SWEvenTeam

E-Mail
sweventeam@outlook.it

Specifica Tecnica

Capitolato 1 ArtificialQI

Informazioni documento

Versione	1.0.0
Redazione	Alessio Barraco Alessandro Damiani Yuri Lunardon Matteo Mazzotti Valentina Schivo Alessio Turetta
Verifica	Alessio Barraco Alessandro Damiani Alba Hui Larrosa Serrano Yuri Lunardon Matteo Mazzotti Valentina Schivo Alessio Turetta
Approvazione	Valentina Schivo

Storia del documento

Versione	Data	Autori	Verificatori	Descrizione
1.0.0	2025-06-09	Valentina Schivo	-	Approvazione documento
0.7.0	2025-06-06	Alessio Barraco, Alessandro Damiani	Alessio Turetta, Matteo Mazzotti	Controllo finale
0.6.0	2025-05-26	Yuri Lunardon, Alessandro Damiani	Valentina Schivo	Aggiornamento e miglioramento contenuti
0.5.0	2025-05-16	Matteo Mazzotti, Alessio Turetta	Alessio Barraco	Redazione frontend
0.4.0	2025-05-13	Yuri Lunardon, Valentina Schivo	Alessandro Damiani	Architettura deployment
0.3.0	2025-05-10	Matteo Mazzotti, Alessio Turetta, Alessio Barraco	Alba Hui Larrosa Serrano, Yuri Lunardon	Descrizione database, backend
0.2.0	2025-05-05	Yuri Lunardon, Alessandro Damiani	Valentina Schivo	Logica del prodotto, Architettura di sistema
0.1.0	2025-04-28	Valentina Schivo	Alessio Barraco	Introduzione, Descrizione prodotto, Tecnologie
0.0.1	2025-04-20	Alessio Barraco	Valentina Schivo	Definizione struttura

Indice

1 Introduzione

1.1 Scopo del documento

Il documento ha come obiettivo quello di illustrare e motivare le scelte architetturali e di design adottate nello sviluppo della web-app "ArtificialQI".
Il documento descrive l'architettura$_G$ logica e di deployment$_G$, i design pattern e le tecnologie utilizzate, corredando la trattazione con diagrammi di classe per chiarire la struttura del software.

1.2 Glossario

Per garantire chiarezza e precisione nella comunicazione, è stato introdotto un elenco di riferimento, denominato Glossario, che raccoglie e spiega i termini tecnici o specifici utilizzati nella documentazione. Questo strumento, concepito per prevenire fraintendimenti o dubbi legati al linguaggio impiegato, include una serie di voci con le relative spiegazioni, permettendo così di definire in modo univoco i concetti e i termini, garantendo l'uniformità del lessico adoperato. La presenza di un termine all'interno del Glossario viene indicata con questo stile$_G$.

1.3 Riferimenti

1.3.1 Normativi

- **Norme di progetto v2.0.0**
  https://sweventeam17.github.io/pdf/2-PB/Documenti%20interni/Norme_di_Progetto_v2.0.0.pdf
  Ultima consultazione: 2025-06-04;
- **Capitolato d'appalto C1: ArtificialQI**
  https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C1.pdf
  Ultima consultazione: 2025-06-04;
- **Regolamento di progetto**
  https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf
  Ultima consultazione: 2025-06-04.

1.3.2 Informativi

- **Analisi dei Requisiti v2.0.0**
  https://sweventeam17.github.io/pdf/2-PB/Documenti%20esterni/Analisi_dei_Requisiti_v2.0.0.pdf
  Ultima consultazione: 2025-06-04;
- **Piano di Qualifica v2.0.0**
  https://sweventeam17.github.io/pdf/2-PB/Documenti%20esterni/Piano_di_Qualifica_v2.0.0.pdf
  Ultima consultazione: 2025-06-04;
- **Pattern architetturali**
  https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
  Ultima consultazione: 2025-06-04;
- **Qualità del software**
  https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/T7.pdf
  Ultima consultazione: 2025-06-04;
- **Langchain**
  https://js.langchain.com/docs/get_started/introduction
  Ultima consultazione: 2025-06-04;

- **Next.js**
  https://nextjs.org/docs
  Ultima consultazione: 2025-06-04;
- **React**
  https://it.legacy.reactjs.org
  Ultima consultazione: 2025-06-04;
- **Django Rest Framework**
  https://www.django-rest-framework.org/
  Ultima consultazione: 2025-06-04;
- **Ollama**
  https://ollama.com/docs
  Ultima consultazione: 2025-06-04;
- **Gemini**
  https://ai.google.dev/gemini/
  Ultima consultazione: 2025-06-04;
- **Sentence-Transformers**
  https://www.sbert.net/
  Ultima consultazione: 2025-06-04;
- **Docker**
  https://docs.docker.com/
  Ultima consultazione: 2025-06-04;
- **MySQL**
  https://www.mysql.com/
  Ultima consultazione: 2025-06-04;
- **Pytest**
  https://docs.pytest.org/en/7.4.x/
  Ultima consultazione: 2025-06-04;
- **Cypress**
  https://docs.cypress.io/
  Ultima consultazione: 2025-06-04;
- **Pylint**
  https://pylint.pycqa.org/en/latest/
  Ultima consultazione: 2025-06-04;
- **ESLint**
  https://eslint.org/
  Ultima consultazione: 2025-06-04;
- **Prettier**
  https://prettier.io/
  Ultima consultazione: 2025-06-04;
- **Black**
  https://black.readthedocs.io/en/stable/
  Ultima consultazione: 2025-06-04;
- **Unittest**
  https://docs.python.org/3/library/unittest.html
  Ultima consultazione: 2025-06-04;
- **Recharts**
  https://recharts.org/en-US
  Ultima consultazione: 2025-06-04.

2 Descrizione del prodotto

2.1 Scopo

ArtificialQI è una web-app con lo scopo principale di valutare in modo sistematico le capacità di risposta$_G$ dei Large Language Models (LLM$_G$). Serve a fornire agli sviluppatori uno strumento avanzato e affidabile per il testing, l'analisi e il confronto delle prestazioni dei diversi LLM. L'obiettivo è superare le limitazioni degli attuali metodi di validazione, come benchmark generici o valutazioni manuali, che sono lunghi e difficili da integrare nei cicli di sviluppo. ArtificialQI consente di archiviare liste di domande e risposte attese, eseguire test$_G$ automatizzati ponendo le domande a un LLM esterno e registrandone la risposta, valutare la correttezza o la verosimiglianza delle risposte ricevute, e presentare i risultati, il tutto integrato in un unico sistema$_G$. Affronta la complessità della valutazione dei modelli che non operano in modo deterministico e permette di verificare e confrontare come le prestazioni degli LLM variano in base alle loro caratteristiche, come il numero di parametri e l'addestramento. Il cuore del problema che l'applicazione risolve è la valutazione della verosimiglianza delle risposte ottenute dagli LLM.

2.2 Funzionalità

ArtificialQI offre le seguenti funzionalità$_G$ integrate in un unico ambiente:

- Gestione modelli: importazione di modelli arbitrari in base alle risorse disponibili e gestione centralizzata;
- Test automatizzati: invio delle domande archiviate tramite API$_G$ e raccolta delle risposte;
- Valutazione: calcolo di similarità semantica e di correttezza tramite un LLM di valutazione$_G$;
- Risultati: punteggi e grafici sintetici per una consultazione rapida;
- Storico e confronto: archivio delle esecuzioni e confronto diretto tra i modelli.

3 Tecnologie

3.1 Codifica

3.1.1 Linguaggi

- **Python**: Linguaggio principale per lo sviluppo del backend$_G$, grazie alla sua versatilità, al vasto ecosistema di librerie e all'integrazione con strumenti di machine learning;
- **JavaScript**: Utilizzato per lo sviluppo frontend$_G$ con Next.js$_G$, consente l'aggiornamento dell'interfaccia in tempo reale;
- **SQL**: Linguaggio per l'interazione con il database$_G$ MySQL$_G$, sfruttato per la gestione e il salvataggio dei dati strutturati.

3.1.2 Strumenti e servizi

- **Ollama**: Piattaforma$_G$ per l'esecuzione locale di modelli linguistici (LLM), che permette di utilizzare modelli pre-addestrati senza dipendere da servizi cloud. Ideale per sperimentazioni e prototipazione rapida;
- **Docker**: Piattaforma per containerizzazione delle applicazioni, che garantisce isolamento, portabilità e facilità di deployment in ambienti diversi;
- **MySQL**: Sistema di gestione di database relazionale open-source;
- **Gemini**: Famiglia di modelli di IA sviluppati da Google DeepMind, utilizzati per funzionalità avanzate di elaborazione del linguaggio naturale (NLP).

3.1.3 Framework

- **Next.js**: Framework$_G$ React$_G$-based per lo sviluppo frontend, con supporto a rendering ibrido (SSR, SSG) e routing integrato. Ottimizzato per performance e SEO;
- **Django Rest Framework (DRF)**: Estensione di Django$_G$ per la creazione di API RESTful, scelto per la sua scalabilità, sicurezza e strumenti built-in (es. autenticazione, serializzazione).

3.1.4 Librerie

- **LangChain**: Libreria$_G$ per l'integrazione di LLM in applicazioni software, facilitando operazioni come la gestione di prompt, connessione a dati esterni e orchestrazione di flussi complessi;
- **Sentence-Transformers**: Libreria Python contenente strumenti utili alla conversione di stringhe in vettori di embedding$_G$ e al calcolo della similarità semantica;
- **React**: Libreria frontend per la costruzione di interfacce utente dinamiche e componenti riutilizzabili.

3.2 Analisi

3.2.1 Statica

- **ESLint**: Strumento per l'identificazione di errori sintattici, problemi di stile e potenziali bug$_G$ in codice JavaScript/TypeScript. Configurabile con regole personalizzate;
- **Pylint**: Analizzatore per Python che verifica la qualità$_G$ del codice, enforce di convenzioni (PEP 8) e rilevamento di code smell;
- **Prettier**: Formattatore automatico per JavaScript/TypeScript, HTML e CSS, per garantire consistenza nello stile del codice;
- **Black**: Formattatore di codice per Python, applica uno stile uniforme senza opzioni configurabili.

3.3 Testing

3.3.1 Linguaggi

- **Python**: Utilizzato per test di unità$_G$ e integrazione nel backend (con librerie come Pytest$_G$);
- **JavaScript**: Impiegato per test frontend (con Cypress$_G$).

3.3.2 Framework

- **Pytest**: Framework di testing per Python, utilizzato per testare le funzioni implementate nel backend;
- **Cypress**: Framework di testing end-to-end per testare il comportamento del frontend simulando le azioni dell'utente in un browser reale.

3.3.3 Librerie

- **Unittest**: Libreria standard Python per test, usata in combinazione con pytest.

4 Architettura

4.1 Logica del prodotto

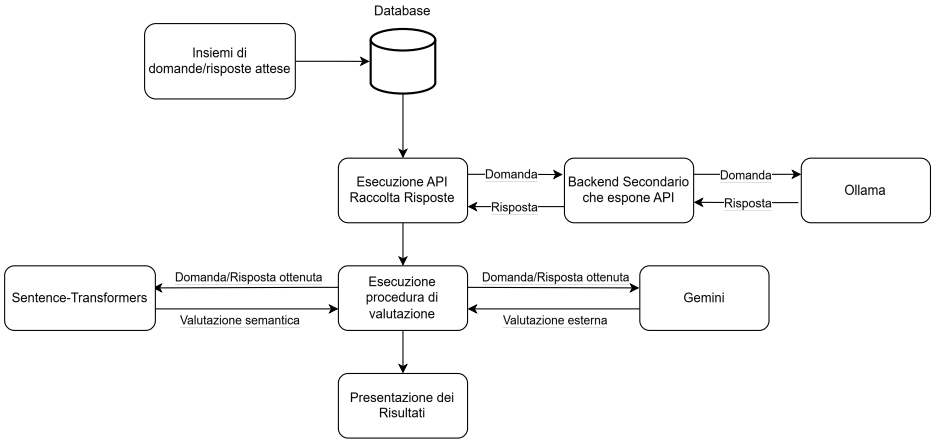

Figura 1: Funzionamento del prodotto

Il sistema utilizza degli insiemi di domande/risposte attese per valutare i Large Language Models installati all'interno di Ollama$_G$. L'utente può creare delle sessioni, ovvero degli ambienti isolati in cui è possibile, tramite interfaccia grafica, selezionare quali LLM valutare e quali insiemi di domande/risposte utilizzare per il test. Il sistema procede alla valutazione ogni volta che il sistema ottiene una risposta ad una domanda presente all'interno del blocco. Per fare ciò vengono utilizzati due metodi:

- **Valutazione semantica**;
- **Valutazione esterna**.

4.1.1 Valutazione semantica

La valutazione semantica viene fornita dalla libreria Sentence-Transformers, la quale utilizza modelli di deep learning in grado di convertire testi in vettori numerici (embedding) e posizionarli in uno spazio vettoriale. Viene poi calcolato, tramite coseno di similitudine, il grado di vicinanza tra il vettore della risposta generata dal modello e quello della risposta attesa. Il valore risultante (compreso tra 0 e 1) rappresenta il livello di similarità semantica tra le due frasi: valori prossimi a 1 indicano risposte molto simili (quindi potenzialmente corrette o accettabili), mentre valori più bassi indicano risposte potenzialmente poco pertinenti o errate.

4.1.2 Valutazione esterna

La valutazione esterna viene fornita dal modello multimodale Gemini$_G$ 2.0 Flash. Il sistema, tramite un prompt specifico, chiede al modello di valutare la similarità tra la risposta attesa e la risposta data dal LLM e di ritornare un valore compreso tra 0 e 100, dove 0 significa che le due risposte sono completamente differenti mentre 100 significa che le due risposte sono identiche.

4.2 Architettura del sistema

Questa sezione descrive l'architettura scelta per il sistema, delineando i componenti principali, il modello architetturale adottato e i vantaggi che ne derivano.

4.2.1 Modello architetturale

Per la realizzazione dell'applicativo è stato scelto un modello architetturale ibrido basato su quattro componenti disaccoppiati:

- **Frontend**: si occupa di definire e fornire l'interfaccia grafica tramite la quale l'utente interagisce con l'applicativo. Inoltra tutte le richieste al backend primario e mostra i risultati ottenuti;
- **Backend primario monolitico**: strutturato secondo l'architettura a layer$_G$, si occupa di gestire le richieste dell'utente e si interfaccia con il database e con il microservizio per modelli LLM;
- **Microservizio per modelli LLM**: microservizio strutturato secondo l'architettura a layer che si occupa dell'integrazione con il server Ollama per interrogare i modelli installati e ottenere le risposte;
- **Database**: memorizza i dati relativi alle sessioni, ai modelli installati, alle domande, alle risposte ottenute e alle valutazioni.

I componenti comunicano tra di loro utilizzando `REST API` e inviando e ricevendo dati in formato strutturato JSON$_G$.

