Il processo di ricerca è stato diviso in 2 macro-operazioni:

1. Ricerca prodotti più coerenti;
2. Ricerca chunk più coerenti, tra quelli che appartengono ai prodotti selezionati.

All'interno del file `ContextExtractorService.py` ci sono delle costanti che possono essere modificate per "scalare" la soluzione (ad esempio, avendo a disposizione più potenza di calcolo):

1. `SIMILARITY_THRESHOLD` è la soglia sotto la quale un prodotto non viene considerato;
2. `TOP_SIMILAR_CHUNKS` è il numero di chunk che vengono inseriti come contesto del prompt;
3. `TOP_PRODUCTS_FINAL` è il numero di prodotti "coerenti" dai quali documenti saranno estratti i `TOP_SIMILAR_CHUNKS`.

`ContextExtractorService` contiene i seguenti metodi:

- `processUserInput(userInput)` è il metodo principale della classe e orchestra l'estrazione dei dati coerenti dal Database.
  `userInput` è il testo del prompt fatto dall'utente;
- `getStructuredProducts()` estrae dal Database le informazioni riguardanti id, titolo, descrizione di ogni prodotto, insieme a 3 vettori che rappresentano rispettivamente l'informazione di "id prodotto", "id e titolo prodotto" e "id, titolo e descrizione prodotto";
- `findTopNSimilarProducts(prodList, promptVector, n)` trova gli `n` prodotti più simili al prompt, utilizzando i vettori sopra citati;
- `aggregateSimilarities(similarProducts)` aggrega le informazioni estratte da `findTopNSimilarProducts`, in questo modo per ogni prodotto si avrà un unico punteggio di similarità;
- `selectChunksEmbeddings(chunks, aggregatedSimilarProduct)` seleziona i chunk dei documenti associati ai prodotti più simili, restituendo i vettori di embedding per il confronto;
- `findTopNSimilar(chunksEmbeddings, userEmbeddings, n)` trova gli `n` chunk più simili rispetto al prompt utente confrontando gli embedding tramite similarità coseno;
- `extractEtim()` estrae dal Database l'informazione relativa agli ETIM (dati tecnici) dei prodotti;
- `selectEtim(etim, textsToEmbed)` associa le informazioni ETIM ai chunk selezionati come contesto, restituendo un dizionario che mappa gli identificativi di prodotto ai relativi codici ETIM;

- `getEmbedding(prompt)` utilizza il modello di embedding `mxbai-embed-large` per generare un vettore di embedding a partire dal testo del prompt;
- `getDocumentsByProductId(productId)` recupera dal Database i documenti associati a un determinato prodotto, includendo il titolo del prodotto e il titolo del documento;
- `loadChunks()` estrae dal Database tutti i chunk di documentazione, inclusi gli embedding associati, per consentire la ricerca di contenuti pertinenti.

3.3.2.1.4 Adapter - LLMResponseService

Il `LLMResponseService` ha il compito di interfacciarsi con un modello di linguaggio per ottenere una risposta contestualizzata basata su una conversazione in corso, una domanda utente e un insieme di informazioni di contesto estratte dal Database. Negli attributi della classe è possibile configurare il modello di linguaggio, il default è `llama3.1:8b`. `LLMResponseService` contiene il seguente metodo:

- `getLlmResponse(conversationPile, question, textsToEmbed, etimToEmbed)` si occupa di formattare i messaggi della conversazione, aggregare il contesto rilevante e inviare la richiesta al modello LLM per ottenere una risposta.

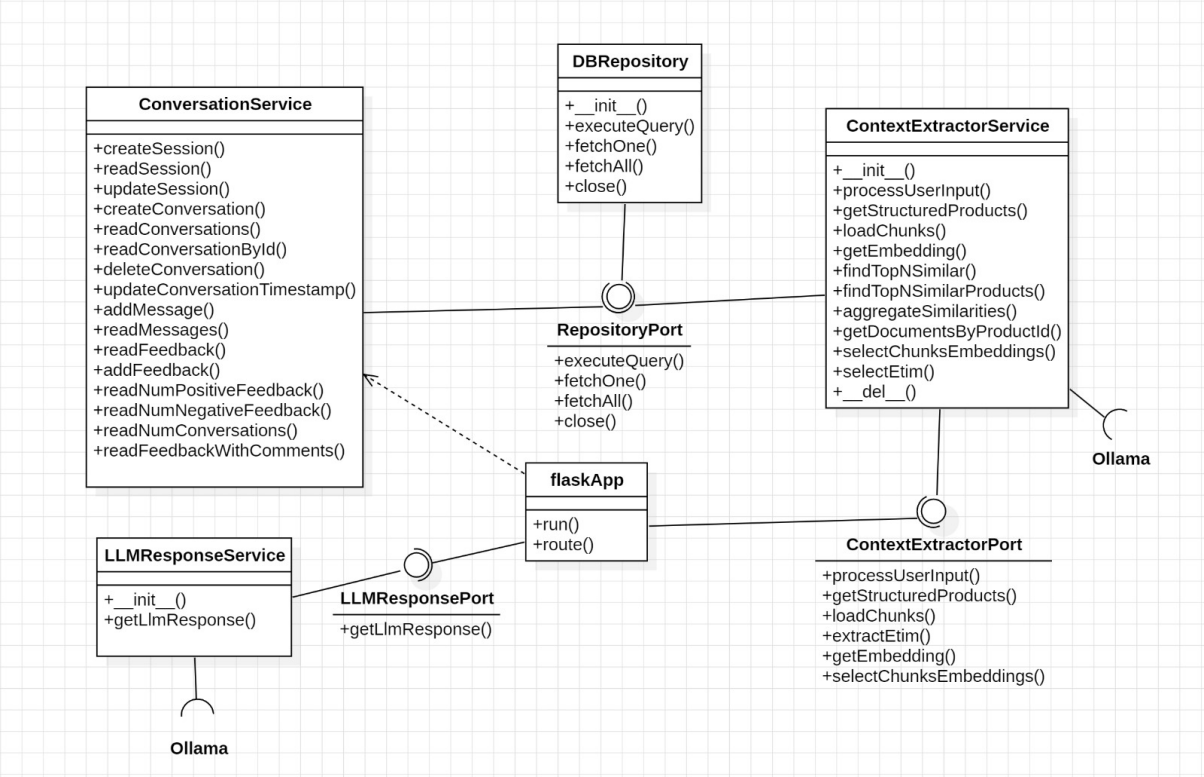


Figura 3: Schema delle classi utilizzate nel backend

3.3.2.1.5 API - APIController

Gli adattatori includono anche `APIController`, che gestisce gli endpoint delle API esposte dal sistema. Esistono vari endpoint per ciascuna delle operazioni descritte, a partire da quelli di test:

- `testApi()` che consente di verificare il funzionamento generale del sistema.

Gestione delle sessioni

- `apiCreateSession()` per creare una nuova sessione;
- `apiUpdateSession(sessionId)` per aggiornare una sessione esistente.

Gestione delle conversazioni

- `apiCreateConversation()` per creare una conversazione;
- `apiReadConversations()` per ottenere tutte le conversazioni di una sessione;
- `apiReadConversationById(conversationId)` per ottenere una conversazione specifica;
- `apiDeleteConversation(conversationId)` per eliminare una conversazione;
- `apiUpdateConversationTimestamp(conversationId)` per aggiornare il timestamp della conversazione.

Gestione dei messaggi

- `apiAddMessage()` per aggiungere un nuovo messaggio;
- `apiReadMessages()` per recuperare i messaggi di una conversazione.

Gestione dei feedback

- `apiReadFeedbackByMessageId(messageId)` recupera il feedback associato a un messaggio;
- `apiAddFeedback()` permette di aggiungere un feedback.

Gestione della dashboard

- `apiReadNumPositiveFeedback()` e `apiReadNumNegativeFeedback()` per ottenere il numero di feedback positivi e negativi;
- `apiReadNumConversations()` per conoscere il numero totale di conversazioni;
- `apiReadFeedbackWithComments()` per ottenere feedback che includono commenti.

Gli endpoint sono protetti da un sistema di sicurezza, con il decoratore `requireAPIKey(f)` che garantisce l'autenticazione tramite chiave API. Inoltre, la configurazione di CORS permette di gestire le richieste provenienti dal frontend.