4.2.2 Suddivisione a livelli del backend

Il backend, nella sua interezza, è suddiviso in diversi livelli logici, ciascuno con responsabilità ben definite:

- **Livello Applicazione**: gestisce il flusso delle richieste provenienti dal frontend, orchestrando i casi d'uso e coordinando l'interazione tra gli altri layer;
- **Livello Dominio**: contiene la logica di business del sistema, ovvero le regole applicative centrali e le operazioni sui dati;
- **Livello Persistenza**: si occupa dell'accesso ai dati, fornendo un'interfaccia per interagire con il sistema di persistenza sottostante.

Il **livello di presentazione** non è gestito dal backend, ma è interamente delegato al frontend, che si occupa della definizione e della visualizzazione dell'interfaccia grafica.

4.2.3 Vantaggi dell'architettura scelta

L'adozione di un'architettura ibrida ha portato a numerosi vantaggi fra cui:

- Maggiore semplicità di sviluppo del backend principale dovuta alla semplicità dell'architettura a layer e alla separazione netta delle responsabilità di ogni layer;
- Maggiore semplicità nello sviluppo dei test in quanto ogni strato svolge un ruolo specifico e può essere facilmente sostituito tramite mocking durante la scrittura dei test;
- Ottima leggibilità del codice grazie alla separazione netta tra viste e servizi il codice risulta leggibile, chiaro e manutenibile;
- Buona predisposizione al cambiamento della base di dati grazie all'utilizzo di un livello persistenza dedicato;
- Maggior scalabilità del sistema rispetto ad un'architettura puramente monolitica in quanto l'operazione computazionalmente più esosa (ovvero l'interrogazione dei modelli) è derogata ad un microservizio separato che può essere scalato in maniera indipendente dal resto del sistema.

4.2.4 Svantaggi dell'architettura a layer

L'adozione di un'architettura a layer comporta alcuni svantaggi, fra cui:

- Una minor predisposizione ad espansioni significative del codice del sistema, a causa della rigidità del pattern scelto;
- Un sacrificio maggiore delle performance in quanto ogni richiesta attraversa più livelli logici prima di essere processata;

- Una scalabilità inferiore rispetto ad architetture più flessibili, come quelle a microservizi distribuiti.

Tuttavia, considerando che il sistema è destinato a un utilizzo interno a fini di sviluppo e testing, tali limitazioni risultano trascurabili rispetto ai vantaggi in termini di manutenibilità e semplicità strutturale.

4.3 Architettura del frontend

Il frontend è stato sviluppato utilizzando il framework Next.js, il quale permette la creazione di pagine composte da componenti logicamente separati e riutilizzabili. Ciascun componente può definire una propria logica e può utilizzare a sua volta altri componenti per la realizzazione delle pagine viste dall'utente. Per separare la logica dalla presentazione, ove necessario, sono stati utilizzati:

- **Custom Hook**: funzioni che incapsulano logica locale, mantenendola separata dalla presentazione del componente. Non prevedono condivisioni di stato o comportamenti con altri componenti, ma forniscono un'istanza isolata a ogni invocazione;
- **Context Provider**: pattern che consente di gestire stato e logica condivisi tra più componenti tramite la creazione di un componente Provider a monte nella gerarchia, che sfrutta le Context API di React per esporre dati e funzioni ai componenti discendenti, permettendo di avere una singola fonte di verità ed eliminando la necessità di passare proprietà manualmente lungo l'intera struttura.

Questa architettura viene applicata a ciascuna delle quattro macro-funzionalità della piattaforma:

- Gestione dei modelli LLM;
- Gestione degli insiemi di domande;
- Esecuzione di test;
- Confronto dei risultati.

I due vantaggi principali nell'utilizzo di questo approccio sono:

- **Manutenibilità**: separare la logica dalla presentazione permette di intervenire su una parte del sistema senza rischiare effetti collaterali altrove. Le modifiche restano circoscritte e il codice è più leggibile.
- **Scalabilità**: l'aggiunta di nuove funzionalità avviene per estensione, non per riscrittura: si possono riusare pattern esistenti e introdurre nuovi comportamenti senza introdurre dipendenze né ripetere codice.

4.3.1 Context e Hook

4.3.1.1 SessionContext:

Questo provider centralizza tutte le operazioni sull'elenco delle sessioni: mantiene lo stato sessions (array di sessioni recuperate dal backend) e fornisce tre metodi principali: fetchSessions() per caricare tutte le sessioni, updateSession(id, data) per aggiornare titolo/descrizione di una singola sessione e deleteSession(id) per rimuovere una sessione.
Garantisce una singola fonte di dati, evitando di dover passare state e callback tramite prop-drilling e assicurando coerenza in tutta l'app.

4.3.1.2 SessionCardContext:

Pensato per la gestione delle card di sessione in modalità di modifica.
Mantiene lo stato locale di UI: isEditing, editedTitle e editedDescription, ed espone i metodi startEditing() e cancelEditing() per attivare o annullare la modalità di modifica, saveChanges() per confermare le modifiche e removeSession() per eliminare la sessione.
Isola la logica e lo stato di interazione (UI) relativi a una singola card dal contesto globale delle sessioni, semplificando i componenti di presentazione.

4.3.1.3 SessionLLMContext:

Gestisce i modelli LLM associati a una sessione: ha in sessionData i dettagli completi (inclusi i LLM già caricati), in remainingLLMs quelli ancora selezionabili e in limit/isLLMDataEmpty eventuali messaggi di errore o di lista vuota.
Usa fetchSessionData() e fetchRemainingLLMs() per sincronizzare i dati, submitLLM() per aggiungere un nuovo modello (massimo 3), e deleteLLM() per rimuoverne uno già assegnato.
Concentra in un unico provider tutta la logica CRUD sui modelli LLM di una sessione.

4.3.1.4 TestFormContext:

Orchestra il flusso di creazione ed esecuzione di un test: tiene in selectedBlocks gli insiemi di domande scelti, in questionBlocks quelli disponibili.
I metodi principali sono fetchQuestionBlocks() per caricare gli insiemi, addBlock(id)/removeBlock(id) per selezionare o togliere un blocco, submitToBackend() per invocare il test, showPrevTests() e handlePrevTestClick() per lo storico, più handleJSONFileChange() e handleJSONSubmit() per la gestione dei test eseguiti importando le domande da file JSON.
Riunisce tutte le operazioni e lo stato necessari al form di test.

4.3.1.5 TestComparatorContext:

Questo provider gestisce la logica di confronto tra due LLM all'interno di una sessione. Mantiene lo stato selectedSessionData (dati completi della sessione scelta, inclusi i modelli associati), selectedLLMS (ID del primo e del secondo LLM selezionati), llmNames (mappa ID-nome per etichettare i grafici) e blockComparisonData (dati di confronto sui blocchi comuni).
Espone fetchSessionData(id) per caricare i dettagli di una sessione e popolare anche llmNames, fetchBlockComparisonData(first, second) per recuperare i confronti semantici ed esterni sugli insiemi, più setSelectedLLMS e il calcolo in chartData per formattare i dati pronti per i grafici.
Contiene la preparazione dei dati di comparazione tra due LLM, facilitando i componenti di visualizzazione (grafici, tabelle).

4.3.1.6 InspectBlockContext:

Si occupa di caricare e gestire i dettagli di un singolo insieme di domande, permettendo la visualizzazione e modifica dei prompt.
Tiene in blockData i dati del blocco (inclusi i prompt), testResults (eventuali esecuzioni salvate) e uniqueId (ID del prompt attivo).
Fornisce fetchBlockData() (GET del blocco via ID), deletePrompt(promptId) per cancellare un singolo prompt dal blocco, handleView(promptId) per recuperare e mostrare i risultati di test di un prompt, e handleEdit(promptId, editedPrompt) per aggiornare un prompt.
Raggruppa in un unico contesto tutte le operazioni CRUD e di navigazione sui prompt di un insieme.

4.3.1.7 QuestionBlockContext:

Gestisce l'elenco globale degli insiemi di domande disponibili e permette di aggiungerne o rimuoverne uno. Lo stato principale è questionBlocks (array di blocchi), mentre espone fetchQuestionBlocks(), addQuestionBlock(newBlock) e deleteQuestionBlock(id).
Fornisce una singola fonte di dat per i blocchi di domande in tutta l'app.

4.3.1.8 LLMManagerContext:

Centralizza la gestione dell'elenco di tutti i modelli LLM registrati. Mantiene LLMList (array di LLM) ed espone fetchLLMList() e deleteLLM(id).
Garantendo che tutti i componenti che usano la lista di modelli siano coerenti.

4.3.2 Componenti

4.3.2.1 Home Page

La Home è la vista di atterraggio che compare all'utente non appena accede alla web-app: in alto ospita una barra fissa che integra la navbar con i collegamenti alle altre sezioni e un'icona "hamburger" per aprire il menù laterale, tramite il quale l'utente può creare una nuova sessione compilando un form con titolo e descrizione.
Nella parte centrale si visualizza l'elenco delle sessioni già create; ciascuna rappresentata da una card che riporta titolo, descrizione, data dell'ultimo accesso e i pulsanti di modifica ed eliminazione.

Navbar

Descrizione:
La NavBar è la barra di navigazione orizzontale sempre visibile in cima all'interfaccia. Svolge tre ruoli fondamentali: mostra il nome della web-app, i collegamenti alle altre sezioni e mantiene costantemente accessibile l'elenco delle sessioni.

Elementi:
La visibilità del menù laterale è gestita dal meccanismo Offcanvas di Bootstrap: un pulsante con gli attributi data-bs-toggle="offcanvas" e data-bs-target="#offcanvasNavbar" affida al framework l'apertura e la chiusura, perciò non esiste uno stato React dedicato.
Il sotto-menù per la creazione di una nuova sessione, presente nel menù laterale, utilizza invece un accordion Bootstrap. L'espansione e il collasso sono gestiti dagli attributi data-bs-toggle="collapse" e data-bs-target="#collapseForm".
L'elenco delle sessioni, mostrato all'interno dell'Offcanvas, accede direttamente all'array sessions fornito dal SessionContext; la funzione fetchSessions aggiorna tale array al mount del contesto e dopo ogni operazione di creazione, modifica o cancellazione.

Requisiti associati:
RF-01;
RF-02;
RF-03;
RF-04;
RF-05;
RF-06;
RF-07;
RF-08;
RF-09.

Body

Descrizione:
Costituisce la porzione centrale della Home: occupa l'area sotto la NavBar e ospita la griglia di SessionCard che rappresentano le sessioni disponibili. Il componente funge da "collettore" di stato: recupera l'elenco delle sessioni dal SessionContext, decide cosa mostrare nei diversi casi d'uso (lista piena, lista vuota, caricamento)

Elementi:
SessionContext esegue, al proprio mount, un hook useEffect che richiama fetchSessions(). Questa chiamata recupera dal backend l'elenco delle sessioni e salva il risultato$_G$ nello state condiviso sessions. Ogni volta che l'utente crea, modifica o elimina una sessione, i metodi createSession, updateSession

e deleteSession, concludono la loro operazione con un nuovo fetchSessions(), così da mantenere coerenti frontend e backend. Viene utilizzato useSessionContext() per ottenere l'array sessions. Con sessions.map(), per ogni elemento, viene istanziata una SessionCard.

Requisiti associati:
RF-06;
RF-07;
RF-08;
RF-09;
RF-10.

SessionCard

Descrizione:
Componente riutilizzabile che rappresenta una singola sessione. Espone titolo, descrizione, ultimo accesso e due pulsanti di azione.

Elementi:
Il componente è strutturato in due sezioni: Corpo (SessionCardBody), incaricato di visualizzare titolo, descrizione e ultimo accesso. Azioni (SessionCardActions), incaricato di mostrare i pulsanti per modificare o eliminare la sessione. Un clic sul corpo della card porta l'utente alla pagina di dettaglio corrispondente (/sessions/[id]).

Requisiti associati:
RF-11;
RF-12;
RF-13;
RF-14.

4.3.2.2 Session content

Questa pagina si apre quando viene cliccata una SessionCard nella Home o nel menù laterale della NavBar e mostra i dettagli specifici della sessione in questione, permettendo di effettuare le operazioni fornite dal sistema. L'interfaccia si divide in due moduli principali per costruire la pagina: SessionLLMList, è dedicato alla gestione dei modelli: si può aggiungere un nuovo LLM tramite un menù a tendina o rimuovere quelli già associati, tenendo sempre sott'occhio l'elenco aggiornato dei modelli collegati alla sessione. Test Form, riguarda invece l'attività$_G$ di sperimentazione: permette di scegliere gli insiemi di domande da testare (o di caricarli da un file JSON), di avviare un nuovo test e di consultare quelli eseguiti in passato. Al termine di un test il modulo stesso mostra i risultati dettagliati oppure l'elenco dei test precedenti.

SessionLLMPanel

Descrizione:
È il componente orchestratore della sezione LLM: racchiude il form per aggiungere nuovi modelli alla sessione e la lista di quelli già collegati.

Elementi:
Non gestisce stato proprio, infatti legge dal contesto la funzione submitLLM, l'array remainingLLMs e l'elenco corrente sessionData.llm. Quando l'utente seleziona un modello nel menù a tendina e preme "Aggiungi", submitLLM effettua una richiesta POST e il provider aggiorna sessionData così da aggiornare la lista in automatico.

Requisiti associati:

RF-19;
RF-20;
RF-21;
RF-22;
RF-45;
RF-46;
RF-47;
RF-50;
RF-51;
RF-52;
RF-53;
RF-54;
RF-55;
RF-60;
RF-61;
RF-64.

SessionLLMForm

Descrizione:
Form che consente di collegare un nuovo modello alla sessione tramite un menù a tendina, che elenca tutti i LLM già presenti nel sistema ma non ancora associati a questa sessione.

Elementi:
Ottiene da SessionLLMContext remainingLLMs, submitLLM, isLLMDataEmpty e l'eventuale messaggio di limit. Il menù a tendina è popolato con i modelli ancora disponibili; se isLLMDataEmpty è true mostra la voce "Nessun LLM disponibile" e disattiva sia la select sia il pulsante "Aggiungi". Se la sessione ha raggiunto il limite massimo di LLM collegabili, il contesto fornisce la stringa limit che il form mostra con un alert rosso ("Solo un massimo di 3 LLM è ammesso.").

Requisiti associati:
RF-45.

SessionLLMList / SessionLLMCard

Descrizione:
La lista (SessionLLMList) contiene una griglia di card, una per ciascun modello collegato, oppure un messaggio neutro se la sessione non ha LLM. Ogni card (SessionLLMCard) mostra nome e numero di parametri del modello e un pulsante per scollegarlo.

Elementi:
SessionLLMList legge sessionData.llm e inserisce le varie card tramite .map(); SessionLLMCard riceve l'oggetto llm e, al click sul pulsante di eliminazione, invoca deleteLLM(llm.id). Il provider elimina il record dal backend e rigenera sessionData.

Requisiti associati:
-

TestForm

Descrizione:
Modulo multi-step che consente di scegliere i blocchi di domande, caricarli da JSON, avviare un nuovo

test o consultare quelli passati.