3.3.3 Database

Figura 4: Diagramma Database

Nel database relazionale sono presenti sette tabelle per la persistenza delle informazioni da utilizzare nella RAG e delle conversazioni degli utenti.
Per quanto riguarda i dati relativi ai prodotti da utilizzare nelle fasi di ricerca del contesto per il modello di linguaggio ci sono tre tabelle di seguito descritte.

- **Product**: immagazzina le informazioni primarie relative ai prodotti disponibili nel sito web
  - id: l'id del prodotto, in formato testuale;
  - title: il titolo o nome del prodotto;
  - description: la breve descrizione del prodotto presente nella pagina web dedicata;
  - etim: i dati tecnici del prodotto presenti nella pagina web dedicata, sottoforma di stringa ("key: value;") a causa della variabilità dei dati disponibili a seconda del tipo di prodotto;
  - idVector: la vettorizzazione dell'id del prodotto per le ricerche;
  - idTitleVector: la vettorizzazione dell'id insieme al titolo del prodotto per le ricerche;
  - idTitleDescrVector: la vettorizzazione dell'insieme di id, titolo e descrizione del prodotto per le ricerche.
- **Document**: immagazzina le informazioni dei documenti estratti dalle pagine web dei prodotti
  - id: id univoco;
  - title: il titolo del documento;
  - productId: l'id del prodotto a cui fa riferimento.

- **Chunk**: immagazzina le informazioni dei chunk creati a partire dai documenti
  - id: id univoco;
  - filename: il titolo del documento da cui è stato estratto il chunk;
  - chunk: il testo del chunk;
  - embedding: la vettorizzazione del chunk per le ricerche.

Per quanto riguarda invece il salvataggio e la permanenza delle conversazioni degli utenti vengono usate quattro tabelle di seguito descritte.

- **Session**: contiene le sessioni, ovvero le connessioni degli utenti
  - sessionId: id univoco che identifica la sessione dell'utente;
  - createdAt: data e ora della creazione della sessione;
  - updatedAt: data e ora dell'ultima modifica alla sessione;
  - isActive: valore booleano che indica se la sessione sia ancora attiva o meno, di default a true, diventa automaticamente false dopo 30 giorni di inattività.
- **Conversation**: contiene le conversazioni degli utenti
  - conversationId: id univoco;
  - sessionId: id della sessione a cui appartiene;
  - createdAt: data e ora della creazione della conversazione;
  - updatedAt: data e ora dell'ultima modifica alla conversazione;
  - toDelete: valore booleano che indica se la conversazione vada eliminata o meno, può diventare true su scelta dell'utente o allo scadere della sessione associata.
- **Message**: contiene tutti i messaggi inviati sia dagli utenti che dal modello LLM
  - messageId: id univoco;
  - conversationId: id della conversazione a cui appartiene;
  - sender: indica se il messaggio è stato inviato dall'utente o è una risposta del modello;
  - content: il contenuto del messaggio stesso;
  - createdAt: data e ora della creazione del messaggio.
- **Feedback**: contiene i feedback lasciati dagli utenti
  - feedbakcId: id univoco;
  - messageId: id del messaggio su cui è stato creato il feedback;
  - isHelpful: booleano, indica un feedback positivo o negativo;
  - content: il contenuto del feedback qualora l'utente desiderasse lasciarlo;
  - createdAt: data e ora della creazione del feedback.

3.3.4 Elaborazione dei dati

Una componente essenziale per il funzionamento del prodotto è la raccolta dei dati dal sito web di Vimar e la loro elaborazione e salvataggio nel database. Queste attività sono ad opera del container `dataProcessing` al cui avvio parte lo scraping del sito web che salva i dati dei prodotti in un file json. Successivamente lo stesso file è il punto di partenza per l'elaborazione dei dati e la generazione dei chunk a partire dai documenti associati ai prodotti, infine vi è il salvataggio nel database. Nel dettaglio:

- `productsElaboration.py` gestisce la manipolazione e l'analisi dei dati relativi ai prodotti, applicando eventuali filtri o trasformazioni;
- `chunkElabotation.py` è particolarmente utile quando si lavora con grandi quantità di dati, poiché divide l'elaborazione in blocchi (chunk) per ottimizzare le prestazioni e ridurre il carico sulla memoria;
- `embeddingLocal.py` è un modulo che si occupa della generazione di embedding locali, utilizzati per analisi semantiche avanzate o per un sistema di raccomandazione basato su similarità tra prodotti;
- `dataSaving.py` è il modulo che si occupa del salvataggio dei dati nel database.

L'esecuzione di questo container, che può richiedere diversi minuti, anche ore per la generazione dei chunk, si conclude dopo il salvataggio dei dati nel database.

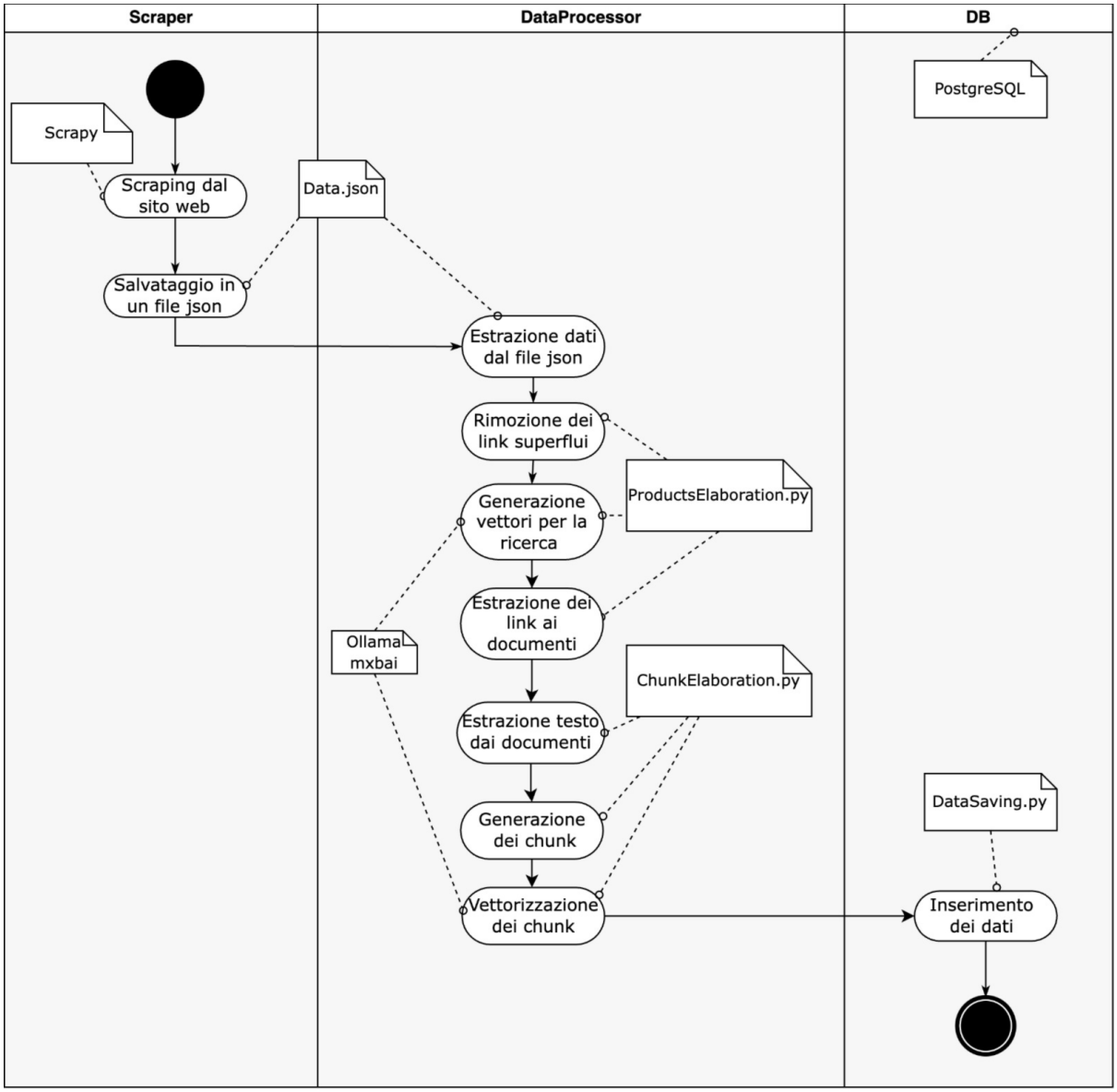


Figura 5: Diagramma delle attività del flusso di Elaborazione dati

3.3.5 Dipendenze e tecnologie utilizzate

L'elenco delle dipendenze è definito nel file requirements.txt, che include una serie di librerie fondamentali per il funzionamento del sistema:

- Flask per la gestione del server web tramite API;
- Ollama per l'elaborazione del linguaggio naturale, utilizzato per l'analisi dei testi associati ai prodotti;
- PostgreSQL come database relazionale, scelto per la sua affidabilità e capacità di gestione di grandi volumi di dati;
- Librerie per la manipolazione e l'analisi dei dati, come numpy e scikit-learn;
- pypdf, che indica la funzionalità legata all'elaborazione di documenti PDF.