Elementi:
Il componente interroga TestFormContext e, in base ai flag, mostra:

- **JSONSelector**: tramite uno switch isJSON consente di passare dalla modalità lista a quella file JSON;
- **JSONView**: il campo file visibile solo in modalità JSON;
- **QuestionBlocksSelector**: lista scrollabile degli insiemi di domande recuperati con fetchQuestionBlocks;
- **TestActions**: due pulsanti, "Inizia il test" (submitToBackend) e "Visualizza test precedenti" (showPrevTests);
- **TestResults** (o **PrevTests**): compare quando activeView vale rispettivamente "results" o "prev".
  - in modalità "results" mostra per ogni blocco le risposte dei vari LLM, i relativi punteggi e un grafico di sintesi, consentendo anche di eliminare singole run;
  - in modalità "prev" elenca i test storici della sessione e permette di aprirne uno per rivederne i dettagli.

Requisiti associati:
RF-19;
RF-20;
RF-21;
RF-22;
RF-46;
RF-60;
RF-61;
RF-64.

TestResults

Descrizione:
Questo componente mostra gli esiti di un test raggruppandoli per insiemi di domande: sotto il nome di ogni insieme appaiono, per ciascun LLM, la domanda, la risposta attesa, la risposta prodotta e le relative valutazioni semantica ed esterna. Al di sotto di questi dettagli viene mostrato un grafico a barre che riassume le medie ottenute da ogni modello sull'insieme, per permettere di confrontare rapidamente le prestazioni.

Elementi:
Viene letto testResults dal contesto per ottenere l'elenco dei blocchi con le relative results e il mapping averages_by_llm usato per popolare il BarChart. L'unica operazione di mutazione è handleDeleteRun(runId), che effettua la DELETE e rimuove la run dall'array results tramite setResults, aggiornando immediatamente l'interfaccia.

Requisiti associati:
RF-52;
RF-53;
RF-54;
RF-55.

4.3.2.3 Gestisci LLM

La pagina "Gestisci LLM" consente all'utente di collegare nuovi modelli di intelligenza artificiale e di gestire quelli già registrati. È raggiungibile tramite la NavBar. L'intera vista è avvolta dal LLMManagerContextProvider, che ne centralizza lo stato. Al centro della pagina compare un'unica card contenitore che mostra nella parte superiore un form per la creazione di un modello, mentre subito sotto visualizza, come griglia, l'elenco degli LLM attualmente collegati.

LLMManager

Descrizione:
È il componente orchestratore: racchiude il form, titoli di sezione e la griglia di card. Interroga il contesto per leggere LLMList e decide se visualizzare la lista, lo stato vuoto o il placeholder di caricamento.

Elementi:
Con l'hook useLLMManagerContext, LLMManager riceve la lista corrente dei modelli (LLMList) e i metodi di servizio$_G$ fetchLLMList e deleteLLM. Al montaggio iniziale il provider invoca subito fetchLLMList, popolando lo stato condiviso con i dati provenienti dal backend; la stessa chiamata viene ripetuta ogni volta che un modello viene creato, modificato o eliminato. In fase di rendering, la lista viene attraversata e ogni elemento genera una GeneralLLMCard all'interno di una griglia Bootstrap.

Requisiti associati:
RF-33;
RF-34;
RF-35;
RF-36;
RF-37;
RF-38;
RF-39;
RF-40;
RF-41;
RF-42;
RF-43.

CreateLLMForm

Descrizione:
Form a due campi (nome, numero di parametri) che permette di registrare un nuovo modello. Include inoltre un pulsante "Carica modelli di Ollama" che interroga l'host locale e compila automaticamente la lista di modelli disponibili.

Elementi:
Viene tenuto in stato solo ciò che serve: name e parameters digitati dall'utente, eventuali formErrors per la validazione immediata, un flag conflict se il back-end risponde 409 (modello già presente) e ollamaError se il caricamento automatico da Ollama fallisce. Il metodo createLLM invia un POST `/llm_list/`; in caso di successo azzera i campi e richiama fetchLLMList (dal context) per aggiornare la lista, altrimenti popola conflict. Il pulsante "Carica modelli di Ollama" attiva loadOllamaModels, che legge /api/ollama/models e, se c'è un problema, scrive ollamaError.

Requisiti associati:
RF-33;
RF-34;
RF-35;
RF-36;
RF-41;
RF-42.

GeneralLLMCard

Descrizione:
Card che rappresenta un singolo modello collegato. Visualizza il nome e il numero di parametri e offre un solo pulsante che permette di eliminare il modello.

Elementi:
Il componente riceve in ingresso l'oggetto llm e, al click sul pulsante "Elimina", invoca deleteLLM(llm.id) esposto dal contesto; una volta completata l'operazione, il contesto richiama fetchLLMList() così la rimozione del modello si riflette immediatamente nell'interfaccia. Non possiede stato interno: la sua unica logica è quella legata all'azione di cancellazione.

Requisiti associati:
RF-38;
RF-39;
RF-40;
RF-43;

4.3.2.4 Confronta risultati

La pagina "Confronta risultati" consente di mettere a paragone, nel contesto di una stessa sessione, le prestazioni di due LLM sugli insiemi di domande a cui hanno risposto entrambi, esponendo in forma grafica le rispettive valutazioni semantiche ed esterne.

TestComparator

Descrizione:
È il componente orchestratore della pagina: racchiude il selettore della sessione, i due menù per scegliere i modelli e il grafico finale.

Elementi:
Interroga il TestComparatorContext per ottenere selectedSessionData, selectedLLMS e chartData; in base a questi valori decide se mostrare i selettori o uno stato "vuoto" e quando i dati sono pronti, rende il grafico di confronto. Un useEffect interno al contesto richiede i dati di confronto ogni volta che entrambi gli ID dei modelli sono impostati, lasciando a TestComparator il solo compito di orchestrare il rendering.

Requisiti associati:
RF-53;
RF-56;
RF-57.

SessionSelector

Descrizione:
Mostra un menu a tendina con tutte le sessioni disponibili create; l'utente deve sceglierne una per proseguire.

Elementi:
Consulta l'array sessions dal SessionContext; quando l'utente seleziona un valore diverso da "Seleziona una sessione", richiama fetchSessionData(id) nel TestComparatorContext. Questa funzione scarica dal backend tutti i dati della sessione selezionata e, prima di procedere, azzera selectedLLMS, così i menu dei modelli vengono svuotati e si riparte da zero con la nuova sessione.

Requisiti associati:
-

LLMSelector

Descrizione:
Diventa visibile dopo che la sessione è stata scelta e mostra due <select> affiancati, uno per ciascun modello da confrontare.

Elementi:
Le opzioni sono calcolate partendo dai modelli presenti in selectedSessionData; quando l'utente sceglie il primo LLM, questo viene rimosso dalle opzioni del secondo elenco e viceversa, grazie all'aggiornamento di selectedLLMS nel contesto. Non appena entrambi gli ID sono valorizzati, il contesto lancia fetchBlockComparisonData(firstLLM, secondLLM) per ottenere i risulati.

Requisiti associati:
-

LLMComparisonChart

Descrizione:
Visualizza due grafici a barre orizzontali, uno per la valutazione Semantica e uno per la valutazione Esterna. In ciascun grafico, per ogni blocco di domande compaiono due barre, una per ciascuno dei modelli scelti, consentendo una visualizzazione immediata del confronto delle rispettive medie.

Elementi:
Legge da TestComparatorContext tre valori: chartData, llmNames e selectedLLMS. Per ciascun <Bar> costruisce dinamicamente il dataKey, concatenando il nome del modello (ricavato da llmNames) con il tipo di valutazione ("Semantica" o "Esterna"). Non possiede stato interno: quando il contesto aggiorna chartData, i grafici si ridisegnano automaticamente grazie al binding reattivo di React.

Requisiti associati:
RF-53;
RF-56;
RF-57.

4.3.2.5 Insiemi di domande

La pagina "Insiemi di domande" permette alll'utente di creare un nuovo question-block – cioè un insieme di coppie Domanda / Risposta attesa – e di gestire quelli già salvati.
L'intera vista vive dentro al QuestionBlockProvider, che mantiene la collezione corrente, espone le operazioni di creazione e cancellazione e provvede a ricaricare la lista dopo ogni mutazione. In alto compare il form di creazione; subito sotto, la griglia delle card che rappresentano i blocchi esistenti. Se la lista è vuota viene mostrato il messaggio "Nessun insieme disponibile".

CreateBlockForm

Descrizione:
Form principale della pagina: raccoglie il nome dell'insieme e un numero arbitrario di righe domanda/-risposta. Un avviso toast "Insieme di domande creato con successo!" conferma la creazione avvenuta con successo.

Elementi:
Il custom-hook useCreateBlockFormHook gestisce gli stati interni: name, l'array questionAnswerPairs, gli eventuali formErrors di validazione e i flag conflict / toastVisible. L'handler handleSubmit invia un POST `/question_blocks/` con i dati compilati; al successo svuota i campi, mostra il toast (SuccessToast) e richiama addQuestionBlock del contesto per aggiornare immediatamente la lista. Il pulsante "Aggiungi riga" usa addQuestionAnswerPair per accodare una nuova coppia vuota; quello di rimozione chiama removeQuestionAnswerPair(index).

Requisiti associati:
RF-15;
RF-16;
RF-17;
RF-18;
RF-23;
RF-24;
RF-25;
RF-26;
RF-27;
RF-28;
RF-29;
RF-30;
RF-31.

BlockNameInput

Descrizione:
Mostra l'input in cui l'utente scrive il nome del nuovo insieme di domande, all'interno del CreateBlockForm.

Elementi:
Collega il ref `inputRefs.current.block_name` al campo testo, affinché il hook possa leggerne il valore all"invio senza passare da stato React aggiuntivo.

Requisiti associati:
-

QuestionAnswerRow

Descrizione:
Riga ripetibile che contiene un input per la domanda, uno per la risposta attesa e, se non è la prima riga, un pulsante per la rimozione.

Elementi:
Riceve pair, index, onChange, onRemove e un flag canRemove che determina il layout (pulsante per la rimozione). I due <input> aggiornano direttamente la coppia in posizione index tramite onChange(index, e). Ogni input registra il proprio nodo nell'oggetto inputRefs (es. inputRefs.current['question-2']), così il form padre può leggere tutti i valori in blocco al momento dell'invio.

Requisiti associati:
RF-15;
RF-16;
RF-17;
RF-18;
RF-24;
RF-25;
RF-26;
RF-27;
RF-28;
RF-29;
RF-30;

QuestionBlockCard

Descrizione:
Card cliccabile che riassume un blocco di domande già salvato. Il click sul corpo porta alla pagina di dettaglio (/question-blocks/[id]); il pulsante rosso in basso elimina il blocco.

Elementi:
Riceve l'oggetto block e la callback onDelete. Quando l'utente preme "Elimina insieme", la card invoca deleteQuestionBlock(block.id) esposto dal contesto; al termine dell'operazione il provider riesegue automaticamente fetchQuestionBlocks(), così la lista si aggiorna immediatamente.

Requisiti associati:
RF-24;
RF-25;
RF-26;

4.3.2.6 Insieme di domande - Inspect block

Quando l'utente seleziona una QuestionBlockCard si apre la pagina Inspect Block, che mostra nel dettaglio il contenuto dell'insieme scelto. La vista, racchiusa dal QuestionBlockProvider, riutilizza i dati già caricati nella pagina "Insiemi di domande". In alto compaiono il nome dell'insieme e il numero totale di prompt presenti, mentre subito sotto, una griglia di card rappresenta ciascuna coppia "domanda / risposta attesa". Ogni card offre la possibilità di modificare o eliminare la coppia e può essere espansa per visualizzare le run in cui quel prompt è stato utilizzato, mostrando le risposte fornite dai diversi LLM a quella coppia con l'evidenza della valutazione migliore e di quella peggiore.

BlockHeader

Descrizione:
Mostra il titolo dell'insieme di domande e, subito sotto, il numero totale di prompt che lo compongono.

Elementi:
Riceve in props name e promptCount e li mostra a schermo

Requisiti associati:
-

PromptList

Descrizione:
È il contenitore che visualizza tutte le coppie "Domanda / Risposta attesa" appartenenti al blocco selezionato.

Elementi:
Accetta l'array prompts e lo percorre con map(), racchiudendo ogni elemento in una colonna Bootstrap. Per ciascun prompt genera una PromptCard. Non possiede logica propria oltre al rendering.

Requisiti associati:
RF-24;
RF-25;
RF-26;

PromptCard

Descrizione:
Card che visualizza una singola coppia "Domanda / Risposta attesa" e permette all'utente di visualizzare le run, modificare la coppia oppure eliminarla.

Elementi:
Il componente ottiene dal contesto le funzioni deletePrompt, handleView e handleEdit. Il pulsante "Visualizza run" invoca handleView(prompt.id), caricando i risultati nel contesto. "Modifica" attiva la modalità di editing locale (isEditing), salvando le modifiche tramite handleEdit(prompt.id, editedPrompt). "Elimina" richiama deletePrompt(prompt.id) e, grazie all'aggiornamento effettuato dal provider, la riga scompare immediatamente dalla lista. A parte il flag isEditing, la card non mantiene altri stati.

Requisiti associati:
RF-27;
RF-28;
RF-29;
RF-30;
RF-31;

PromptResults

Descrizione:
Mostra i risultati delle run associate al prompt selezionato, mettendo in evidenza il migliore e il peggiore, fornendo il dettaglio di ogni risposta e, per ciascun LLM, visualizzando un grafico a barre con le valutazioni semantica ed esterna.

Elementi:
Il componente prende testResults dal contesto e genera:

- summaryResults: riquadri "risultato migliore" e "risultato peggiore";
- l'elenco completo delle risposte al prompt nelle varie run in cui è stato utilizzato;
- la serie di valori (semantica / esterna) nel formato richiesto dal BarChart di Recharts.

Non tocca lo stato globale. L'unica mutazione è che avviene in locale è tramite handleDeleteRun(runId) che rimuove la run da results con setResults.

Requisiti associati:
RF-58;
RF-59;

4.4 Architettura del backend

L'organizzazione strutturale del backend e del microservizio dedicato all'integrazione con Ollama riflette i principi organizzativi dell'architettura a layer. In particolare, le classi sono organizzate in:

- Repository$_G$;
- Servizi;
- Viste;
- Modelli;
- Serializzatori.

I file denominati `urls.py` associano ad ogni vista un endpoint$_G$ specifico. Le viste a loro volta chiamano i servizi adatti, i quali contengono tutta la logica di business necessaria ad eseguire le operazioni dettate dall'utente. I servizi utilizzano i repository per creare, leggere, modificare ed eliminare i dati presenti all'interno del database. Una volta terminate le operazioni, le viste utilizzano i serializzatori per convertire gli oggetti in formato JSON e procedono poi a ritornare come risposta lo stato dell'operazione ed eventualmente i dati necessari alla visualizzazione dei risultati nel frontend. Le tipologie di entità disponibili sono definite all'interno dei modelli, i quali definiscono la composizione delle tabelle nel database e le eventuali relazioni fra loro.