4 Tracciamento dei requisiti

Si riportano i requisiti funzionali della tabella presente nel documento Analisi dei Requisiti v2.0.0 - Sez. Req. Funzionali, qui è presente una colonna indicante la soddisfazione di tale requisito.

4.1 Tabella dei requisiti funzionali

ID requisito	Descrizione	Importanza	Stato
RF.O.001	Il *sistema*$_{\text{G}}$ deve permettere all'installatore di effettuare ricerche testuali e ricevere informazioni dettagliate sui prodotti *Vimar*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.O.002	Il *sistema*$_{\text{G}}$ deve prevedere l'autenticazione tramite *password*$_{\text{G}}$ per l'accesso alla *dashboard*$_{\text{G}}$ per amministratori.	Obbligatorio	Soddisfatto
RF.O.003	Il *cruscotto informativo*$_{\text{G}}$ deve includere una sezione per la visualizzazione di statistiche di utilizzo.	Obbligatorio	Soddisfatto
RF.O.004	Il *sistema*$_{\text{G}}$ deve permettere agli utenti di fornire un *feedback*$_{\text{G}}$ positivo o negativo dopo ogni risposta ricevuta.	Obbligatorio	Soddisfatto
RF.O.005	Il *sistema*$_{\text{G}}$ deve essere in grado di identificare e bloccare le richieste che riguardano argomenti non pertinenti ai prodotti *Vimar*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.D.006	Il *sistema*$_{\text{G}}$ deve includere la possibilità di visualizzare *link*$_{\text{G}}$ di riferimento alle fonti delle informazioni fornite.	Desiderabile	Non soddisfatto
RF.P.007	Il *sistema*$_{\text{G}}$ deve fornire un'interfaccia con menu e sotto-menu per costruire richieste specifiche in conversazioni guidate.	Opzionale	Non soddisfatto
RF.D.008	Il componente di interrogazione deve prevedere un controllo sull'*output*$_{\text{G}}$ per verificare che il contenuto non vada in conflitto con argomenti proibiti.	Desiderabile	Soddisfatto

Tabella 2: Requisiti di funzionalità (1$^{\text{a}}$ parte)

RF.O.009	L'utente deve poter fare richieste testuali limitate a un certo numero di caratteri.	Obbligatorio	Soddisfatto
RF.O.010	L'utente deve poter visualizzare uno storico dei messaggi nella stessa conversazione.	Obbligatorio	Soddisfatto
RF.D.011	L'utente può fare richieste in italiano e/o in inglese.	Desiderabile	Soddisfatto
RF.O.012	Le conversazioni avute possono essere salvate al termine della *sessione*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.O.013	Le conversazioni devono poter essere cancellate.	Obbligatorio	Soddisfatto
RF.O.014	Ogni utente dovrà avere un limite massimo di conversazioni.	Obbligatorio	Soddisfatto
RF.P.015	L'*amministratore*$_{\text{G}}$ deve poter visualizzare nel *cruscotto informativo*$_{\text{G}}$ il numero totale di richieste effettuate con *conversazione libera*$_{\text{G}}$ o *guidata*$_{\text{G}}$.	Opzionale	Non soddisfatto
RF.P.016	L'*amministratore*$_{\text{G}}$ deve poter visualizzare nel *cruscotto informativo*$_{\text{G}}$ la classifica sul numero di termini usati maggiormente nelle richieste.	Opzionale	Non soddisfatto
RF.D.017	L'*amministratore*$_{\text{G}}$ deve poter visualizzare nel *cruscotto informativo*$_{\text{G}}$ il numero di risposte positive o negative ricevute dal sistema di *feedback*$_{\text{G}}$.	Desiderabile	Soddisfatto
RF.D.018	L'utente deve poter scaricare i file di istruzioni dei prodotti in formato PDF.	Desiderabile	Non soddisfatto
RF.O.019	Il *sistema*$_{\text{G}}$ deve essere in grado di ricavare le informazioni utili dai PDF ed estrarre le immagini degli schemi elettrici.	Desiderabile	Non soddisfatto
RF.O.020	Il *sistema*$_{\text{G}}$ deve impiegare un *database*$_{\text{G}}$ per collezionare le informazioni relative ai prodotti.	Obbligatorio	Soddisfatto