4.4.1 Modelli

I modelli sono classi utilizzate da Django per rappresentare le tabelle presenti all'interno del database. Una volta definiti i modelli, è possibile generare automaticamente le migrazioni per popolare la base di dati correttamente.

4.4.1.1 LLM

Descrizione

Rappresenta i modelli di linguaggio (Large Language Models) installati all'interno del sistema ArtificialQI (es. quelli caricati tramite Ollama).

Campi

- `name`: Nome univoco del modello;
- `n_parameters`: Numero di parametri del modello (come testo).

Relazioni

- Relazione `ManyToMany` con `Session`;
- Relazione `ForeignKey` da `Run`.

4.4.1.2 Session

Descrizione

Rappresenta una sessione di benchmark composta da un massimo di 3 modelli LLM.

Campi

- `title`: Titolo univoco della sessione;
- `description`: Descrizione della sessione;
- `created_at`: Timestamp di creazione;
- `updated_at`: Timestamp dell'ultimo aggiornamento.

Relazioni

- Relazione `ManyToMany` con `LLM`;
- Relazione `ForeignKey` da `BlockTest`.

4.4.1.3 Prompt

Descrizione

Rappresenta una coppia domanda/risposta attesa utilizzata nei test.

Campi

- `prompt_text`: Il testo della domanda;
- `expected_answer`: La risposta attesa;
- `timestamp`: Data e ora di creazione.

Relazioni

- Relazione `ManyToMany` con `Block`;
- Relazione `ForeignKey` da `Run`.

4.4.1.4 Evaluation

Descrizione

Contiene le valutazioni (semantica ed esterna) di una risposta fornita da un modello.

Campi

- `semantic_evaluation`: Valutazione semantica (decimale);
- `external_evaluation`: Valutazione esterna (decimale).

Relazioni

- Relazione `ForeignKey` da `Run`.

4.4.1.5 Block

Descrizione

Rappresenta un blocco di domande, utilizzato per raggruppare i prompt nei test.

Campi

- `name`: Nome del blocco.

Relazioni

- Relazione `ManyToMany` con `Prompt`;
- Relazione `ForeignKey` da `BlockTest`.

4.4.1.6 Run

Descrizione

Rappresenta una singola esecuzione di test per un LLM su un prompt.

Campi

- `llm_answer`: Risposta fornita dal modello.

Relazioni

- `ForeignKey` verso `LLM`;
- `ForeignKey` verso `Prompt`;
- `ForeignKey` verso `Evaluation`;
- Relazione `ManyToMany` da `BlockTest`.

4.4.1.7 BlockTest

Descrizione

Rappresenta un test richiesto da un utente, che include un blocco di domande e una sessione di modelli.

Campi

- `timestamp`: Data e ora di creazione del test.

Relazioni

- `ForeignKey` verso `Session`;
- `ForeignKey` verso `Block`;
- `ManyToMany` con `Run`.

4.4.1.8 Schema E-R generato

I modelli sopra elencati generano, tramite il sistema di migrazioni di Django, la seguente base di dati:

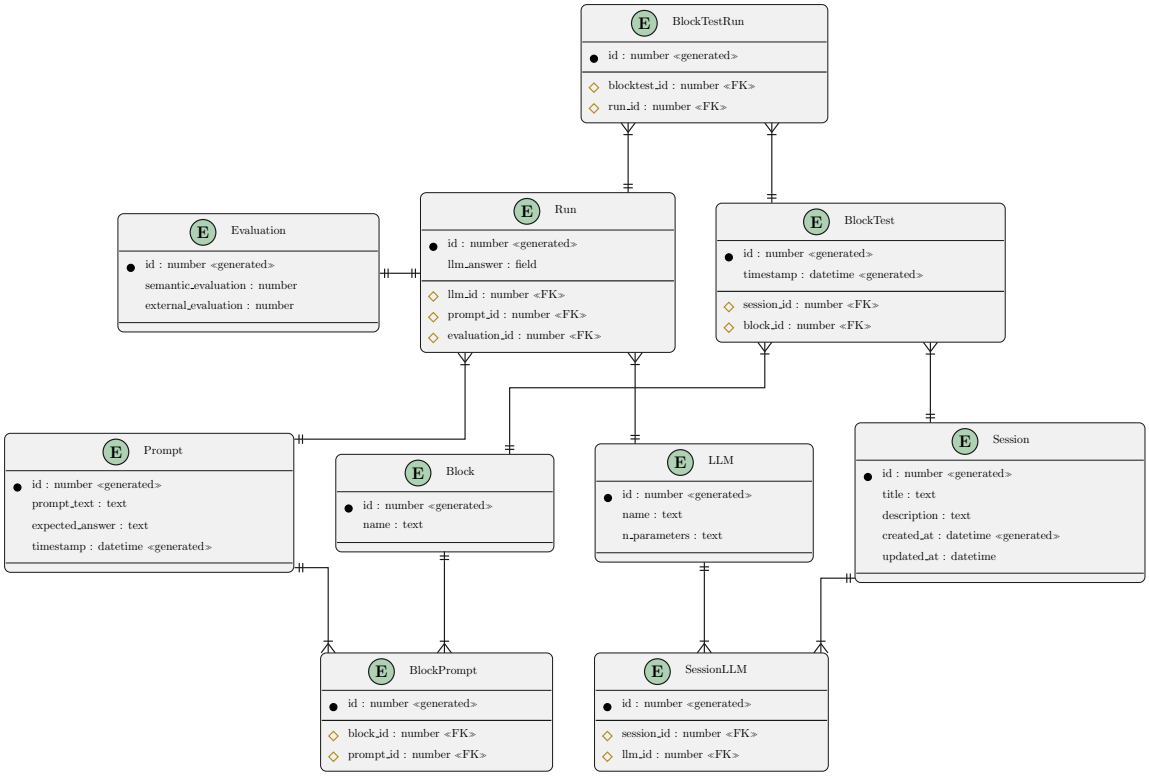

Figura 2: Schema E-R della base di dati.

4.4.2 Repository

4.4.2.1 AbstractRepository

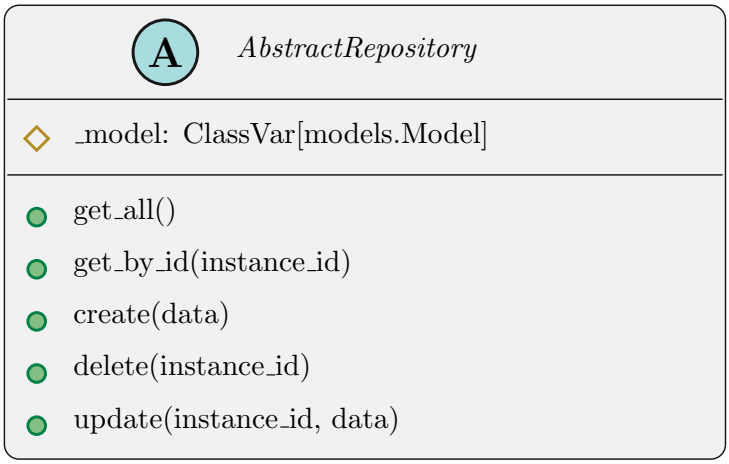

Figura 3: Diagramma UML della classe AbstractRepository.

Attributi

- _model: ClassVar[models.Model]

Metodi

- get_all(cls)
- get_by_id(cls, instance_id: int) -⟩models.Model — None
- create(cls, data: dict) -⟩models.Model
- delete(cls, instance_id: int) -⟩bool
- update(cls, instance_id: int, data: dict) -⟩models.Model — None

Descrizione

Questa classe repository astratta definisce i metodi comuni a tutte le classi repository derivate. Gestisce le operazioni di:

- **Fetch di tutte le istanze dal database** tramite chiamata al metodo `get_all`;
- **Fetch di una istanza singola** tramite chiamata al metodo `get_by_id`;
- **Creazione di una nuova istanza** tramite chiamata al metodo `create`;
- **Cancellazione di una istanza esistente** tramite chiamata al metodo `delete`;
- **Aggiornamento di una istanza esistente** tramite chiamata al metodo `update`.

4.4.2.2 BlockRepository

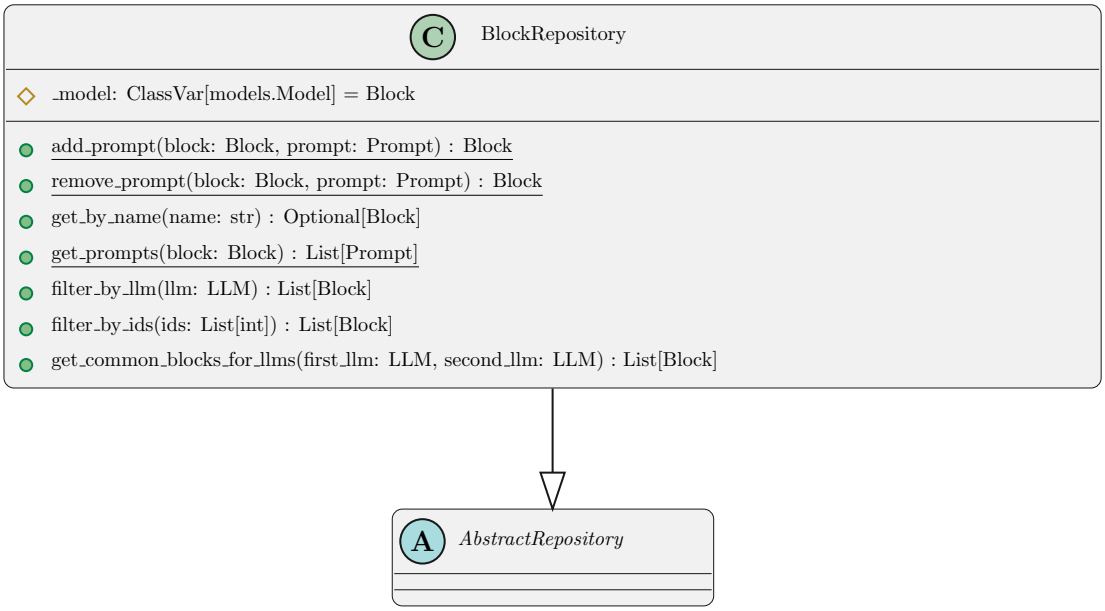

Figura 4: Diagramma UML della classe BlockRepository.

Attributi

- _model: ClassVar[models.Model] = Block

Metodi

- add_prompt(block: Block, prompt: Prompt) -⟩Block
- remove_prompt(block: Block, prompt: Prompt) -⟩Block
- get_by_name(name: str) -⟩Block — None
- get_prompts(block: Block) -⟩List[Prompt]
- filter_by_llm(llm: LLM) -⟩List[Block]
- filter_by_ids(ids: List[int]) -⟩List[Block]
- get_common_blocks_for_llms(first_llm: LLM, second_llm: LLM) -⟩List[Block]

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano i blocchi di domande nel database. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Aggiunta di una domanda ad un blocco esistente;
- Rimozione di una domanda da un blocco esistente;
- Filtraggio di un blocco per nome (attributo univoco);
- Filtraggio di tutte le domande appartenenti ad un blocco;
- Filtraggio di blocchi multipli per identificativi;
- Filtraggio di blocchi a cui hanno risposto due LLM contemporaneamente.

Dipendenze

- AbstractRepository
- Block

4.4.2.3 BlockTestRepository

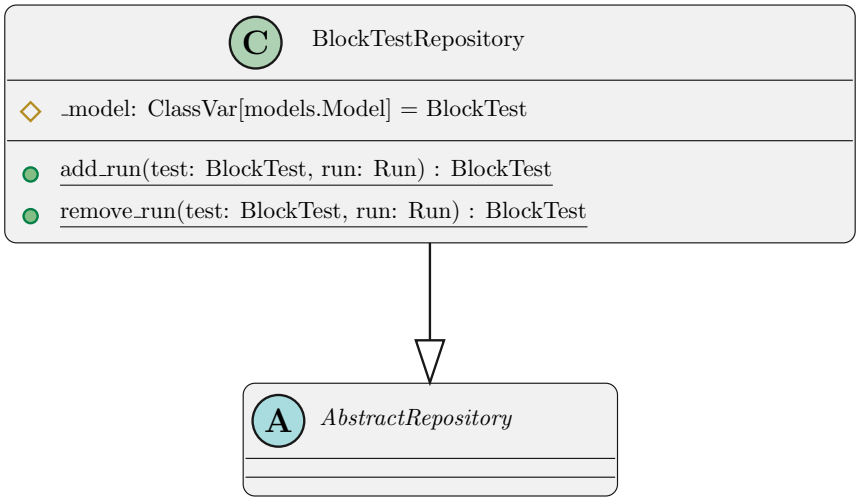

Figura 5: Diagramma UML della classe BlockTestRepository.

Attributi

- _model: ClassVar[models.Model] = BlockTest

Metodi

- add_run(test: BlockTest, run: Run) -⟩BlockTest
- remove_run(test: BlockTest, run: Run) -⟩BlockTest

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano i test richiesti dall'utente. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Aggiunta di una istanza di tipo Run ad un BlockTest esistente;
- Rimozione di una istanza di tipo Run da un BlockTest esistente.

Dipendenze

- AbstractRepository
- BlockTest

4.4.2.4 EvaluationRepository

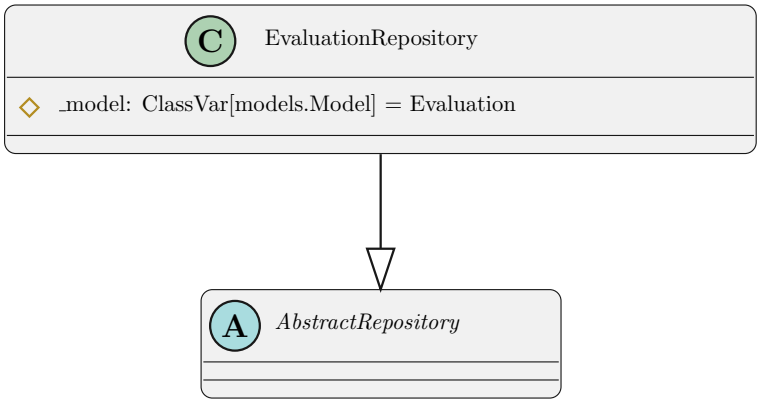

Figura 6: Diagramma UML della classe EvaluationRepository.

Attributi

- _model: ClassVar[models.Model] = Evaluation

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano le valutazioni alle risposte dei modelli.

Dipendenze

- AbstractRepository;
- Evaluation.

4.4.2.5 LLMRepository

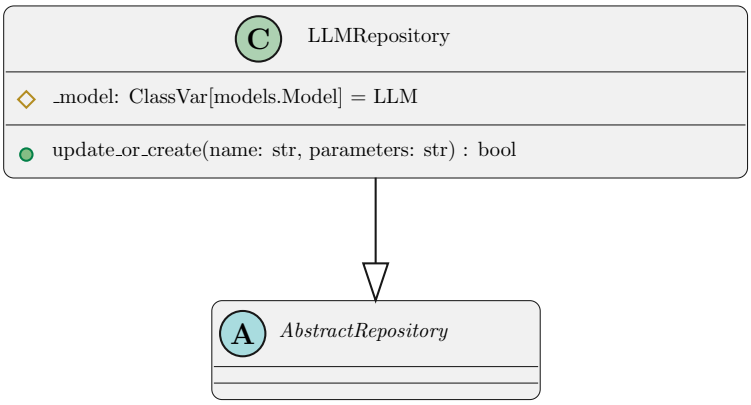

Figura 7: Diagramma UML della classe LLMRepository.