Tabella 3: Requisiti di funzionalità (1ª parte)

RF.O.021	Il componente di interrogazione deve fornire in $output_{\mathrm{G}}$ la risposta del modello AI_{G} (LLM_{G}).	Obbligatorio	Soddisfatto
RF.O.022	Il modello AI_{G} (LLM_{G}) deve essere in grado di rispondere a domande sui prodotti appartenenti agli impianti Smart.	Obbligatorio	Soddisfatto
RF.O.023	Il modello AI_{G} (LLM_{G}) deve essere in grado di rispondere a domande sui prodotti appartenenti agli impianti Domotici.	Obbligatorio	Soddisfatto
RF.D.024	Il modello AI_{G} (LLM_{G}) deve essere in grado di rispondere a domande sui prodotti appartenenti agli impianti Tradizionali.	Desiderabile	Non soddisfatto
RF.D.025	L'utente deve poter visualizzare il numero della pagina del documento relativo alla risposta.	Desiderabile	Non soddisfatto
RF.D.026	L'utente deve poter visualizzare il nome del documento relativo alla risposta.	Desiderabile	Non soddisfatto
RF.O.027	L'utente deve poter confermare l'eliminazione di una conversazione con il $chatbot_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RF.O.028	L'utente deve poter visualizzare una lista delle $sessioni_{\mathrm{G}}$ di conversazione attive col $chatbot_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RF.O.029	L'utente deve poter creare una nuova $sessione_{\mathrm{G}}$ di conversazione col $chatbot_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RF.O.030	L'utente deve ricevere una risposta di cortesia quando pone una domanda relativa a dei contenuti proibiti.	Obbligatorio	Soddisfatto
RF.O.031	L'utente deve essere notificato da un errore se non ci sono informazioni sul prodotto ricercato.	Obbligatorio	Soddisfatto
RF.O.032	L'utente deve poter digitare la domanda da porgere al $chatbot_{\mathrm{G}}$ tramite tastiera.	Obbligatorio	Soddisfatto

Tabella 4: Requisiti di funzionalità (1ª parte)

RF.O.033	L'utente deve poter inviare le domande da porgere al *chatbot*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.P.034	L'utente deve poter visualizzare una conversazione salvata in precedenza.	Opzionale	Soddisfatto
RF.P.035	L'utente può visualizzare la data di invio di un messaggio.	Opzionale	Non soddisfatto
RF.P.036	L'utente può visualizzare l'ora di invio di un messaggio.	Opzionale	Non soddisfatto
RF.P.037	L'utente deve poter consultare il contenuto del documento di interesse.	Opzionale	Non soddisfatto
RF.D.038	L'interfaccia utente del *sistema*$_{\text{G}}$ deve essere responsive, adattandosi a diversi dispositivi.	Desiderabile	Soddisfatto
RF.O.039	L'utente deve poter creare una *conversazione libera*$_{\text{G}}$ o *guidata*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.O.040	L'utente deve essere notificato da un errore quando supera il numero limite di conversazioni.	Obbligatorio	Soddisfatto
RF.O.041	L'installatore deve poter visualizzare la singola *conversazione libera*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.O.042	L'installatore deve poter visualizzare la singola *conversazione guidata*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.P.043	L'installatore deve essere in grado di modellare il *prompt*$_{\text{G}}$ tramite una serie di menù nella *conversazione guidata*$_{\text{G}}$.	Opzionale	Non soddisfatto
RF.P.044	L'installatore deve visualizzare un suggerimento sulla domanda successiva nella *conversazione libera*$_{\text{G}}$.	Opzionale	Non soddisfatto
RF.O.045	L'installatore deve poter visualizzare l'area per scrivere il messaggio da inviare al *sistema*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RF.O.046	L'installatore deve visualizzare data e ora sia dei messaggi inviati sia dei messaggi ricevuti.	Obbligatorio	Soddisfatto
RF.O.047	L'utente deve poter salvare in formato PDF una conversazione.	Obbligatorio	Soddisfatto