Attributi

- _model: ClassVar[models.Model] = LLM

Metodi

- update_or_create(name: str, parameters: str) -⟩bool

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano i modelli. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Aggiornamento o creazione di un modello di un modello dato il nome di quest'ultimo.

Dipendenze

- AbstractRepository;
- LLM.

4.4.2.6 PromptRepository

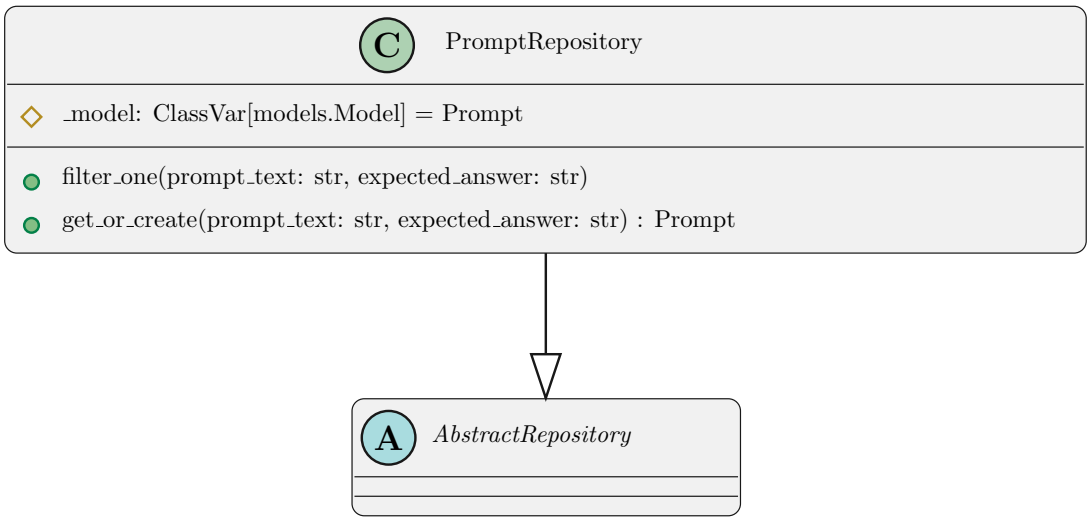

Figura 8: Diagramma UML della classe PromptRepository.

Attributi

- _model: ClassVar[models.Model] = Prompt

Metodi

- filter_one(cls, prompt_text: str, expected_answer: str)
- get_or_create(cls, prompt_text: str, expected_answer: str) -⟩Prompt

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano le coppie domanda/risposta attesa. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Filtraggio di un prompt per testo della domanda e testo della risposta attesa;
- Creazione o fetch di un prompt dato il testo della domanda e della risposta attesa.

Dipendenze

- AbstractRepository;
- Prompt.

4.4.2.7 RunRepository

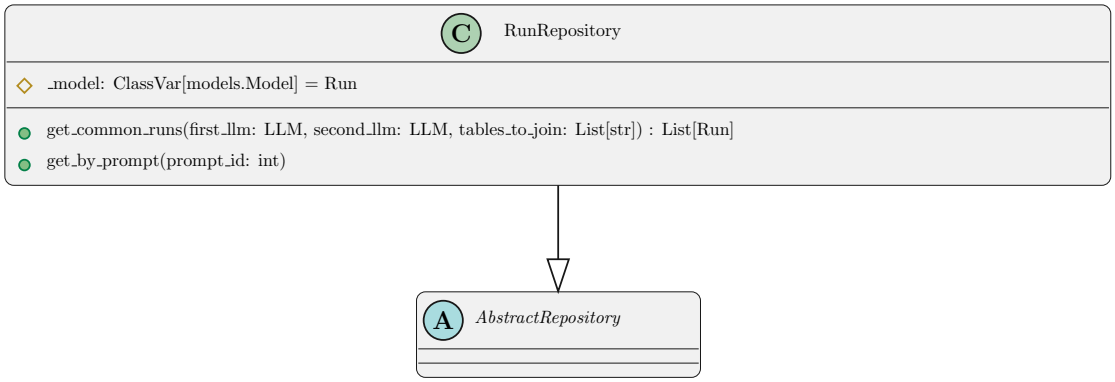

Figura 9: Diagramma UML della classe RunRepository.

Attributi

- _model: ClassVar[models.Model] = Run

Metodi

- get_common_runs(cls, first_llm: LLM, second_llm: LLM, tables_to_join: List[str]) -⟩List[Run]
- get_by_prompt(cls, prompt_id: int)

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano le istanze singole di interrogazione e valutazione della risposta di un modello. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Filtraggio delle Run comuni a due LLM;
- Filtraggio dei blocchi dato un determinato Prompt.

Dipendenze

- AbstractRepository;
- Run.

4.4.2.8 SessionRepository

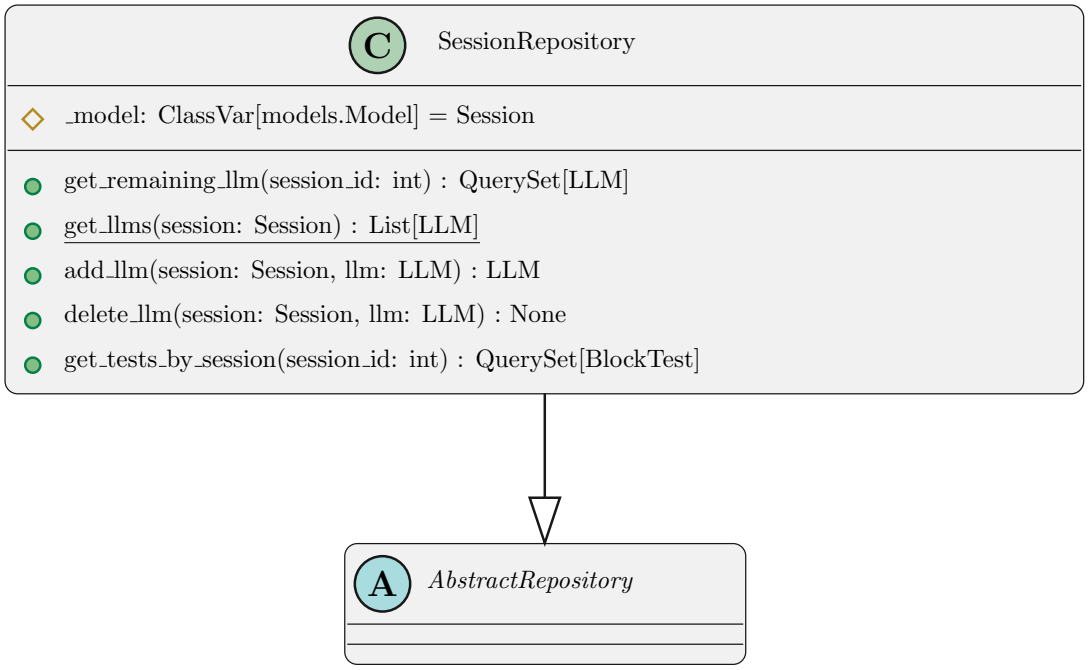

Figura 10: Diagramma UML della classe SessionRepository.

Attributi

- _model: ClassVar[models.Model] = Session

Metodi

- get_remaining_llm(cls, session_id: int)
- get_llms(session: Session)
- add_llm(cls, session: Session, llm: LLM) -⟩LLM
- delete_llm(cls, session: Session, llm: LLM) -⟩None
- get_tests_by_session(cls, session_id: int)

Descrizione

Questa classe repository astratta derivata da AbstractRepository definisce i metodi eseguibili per creare, leggere, filtrare, modificare ed eliminare le istanze che rappresentano le istanze delle Sessioni. Gestisce tutte le operazioni definite in AbstractRepository, oltre a:

- Filtraggio di tutti i modelli non collegati ad una sessione;
- Filtraggio di tutti i modelli collegati ad una sessione;
- Aggiunta di un LLM ad una sessione;
- Cancellazione di un modello da una sessione;
- Fetch di tutti i test in una sessione.

Dipendenze

- AbstractRepository;
- Session.

4.4.3 Servizi

4.4.3.1 AbstractService

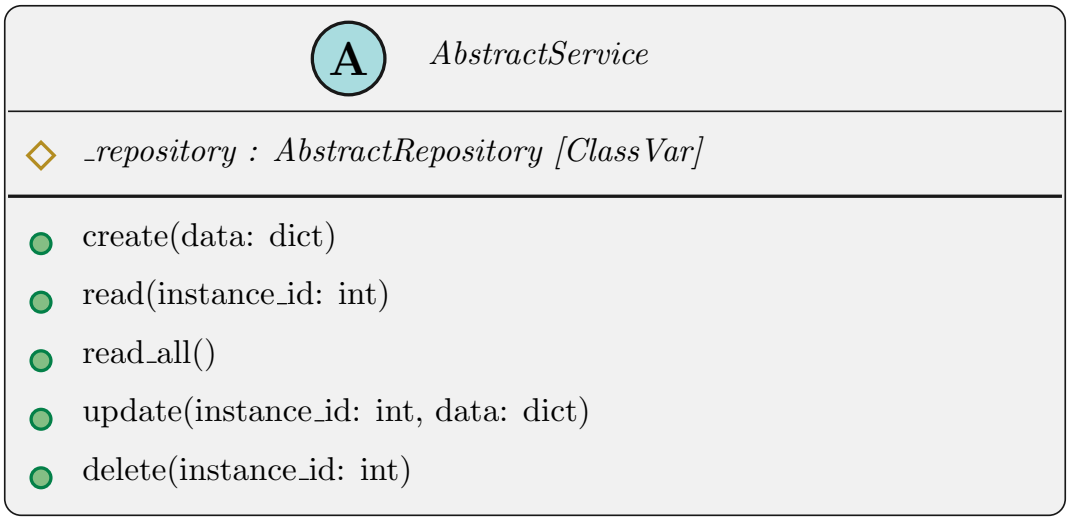

Figura 11: Diagramma UML della classe AbstractService.

Attributi

- _repository: ClassVar[AbstractRepository]

Metodi

- create(cls, data: dict)
- read(cls, instance_id: int)
- read_all(cls)
- update(cls, instance_id: int, data: dict)
- delete(cls, instance_id: int)

Descrizione

Questa classe definisce una struttura astratta per tutti i servizi derivanti da essa. Sono definite le operazioni di lettura, creazione, modifica e cancellazione delle istanze.

4.4.3.2 BlockService

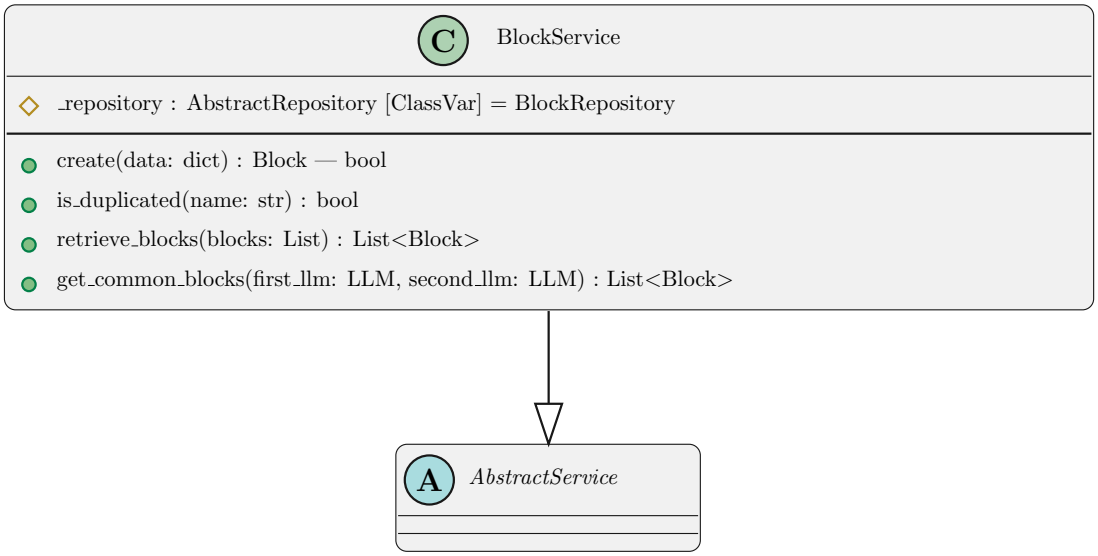

Figura 12: Diagramma UML della classe BlockService.

Attributi

- _repository: ClassVar[AbstractRepository] = BlockRepository

Metodi

- create(cls, data: dict) -⟩Block — bool
- is_duplicated(cls, name: str)
- retrieve_blocks(cls, blocks: List[Block])
- get_common_blocks(cls, first_llm: LLM, second_llm: LLM) -⟩List[Block]

Descrizione

Questa classe definisce la logica di business per la gestione delle operazioni riguardanti gli insiemi di coppie domanda/risposta attesa definiti dall'utente. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Override della funzione create di AbstractService, permette la creazione di un blocco e di popolarlo direttamente con le coppie domanda/risposta attesa inserite dall'utente senza duplicare eventuali coppie già presenti;
- Controllo dell'esistenza di un insieme con nome duplicato;
- Fetch di tutte le istanze di Block a partire da una lista in formato JSON;
- Fetch di tutti i blocchi a cui hanno risposto due LLM.

Dipendenze

- BlockRepository;
- AbstractService.

4.4.3.3 BlockTestService

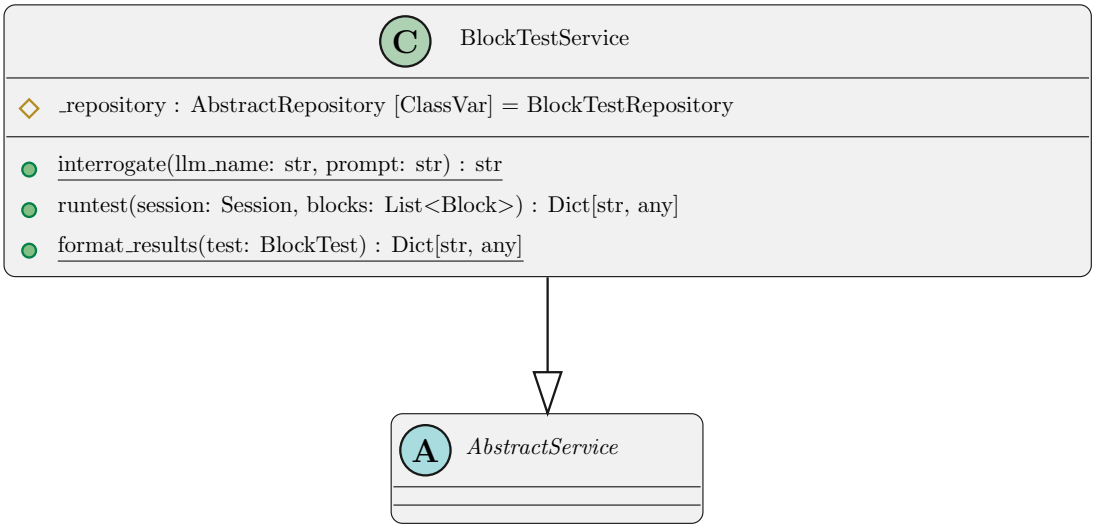

Figura 13: Diagramma UML della classe BlockTestService.

Attributi

- _repository: ClassVar[AbstractRepository] = BlockTestRepository

Metodi

- interrogate(llm_name: str, prompt: str) -⟩str
- runtest(cls, session: Session, blocks: List[Block]) -⟩Dict[str, any]
- format_results(test: BlockTest) -⟩Dict[str, any]

Descrizione

Questa classe definisce la logica di business per la gestione dei test. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Interrogazione di un modello tramite microservizio per LLM;
- Esecuzione di un test;
- Formattazione dei risultati di un test per la visualizzazione nel frontend.

Dipendenze

- BlockTestRepository;
- SessionRepository;
- BlockRepository;
- AbstractRepository;
- AbstractService;
- EvaluationService;
- RunService;
- LLMController.

4.4.3.4 EvaluationService

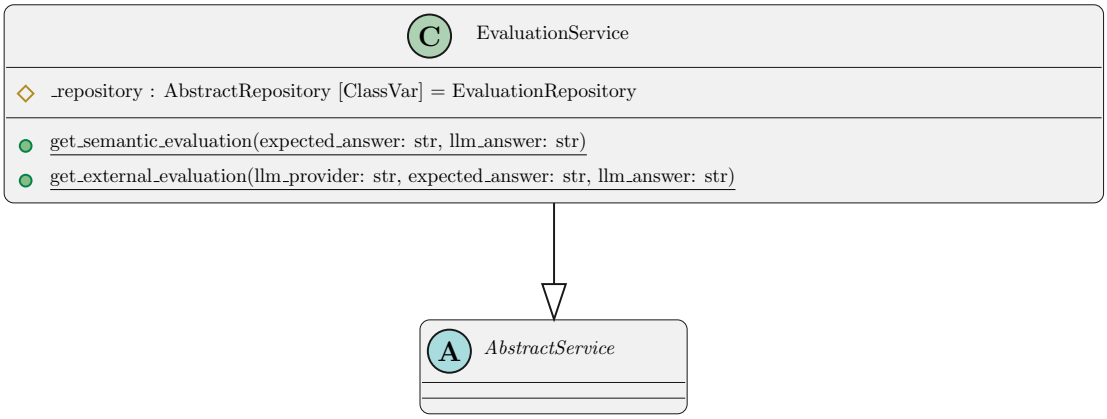

Figura 14: Diagramma UML della classe EvaluationService.

Attributi

- _repository: ClassVar[AbstractRepository] = EvaluationRepository

Metodi

- get_semantic_evaluation(expected_answer: str, llm_answer: str)
- get_external_evaluation(llm_provider: str, expected_answer: str, llm_answer: str)

Descrizione

Questa classe definisce la logica di business per la gestione delle valutazioni. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Calcolo della valutazione semantica;
- Calcolo della valutazione esterna.

Dipendenze

- EvaluationRepository;
- AbstractService.

4.4.3.5 LLMService

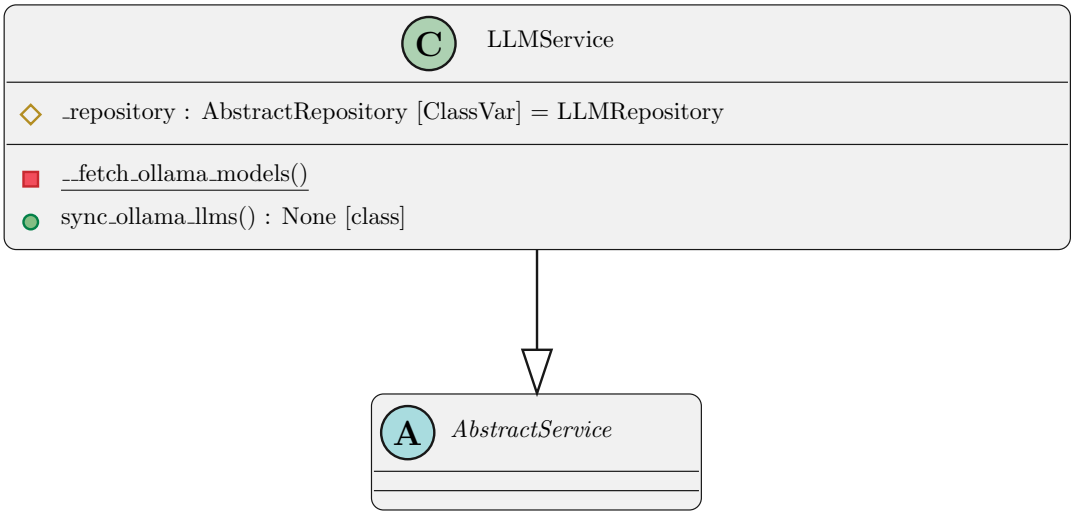

Figura 15: Diagramma UML della classe LLMService.

Attributi

- _repository: ClassVar[AbstractRepository] = LLMRepository

Metodi

- __fetch_ollama_models()
- sync_ollama_llms(cls) -⟩None

Descrizione

Questa classe definisce la logica di business per la gestione dei LLM. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Fetch dei LLM connessi ad Ollama tramite chiamata API al microservizio dedicato;
- Sincronizzazione dei LLM connessi ad Ollama con Database.

Dipendenze

- AbstractService;
- LLMRepository.

4.4.3.6 PromptService

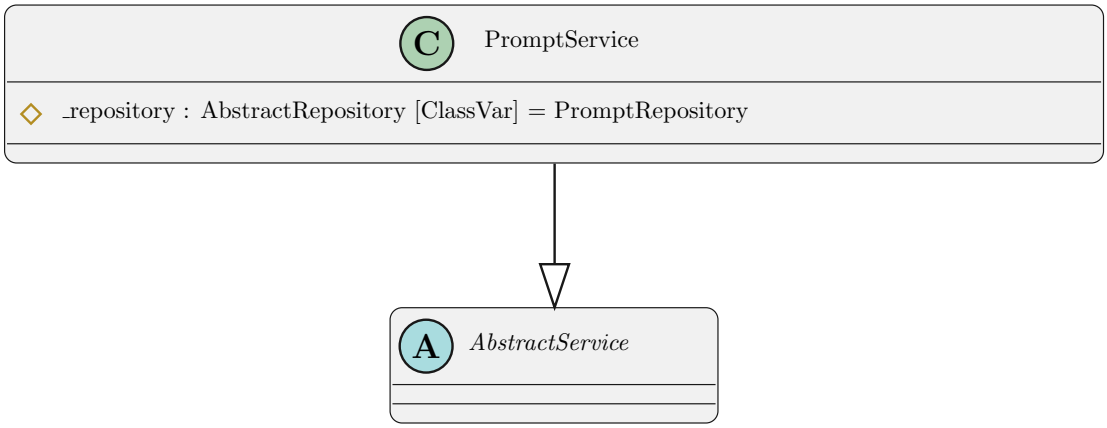

Figura 16: Diagramma UML della classe PromptService.

Attributi

- _repository: ClassVar[AbstractRepository] = PromptRepository

Descrizione

Questa classe definisce la logica di business per la gestione delle singole coppie domanda/risposta attesa.

Dipendenze

- AbstractService;
- PromptRepository.

4.4.3.7 RunService

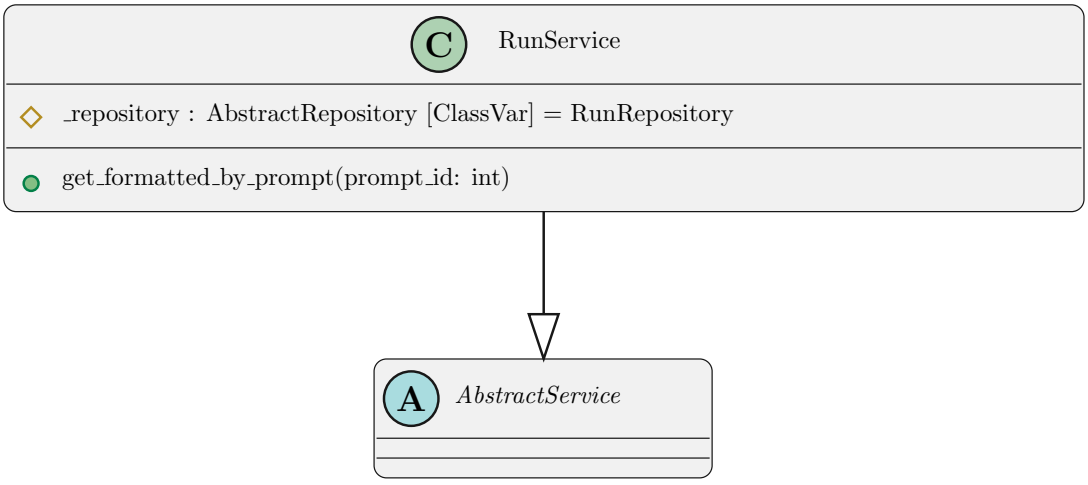

Figura 17: Diagramma UML della classe RunService.

Attributi

- _repository: ClassVar[AbstractRepository] = RunRepository

Metodi

- get_formatted_by_prompt(cls, prompt_id: int)

Descrizione

Questa classe definisce la logica di business per la gestione delle singole istanze di Run. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Formattazione dei dati relativi ad un Prompt specifico per la visualizzazione nel frontend.

Dipendenze

- RunRepository;
- PromptRepository;
- AbstractService.

4.4.3.8 SessionService

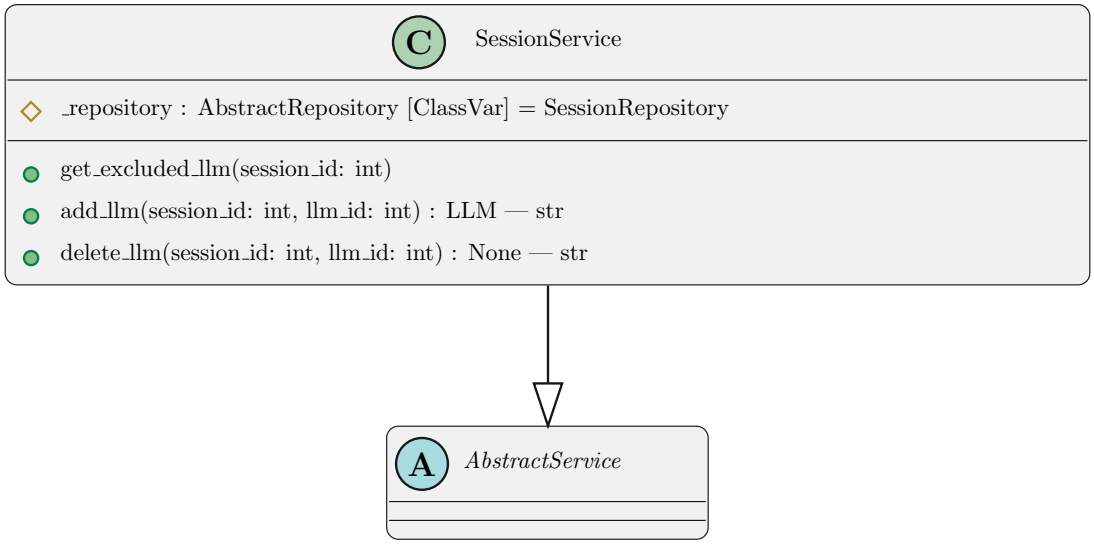

Figura 18: Diagramma UML della classe SessionService.

Attributi

- _repository: ClassVar[AbstractRepository] = SessionRepository

Metodi

- get_excluded_llm(cls, session_id: int)
- add_llm(cls, session_id: int, llm_id: int) -¿ LLM — str
- delete_llm(cls, session_id: int, llm_id: int) -¿ None — str
- get_tests_by_session(cls, session_id: int)

Descrizione

Questa classe definisce la logica di business per la gestione delle sessioni. In particolare, oltre alle operazioni definite in AbstractService, sono disponibili le seguenti funzionalità:

- Fetch di tutti i modelli che non appartengono ad una sessione;
- Aggiunta di un LLM ad una sessione;
- Rimozione di un LLM da una sessione;
- Fetch di tutti i test di una sessione.

Dipendenze

- AbstractService;
- SessionRepository;
- LLMRepository.

4.4.3.9 OllamaLLMIntegrationService

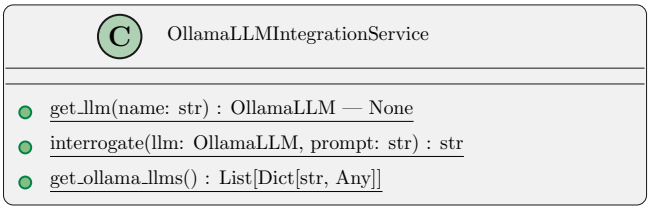

Figura 19: Diagramma UML della classe OllamaLLMIntegrationService.

Metodi

- get_llm(name: str) -¿ OllamaLLM — None
- interrogate(llm: OllamaLLM, prompt: str) -¿ str
- get_ollama_llms() -¿ List[Dict[str, Any]]

Descrizione

Questa classe appartenente al microservizio per modelli LLM si occupa dell'integrazione con Ollama. Sono disponibili le seguenti funzionalità:

- Creazione di un oggetto di tipo OllamaLLM (classe di LangChain$_G$) a partire dal nome del modello;
- Interrogazione di un modello a partire dal suo oggetto di tipo OllamaLLM (classe di LangChain) e dal prompt;
- Fetch di tutti i modelli installati su Ollama.

4.4.4 Viste

Le viste sono classi dedicate alla gestione delle richieste effettuate dal frontend. Ogni vista è associata ad un determinato endpoint e ad una particolare tipologia di metodo (GET, POST, PUT o DELETE). Le viste che si occupano della creazione, modifica, eliminazione o lettura dei dati derivano tutte da AbstractView, le viste che non si occupano di queste operazioni derivano invece da APIView. Le definizioni delle viste sono tutte presenti nella cartella `views_def` e nei file `views.py`.

4.4.4.1 AbstractView

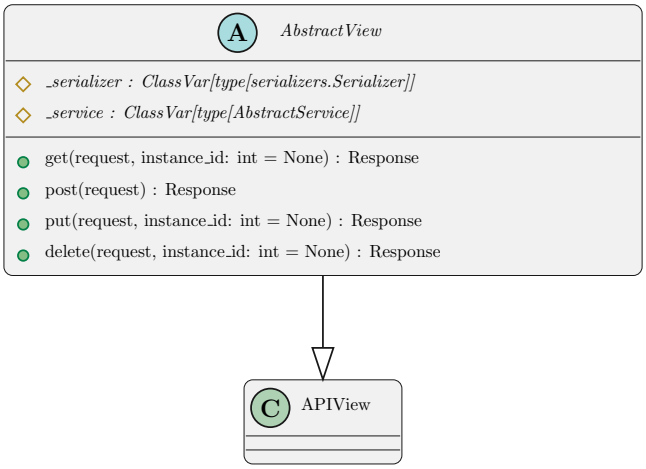

Figura 20: Diagramma UML della classe AbstractView.