Tabella 5: Requisiti di funzionalità (1ª parte)

RF.O.048	L'amministratore deve essere notificato da un errore se le credenziali inserite sono errate.	Obbligatorio	Soddisfatto
RF.O.049	L'installatore deve essere notificato dell'assenza di alcuni elementi della conversazione, non presenti a causa della mancanza di messaggi pregressi nella conversazione.	Obbligatorio	Soddisfatto
RF.O.050	L'installatore deve essere notificato dell'impossibilità di salvare una conversazione vuota.	Obbligatorio	Soddisfatto
RF.O.051	L'installatore deve poter identificare le conversazioni presenti nel $sistema_{\mathrm{G}}$ tramite un nome.	Obbligatorio	Soddisfatto
RF.D.052	L'installatore deve poter rinominare le conversazioni presenti nel $sistema_{\mathrm{G}}$.	Desiderabile	Non soddisfatto

Tabella 6: Requisiti di funzionalità (5ª parte)

4.2 Tabella dei requisiti Vincolo

ID requisito	Descrizione	Importanza	Stato
RV.O.001	Il *sistema*$_{\mathrm{G}}$ deve integrare il modello AI_{G} (LLM_{G}) *Open Source*$_{\mathrm{G}}$ Llama 3.1 8B.	Obbligatorio	Soddisfatto
RV.O.002	L'infrastruttura *Cloud*$_{\mathrm{G}}$ deve utilizzare *Docker*$_{\mathrm{G}}$ insieme a *Docker*$_{\mathrm{G}}$ Compose, al fine di rispettare il principio di Infrastructure as Code.	Obbligatorio	Soddisfatto
RV.D.003	L'applicativo può essere ospitato su AWS_{G}.	Desiderabile	Soddisfatto
RV.O.004	Il modello AI_{G} (LLM_{G}) deve essere *Open Source*$_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RV.O.005	Il componente di interrogazione deve essere in grado di interfacciarsi con il *sistema*$_{\mathrm{G}}$ di indicizzazione e con il modello AI_{G} (LLM_{G}).	Obbligatorio	Soddisfatto
RV.O.006	Il componente di interrogazione deve poter essere contattato da un altro servizio sotto-forma di API_{G} autenticata (ad esempio tramite API_{G}-KEY)	Obbligatorio	Soddisfatto
RV.O.007	L'infrastruttura deve utilizzare la tecnologia dei *container*$_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RV.O.008	Il risultato atteso è che la parte applicativa possa essere costruita e replicata con un solo comando.	Obbligatorio	Soddisfatto
RV.O.009	Il *repository*$_{\mathrm{G}}$ di lavoro deve essere versionato tramite Git e deve essere pubblicamente accessibile.	Obbligatorio	Soddisfatto
RV.O.010	La licenza per i sorgenti dovrà essere *Open Source*$_{\mathrm{G}}$.	Obbligatorio	Soddisfatto
RV.O.011	Il modello AI_{G} (LLM_{G}) dovrà fare uso dell'approccio RAG_{G}.	Obbligatorio	Soddisfatto
RV.O.012	L'applicazione deve essere compatibile con il browser Chrome dalla versione 108.	Obbligatorio	Soddisfatto
RV.O.013	L'applicazione deve essere compatibile con il browser Edge dalla versione 94.0.992.31.	Obbligatorio	Soddisfatto
RV.O.014	L'applicazione deve essere compatibile con il browser Opera dalla versione 95.	Obbligatorio	Soddisfatto

Tabella 7: Requisiti di vincolo (1ª parte)