Descrizione

La classe astratta AbstractView definisce le operazioni CRUD da eseguire quando, rispettivamente, le viste vengono chiamate utilizzando metodi di tipo `GET, POST, PUT o DELETE`. È dotata di due attributi in modo tale da assegnare facilmente il giusto serializer e il giusto service$_G$ alle viste derivate.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Recupera tutti gli oggetti o uno specifico se fornito `instance_id`.	200 OK: Dati serializzati
`GET`	Errore durante il recupero.	400 Bad Request: {`"error": "..."`}
`POST`	Crea una nuova istanza con i dati validati.	201 Created: Dati serializzati
`POST`	Dati non validi o errore interno.	400 Bad Request: {`"error": "..."`}
`PUT`	Aggiorna una istanza esistente.	200 OK: Dati aggiornati
`PUT`	Istanza non trovata.	404 Not Found: {`"error": "Istanza non trovata"`}
`PUT`	Errore nella validazione o aggiornamento.	400 Bad Request: {`"error": "..."`}
`DELETE`	Eliminazione riuscita.	204 No Content
`DELETE`	Errore durante l'eliminazione.	400 Bad Request: {`"error": "..."`}

Tabella 1: Comportamento dettagliato dei metodi HTTP nella vista `AbstractView`

4.4.4.2 BlockView

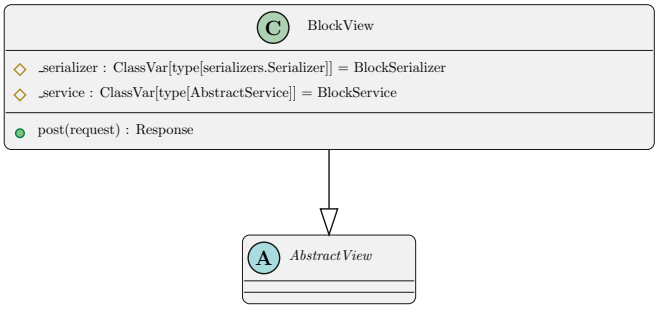

Figura 21: Diagramma UML della classe BlockView.

Descrizione

La classe BlockView deriva da AbstractView e ridefinisce la logica di controllo della creazione di un blocco.

Rotte API

- `question_blocks/`
- `question_blocks/<int:instance_id>/`

Dipendenze

- BlockSerializer;
- BlockService.

Risposte possibili

Metodo	Descrizione	Risposta
`POST`	Crea un nuovo blocco a partire da nome e lista di domande.	201 Created: Dati serializzati
`POST`	Nome già esistente nel database.	500 Internal Server Error: {`"error": "Nome duplicato"`}
`POST`	Errore imprevisto durante la creazione del blocco.	400 Bad Request: {`"error": "..."`}

Tabella 2: Comportamento dettagliato dei metodi HTTP nella vista `BlockView`

4.4.4.3 BlockTestView

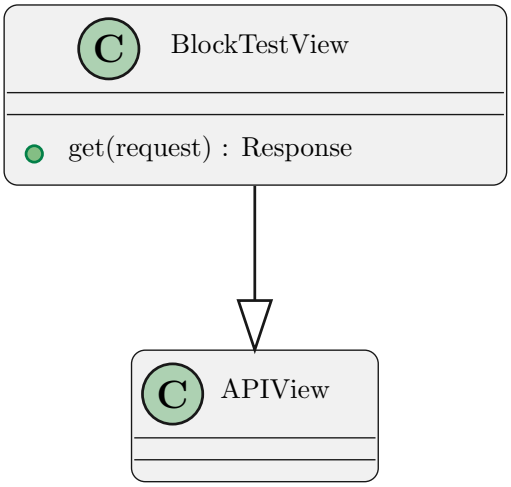

Figura 22: Diagramma UML della classe BlockTestView.

Descrizione

La classe BlockTestView deriva da APIView e definisce la logica di fetch dei dati necessari per eseguire il confronto tra due LLM sulla base dei blocchi in comune.

Rotte API

- `question_blocks/compare/?first_llm_id=&second_llm_id=`

Dipendenze

- BlockRepository;
- RunRepository;
- LLMService.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Calcola e restituisce i blocchi comuni tra due LLM identificati da `first_llm_id` e `second_llm_id`, con medie e valori grezzi di valutazioni semantiche ed esterne.	200 OK: `{"common_blocks": [...]}`
`GET`	Uno o entrambi gli ID dei LLM non sono forniti o non validi.	400 Bad Request: `{"error": "Missing LLM IDs"}`

Tabella 3: Comportamento dettagliato dei metodi HTTP nella vista `BlockTestView`

4.4.4.4 LLMView

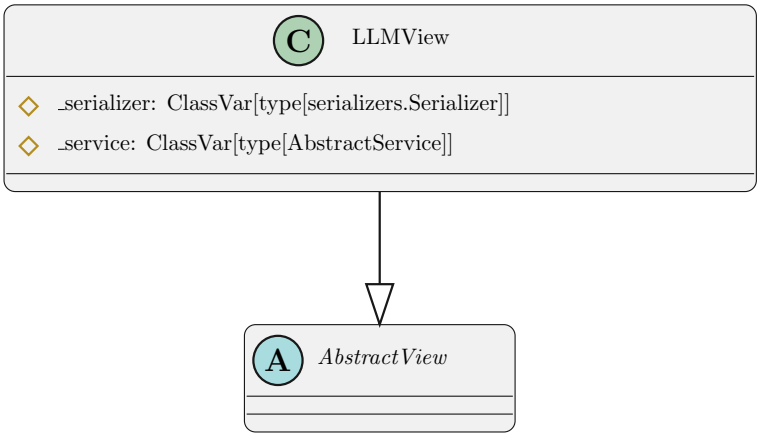

Figura 23: Diagramma UML della classe LLMView.

Descrizione

La classe LLMView deriva da AbstractView, gestisce la logica delle operazioni CRUD sulle istanze dei LLM in database.

Rotte API

- `llm_list/`
- `llm_list/<int:instance_id>/`

Dipendenze

- LLMSerializer;
- LLMService.

4.4.4.5 OllamaView

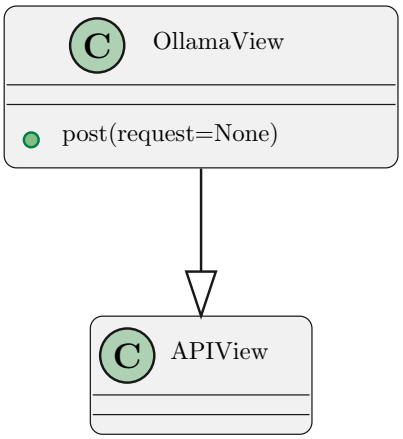

Figura 24: Diagramma UML della classe OllamaView.

Descrizione

La classe OllamaView deriva da APIView, gestisce l'integrazione tra backend primario e secondario per la sincronizzazione dei LLM installati su Ollama.

Rotte API

- `llm_list/load_ollama/`

Dipendenze

- OllamaView.

Risposte possibili

Metodo	Descrizione	Risposta
`POST`	Sincronizza i modelli LLM disponibili su Ollama con il database locale.	200 OK: {`"message": "LLM models loaded successfully from Ollama server"`}
`POST`	Errore durante la sincronizzazione dei modelli.	500 Internal Server Error: {`"error": "..."`}

Tabella 4: Comportamento dettagliato dei metodi HTTP nella vista `OllamaView`

4.4.4.6 PrevTestView

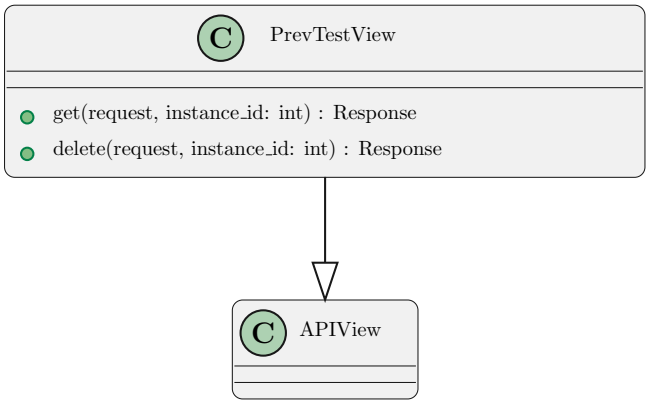

Figura 25: Diagramma UML della classe PrevTestView.

Descrizione

La classe PrevTestView deriva da APIView, gestisce il fetch e la cancellazione dei test precedentemente eseguiti in una sessione.

Rotte API

- `previous_tests/<int:instance_id>/`

Dipendenze

- BlockTestService;
- SessionService.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Recupera tutti i test precedenti di una sessione specifica tramite `instance_id`. Se viene fornito `test_id` come query param, restituisce i dettagli di quel test specifico.	200 OK: dati del test o lista di test serializzati
`GET`	Test non trovato (ad esempio, `test_id` non valido).	404 Not Found: {`"error": "Test not found"`}
`GET`	Errore generico durante il recupero dei test.	400 Bad Request: {`"error": "..."` }
`DELETE`	Elimina un test specifico identificato da `instance_id`.	204 No Content: nessun contenuto
`DELETE`	Test non trovato durante l'eliminazione.	404 Not Found: {`"error": "Test not found"`}
`DELETE`	Errore generico durante l'eliminazione.	400 Bad Request: {`"error": "..."` }

Tabella 5: Comportamento dettagliato dei metodi HTTP nella vista `PrevTestView`

4.4.4.7 PromptView

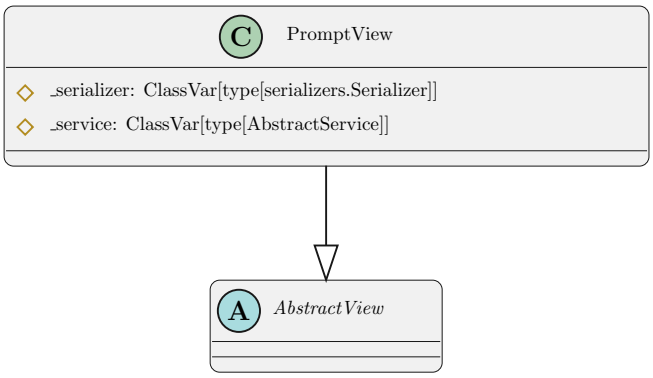

Figura 26: Diagramma UML della classe PromptView.

Descrizione

La classe PromptView deriva da AbstractView, gestisce la logica delle operazioni CRUD sulle istanze dei Prompt in database.

Rotte API

- `prompt_list/`
- `prompt_list/<int:instance_id>/`

Dipendenze

- PromptSerializer;
- PromptService.

4.4.4.8 RunPromptView

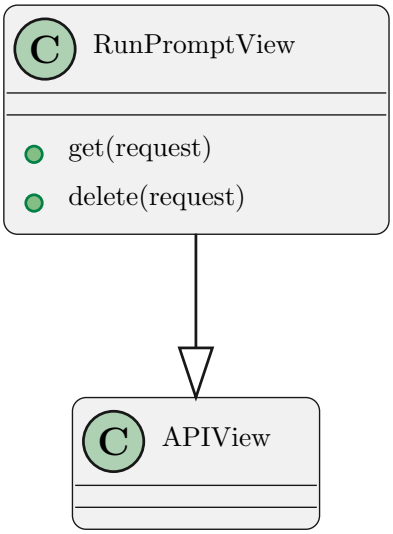

Figura 27: Diagramma UML della classe RunPromptView.

Descrizione

La classe RunPromptView deriva da APIView, gestisce la logica delle operazioni di fetch dei dati di un prompt e di cancellazione delle run.

Rotte API

- `prompt_runs/`

Dipendenze

- RunService.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Recupera i dati formattati di un prompt specificato tramite `prompt_id` passato come query parameter.	200 OK: dati del prompt formattati
`GET`	Parametro `prompt_id` mancante nella richiesta.	400 Bad Request: `{"error": "Missing prompt_id."}`
`GET`	`prompt_id` non valido (non convertibile in intero).	400 Bad Request: `{"error": "Invalid prompt_id."}`
`GET`	Prompt non trovato con l'`prompt_id` fornito.	404 Not Found: `{"error": "Prompt not found."}`
`DELETE`	Elimina un run specificato tramite `run_id` passato come query parameter.	204 No Content: nessun contenuto
`DELETE`	Parametro `run_id` mancante nella richiesta.	400 Bad Request: `{"error": "Missing run_id."}`
`DELETE`	`run_id` non valido (non convertibile in intero).	400 Bad Request: `{"error": "Invalid run_id."}`
`DELETE`	Run non trovato durante l'eliminazione, con messaggio di errore specifico.	404 Not Found: `{"error": "..."}`
`DELETE`	Errore generico durante l'eliminazione.	400 Bad Request: `{"error": "..." }`

Tabella 6: Comportamento dettagliato dei metodi HTTP nella vista `RunPromptView`

4.4.4.9 SessionsView

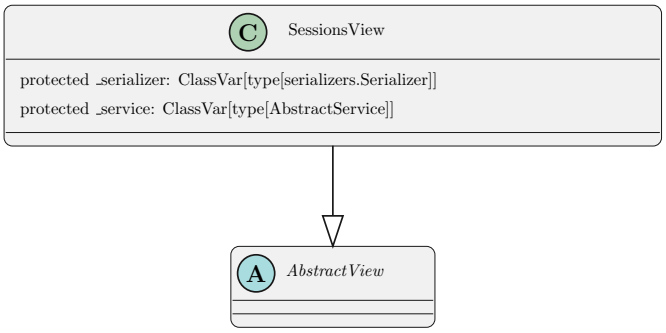

Figura 28: Diagramma UML della classe SessionsView.

Descrizione

La classe SessionsView deriva da AbstractView, gestisce la logica delle operazioni CRUD sulle istanze di Session in database.

Rotte API

- `session_list/`
- `session_list/<int:instance_id>/`

Dipendenze

- SessionSerializer;
- SessionService.

4.4.4.10 RunBlockTestView

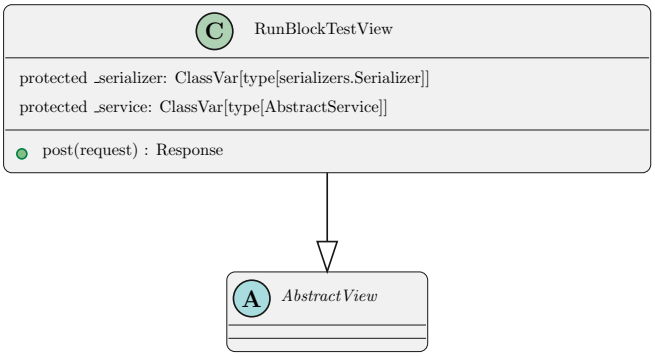

Figura 29: Diagramma UML della classe RunBlockTestView.

Descrizione

La classe SessionsView deriva da AbstractView, gestisce la logica delle operazioni CRUD sulle istanze di BlockTest in database. Ridefinisce il metodo per la creazione di un test, chiamando il metodo corretto del servizio assegnato.

Rotte API

- `runtest/`

Dipendenze

- BlockTestSerializer;
- BlockTestService.

Risposte possibili

Metodo	Descrizione	Risposta
`POST`	Esegue un test con i blocchi forniti nella richiesta e la sessione specificata tramite `sessionId` nel payload JSON.	200 OK: dati del test eseguito
`POST`	Errore di connessione durante l'esecuzione del test.	503 Service Unavailable: `{"error": "..."}`
`POST`	File non trovato durante l'esecuzione del test.	500 Internal Server Error: `{"error": "..."}`
`POST`	Errore inatteso durante l'esecuzione del test.	500 Internal Server Error: `{"error": "Errore inatteso: ..."}`

Tabella 7: Comportamento dettagliato dei metodi HTTP nella vista `RunBlockTestView`

4.4.4.11 SessionLLMView

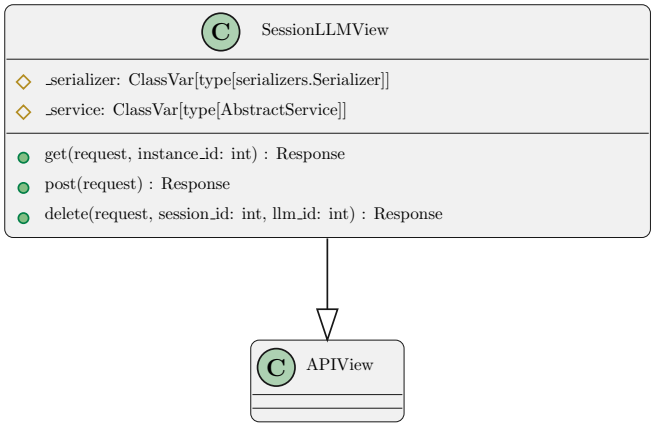

Figura 30: Diagramma UML della classe SessionLLMView.

Descrizione

La classe SessionLLMView deriva da APIView, gestisce la logica di fetch, creazione ed eliminazione dei collegamenti tra Session ed LLM in database.

Rotte API

- `llm_list_by_session/<int:instance_id>/`
- `llm_add/`
- `llm_remaining/<int:instance_id>`
- `llm_delete/<int:session_id>/<int:llm_id>`

Dipendenze

- LLMSerializer;
- SessionService.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Restituisce tutti i modelli LLM esclusi da una sessione specificata tramite `instance_id`.	200 OK: lista dei modelli LLM esclusi
`GET`	Sessione non trovata.	404 Not Found: `{"error": "Session not found"}`
`GET`	Nessun LLM trovato associato.	404 Not Found: `{"error": "LLM not found"}`
`GET`	Errore generico lato server.	500 Internal Server Error: `{"error": "..."}`
`POST`	Collega un LLM a una sessione. Richiede `sessionId` e `llmId` nel corpo della richiesta.	201 Created: oggetto LLM collegato serializzato
`POST`	Sessione non trovata.	404 Not Found: `{"error": "Session not found"}`
`POST`	LLM non trovato.	404 Not Found: `{"error": "LLM not found"}`
`POST`	Errore generico lato server.	500 Internal Server Error: `{"error": "..."}`
`DELETE`	Rimuove un LLM da una sessione, specificando `session_id` e `llm_id` nell'URL.	204 No Content
`DELETE`	Sessione non trovata.	404 Not Found: `{"error": "Session not found"}`
`DELETE`	LLM non trovato.	404 Not Found: `{"error": "LLM not found"}`
`DELETE`	Errore generico lato server.	500 Internal Server Error: `{"error": "..."}`

Tabella 8: Comportamento dettagliato dei metodi HTTP nella vista `SessionLLMView`

4.4.4.12 LLMServiceView

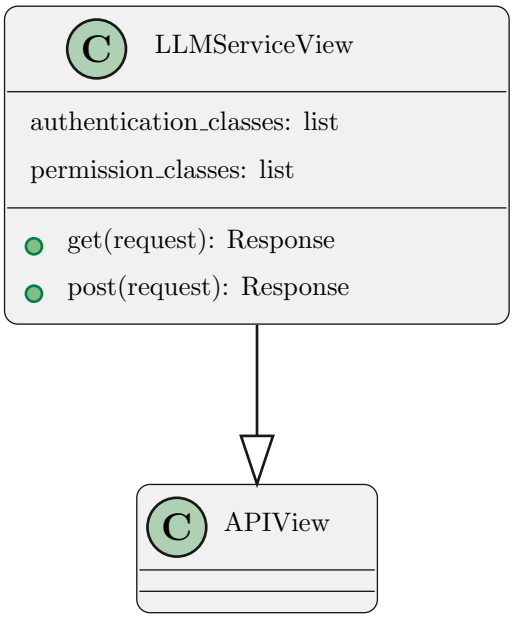

Figura 31: Diagramma UML della classe LLMServiceView.

Descrizione

Questa vista appartenente al microservizio per modelli LLM gestisce le operazioni di fetch ed interrogazione dei modelli installati su Ollama.

Rotte API

- `interrogate/`

Dipendenze

- OllamaLLMIntegrationService;
- LLMRequestSerializer;
- LLMResponseSerializer.

Risposte possibili

Metodo	Descrizione	Risposta
`GET`	Restituisce la lista dei modelli LLM disponibili tramite il servizio Ollama.	200 OK: lista dei modelli LLM disponibili
`GET`	Errore di connessione con Ollama durante il recupero dei modelli.	500 Internal Server Error: {`"error": "Connessione con Ollama fallita"`}
`POST`	Interroga un modello LLM specifico con un prompt, inviando il nome LLM e il prompt nel body della richiesta.	200 OK: risposta con `llm_name`, `prompt` e `answer`
`POST`	Validazione dei dati di input fallita (nome LLM o prompt mancanti o non validi).	400 Bad Request: errori di validazione
`POST`	Errore di connessione con Ollama durante l'interrogazione del modello.	500 Internal Server Error: {`"error": "Connessione con Ollama fallita"`}

Tabella 9: Comportamento dettagliato dei metodi HTTP nella vista `LLMServiceView`

4.5 Architettura di deployment

L'architettura di deployment$_G$ adottata per il progetto$_G$ si basa sulla containerizzazione attraverso Docker Compose$_G$, al fine di garantire una gestione semplificata e una configurazione coerente degli ambienti. Tale approccio risponde alla necessità di orchestrare servizi differenti con esigenze specifiche, mantenendo al contempo una facile scalabilità e una chiara separazione delle responsabilità. In particolare, l'ambiente Docker$_G$ definito dal file docker-compose.yml comprende i seguenti servizi e relative immagini Docker utilizzate:

- **Server (backend principale)**: basato su Django, espone tramite Daphne$_G$ una API ASGI$_G$ sulla porta 8000. Questo servizio implementa la logica applicativa principale e dipende direttamente da un database MySQL, assicurando l'integrità e la persistenza dei dati. Utilizza l'immagine python.
- **MySQL**: gestisce l'archiviazione persistente dei dati, configurato per essere accessibile tramite la rete interna Docker e garantendo la persistenza tramite volumi dedicati. Utilizza l'immagine ufficiale mysql.
- **LLM Service**: servizio Django indipendente dedicato alla gestione dei flussi LangChain. Esposto sulla porta 8001 per la comunicazione interna. Utilizza l'immagine python.
- **Ollama**: container$_G$ specializzato per l'inferenza LLM, basato sull'immagine ufficiale ollama. Viene esposto internamente tramite porta 11434 e mappato esternamente su porta 5000.

- **Frontend**: applicazione client sviluppata con React e Next.js, la quale viene compilata e avviata tramite un server Node.js integrato. Il deployment effettivo avviene sfruttando un container basato sull'immagine ufficiale node, che esegue i comandi di build (npm run build) e start (next start), esponendo il servizio sulla porta 3000. L'applicazione frontend comunica direttamente con il backend tramite API REST.

4.6 Design patterns utilizzati

4.6.1 Layered Architecture

Il pattern architetturale a livelli (Layered Architecture) suddivide l'applicativo in strati distinti, ciascuno con responsabilità ben definite, al fine di migliorare la manutenibilità$_G$, la testabilità e la scalabilità del sistema.

4.6.1.1 Livello Applicazione (API Layer)

Definizione:
Il livello applicazione espone gli endpoint REST, riceve e valida le richieste HTTP e restituisce risposte JSON.

Implementazione:
Classe base: AbstractView (estende APIView di Django REST Framework) che definisce il flusso CRUD standard:

- Riceve la request HTTP e la passa al serializer per la validazione;
- In caso di dati validi, delega al `service` la logica di business (create/read/update/delete);
- Utilizza il serializer per (de)serializzare i dati di input/output;
- Restituisce un oggetto `Response` con JSON e codice HTTP appropriato.

View concrete: ereditano da AbstractView e dichiarano solo:

- il **serializer** specifico per la vista;
- Il **service** che implementa la logica di business.

Alcune viste, tuttavia, non ereditano da AbstractView quando richiedono implementazioni personalizzate, non completamente aderenti al flusso CRUD standard, oppure quando coinvolgono operazioni specializzate.

Esempio di implementazione concreta:

```
# urls.py
from .views_def.block_view import BlockView

urlpatterns = [
    # POST/GET
    path("question_blocks/", BlockView.as_view()),

    # GET/PUT/DELETE
    path("question_blocks/<int:id>/", BlockView.as_view()),
]

# API/views_def/block_view.py
class BlockView(AbstractView):
    # Assegna un serializzatore alla vista
        _serializer: ClassVar[type[serializers.Serializer]] = BlockSerializer
    # Assegna un servizio alla vista
        _service: ClassVar[type[AbstractService]] = BlockService

    def post(self, request) -> Response:
        # 1. estrai i dati dal JSON
         data: dict = {
            "name": request.data.get("name"),
            "questions": request.data.get("questions"),
        }

        # 2. crea il blocco via il layer di business
        try:
            result = self._service.create(data=data)

            # 3. affida al layer di business la logica di dominio
            if result == False:
                return Response(
                    {"error": "Nome duplicato"},
                    status=status.HTTP_500_INTERNAL_SERVER_ERROR,
                )
             # 3. serializza l'oggetto risultante in JSON
            serialized = self._serializer(result)
            # 4. ritorna il JSON al client con codice adeguato
            return Response(serialized.data, status=status.HTTP_201_CREATED)
        except Exception as e:
            return Response({"error": str(e)}, status=status.HTTP_400_BAD_REQUEST)
```

Motivazioni

- **Indipendenza**: gestisce solo le richieste HTTP, senza includere logica di dominio o accesso al database, delegando i dettagli di persistenza e validazione avanzata ai layer sottostanti.
- **Scalabilità**: per aggiungere un nuovo endpoint, è sufficiente creare una sottoclasse di `AbstractView`.

4.6.1.2 Livello Dominio

Definizione:
Il Livello Dominio centralizza le regole di dominio, applica validazioni complesse e orchestra operazioni atomiche su più sorgenti dati, garantendo coerenza.

Implementazione:
Classe base: `AbstractService` (in `API/services/abstract_service.py`) fornisce un'implementazione generica del flusso CRUD:

- Riceve i dati dalla *Presentation Layer*;
- Invoca il repository associato (`self.repository`) per operazioni di persistenza (create/read/update/delete);
- Gestisce eccezioni e rollback di transazione se necessario;
- Restituisce oggetti Python (model instances) già validati, pronti per la serializzazione.

Service concreti:
Estendono `AbstractService` e dichiarano soltanto:

- `repository`: il repository specifico (classe derivata da `AbstractRepository`) per il modello di dominio;
- Eventuali metodi custom di dominio per operazioni non-standard.

Esempio di implementazione concreta:

```
class BlockService(AbstractService):

    _repository: ClassVar[AbstractRepository] = BlockRepository

    @classmethod
    def create(cls, data: dict) -> Block | bool:

            # 1. Verifica che non esista un Block con lo stesso name
        duplicate = BlockRepository.get_by_name(data["name"])
        if duplicate == None:

            # 2. Crea il Block principale
         new_block = BlockRepository.create({"name": data["name"]})

         # 3. Per ogni domanda, vengono coordinati due repository:
         #    - PromptRepository: crea o recupera il Prompt
         #    - BlockRepository: associa il Prompt al Block
         for prompt in data["questions"]:
             instance = PromptRepository.get_or_create(
                 prompt_text=prompt["question"], expected_answer=prompt["answer"]
             )
             BlockRepository.add_prompt(block=new_block, prompt=instance)
             return new_block
         return False
```

Motivazioni

- **Incapsulamento delle regole di dominio**: mantiene tutta la logica di validazione e le policy applicative in un'unica posizione.
- **Orchestrazione di operazioni complesse**: coordina chiamate multiple a diversi repository in un'unica transazione atomica.

4.6.1.3 Livello Persistenza

Definizione:
Questo strato si occupa unicamente di persistenza: leggere e scrivere dati nel database tramite l'ORM di Django, senza alcuna logica di dominio o regole di business. **Implementazione:**

Classe base: AbstractRepository (in API/repositories/abstract_repository.py) che implementa i metodi CRUD generici:

- `get_all()`: Recupera tutte le istanze del modello dal database.
- `get_by_id(id)`: Recupera una singola istanza del modello tramite il suo identificativo.
- `create(data)`: Crea una nuova istanza del modello con i dati forniti.
- `update(id, data)`: Aggiorna un'istanza esistente del modello con i dati forniti.
- `delete(id)`: Elimina un'istanza del modello tramite il suo identificativo.

Repository concreti: ereditano da AbstractRepository e dichiarano solo:

- `model`: Specifica il modello ORM associato al repository.
- Query specializzate e operazioni di utilità: Metodi personalizzati per esigenze specifiche, come filtri avanzati o operazioni batch.

Esempio di implementazione concreta:

```
class BlockRepository(AbstractRepository):
    _model: ClassVar[models.Model] = Block

    @staticmethod
    def add_prompt(block: Block, prompt: Prompt) -> Block:

        block.prompt.add(prompt)
        return block

    @classmethod
    def get_common_blocks_for_llms(cls, first_llm: LLM, second_llm: LLM) -> List[Block]:
        return (
            cls._model.objects.filter(
                Q(prompt__run__llm=first_llm) | Q(prompt__run__llm=second_llm)
            )
            .annotate(llm_count=Count("prompt__run__llm", distinct=True))
            .filter(llm_count=2)
            .distinct()
        )
```

Motivazioni

- **Single Responsibility**: gestisce solo operazioni ORM, senza regole di business.
- **Riutilizzabile**: ogni repository espone un'interfaccia chiara per lavorare su un singolo modello.
- **Testabile**: può essere mockato o sostituito con versioni in-memory per i test.