RV.O.015	L'applicazione deve essere compatibile con il browser Firefox dalla versione 109.	Obbligatorio	Soddisfatto
RV.O.016	L'applicazione deve essere compatibile con il browser Safari dalla versione 16.	Obbligatorio	Soddisfatto
RV.O.017	L'applicativo deve prevedere un *sistema*$_{\text{G}}$ di estrazione e raccolta delle informazioni dal sito web dell'azienda.	Obbligatorio	Soddisfatto
RV.O.018	Il *sistema*$_{\text{G}}$ deve essere in grado di navigare un elenco di prodotti, estrarre le informazioni utili e immagazzinare le informazioni correlandole in modo opportuno.	Obbligatorio	Soddisfatto
RV.O.019	Il *sistema*$_{\text{G}}$ di estrazione e raccolta deve essere realizzato sotto-forma di *pipeline*$_{\text{G}}$ e automatizzato.	Obbligatorio	Soddisfatto
RV.O.020	L'applicativo deve prevedere un *sistema*$_{\text{G}}$ di indicizzazione delle informazioni a partire dal *database*$_{\text{G}}$ in cui sono stati salvati i dati precedentemente estratti dal sito web.	Obbligatorio	Soddisfatto

Tabella 8: Requisiti di vincolo (2$^{\text{a}}$ parte)

4.3 Tabella dei requisiti Qualità

ID requisito	Descrizione	Importanza	Stato
RQ.O.001	È necessario fornire un documento che descriva le attività di *bug*$_{\text{G}}$ reporting effettuate.	Obbligatorio	Soddisfatto
RQ.O.002	Il progetto deve essere svolto seguendo le regole contenute nel documento Norme di Progetto.	Obbligatorio	Soddisfatto
RQ.O.003	È necessario fornire al proponente il codice sorgente dell'applicativo in un *repository*$_{\text{G}}$ *GitHub*$_{\text{G}}$.	Obbligatorio	Soddisfatto
RQ.O.004	È necessario fornire il Manuale Utente dell'applicativo.	Obbligatorio	Soddisfatto

Tabella 9: Requisiti di qualità

4.4 Grafici requisiti soddisfatti

Riguardo alla soddisfazione dei requisiti il gruppo PEBKAC ha soddisfatto 61 su 76, arrivando ad una copertura del 80%

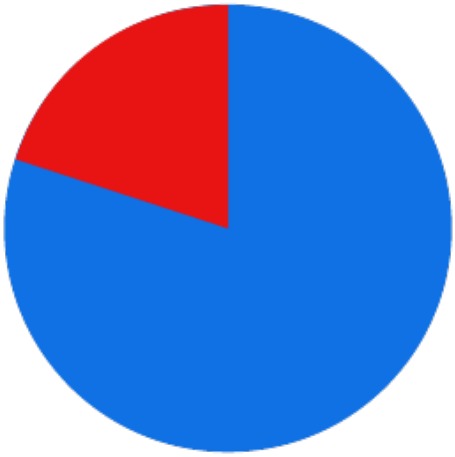


Figura 6: Grafico requisiti

Per quanto riguarda la copertura dei requisiti obbligatori, la copertura rilevata è di 55 su 55 requisiti, arrivando quindi ad un 100% sul totale

Figura 7: Grafico requisiti obbligatori

Per quanto riguarda la copertura dei requisiti desiderabili, la copertura rilevata è di 5 su 12 requisiti, arrivando quindi ad un 41% sul totale

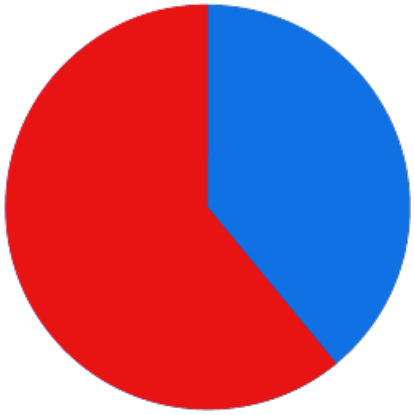


Figura 8: Grafico requisiti desiderabili

Per quanto riguarda la copertura dei requisiti opzionali, la copertura rilevata è di 1 su 9 requisiti, arrivando quindi ad un 11% sul totale

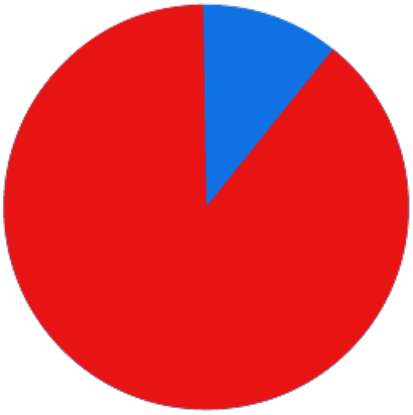


Figura 9: Grafico requisiti opzionali