# Specifica Tecnica

versione 1.0.0

7commits@gmail.com

**Progetto di Ingegneria del Software**

A.A. 2024/2025

| Destinatari | Responsabile | Redattori | Verificatori |
|---|---|---|---|
| Prof. Tullio Vardanega | Marco Cola | Stefano Dal Poz | Giulia Hu |
| Prof. Riccardo Cardin | | Marco Cola | Ruize Lin |
| Gruppo 7Commits | | Ruize Lin | Giada Rossi |
| | | Mattia Piva | Marco Cola |

# Registro delle modifiche

| Versione | Data | Autori | Verificatori | Descrizione |
|---|---|---|---|---|
| v1.0.0 | 2025-08-28 | Marco Cola | Ruize Lin | Approvazione per PB |
| v0.2.1 | 2025-08-27 | Ruize Lin | Marco Cola | Aggiunto Diagramma delle classi in 3.4 |
| v0.2.0 | 2025-08-26 | Marco Cola | Ruize Lin | Rimossa sezione 3.3.1.1 sul modello esagonale in seguito all'incontro con il Prof. Cardin |
| v0.1.2 | 2025-08-21 | Marco Cola | Ruize Lin, Giulia Hu | Uniformato il documento all'uso corretto di inline code (`es`) invece del corsivo (*es*) per riferimenti a codice. |
| v0.1.1 | 2025-08-20 | Marco Cola | Ruize Lin | Aggiunti riferimenti al Glossario v2.0.0 |
| v0.1.0 | 2025-08-19 | Marco Cola | Ruize Lin | Revisione della grammatica sezione «Tecnologie» |
| v0.0.15 | 2025-08-18 | Marco Cola | Ruize Lin, Mattia Piva | Revisione della grammatica sezione «Architettura» |
| v0.0.14 | 2025-08-16 | Marco Cola | Ruize Lin | Revisione della grammatica sezioni «Flusso di dati e Interazione API» e «Stato dei requisiti funzionali» |
| v0.0.13 | 2025-08-15 | Marco Cola | Ruize Lin | Corretta spaziatura tra sezioni del documento |
| v0.0.12 | 2025-08-14 | Mattia Piva | Marco Cola, Ruize Lin | Revisione struttura e grammatica delle sezioni «Funzionalità principali» e «Moduli e Componenti» |
| v0.0.11 | 2025-08-12 | Ruize Lin | Mattia Piva | Aggiunta e integrata la sezione «Architettura» |
| v0.0.10 | 2025-08-12 | Ruize Lin | Mattia Piva | Aggiunta e integrata la sezione «Flusso di Dati e Interazione API» |
| v0.0.9 | 2025-08-11 | Ruize Lin | Mattia Piva | Terminata sezione «Moduli e Componenti» |
| v0.0.8 | 2025-08-10 | Ruize Lin | Mattia Piva | Terminata sezione «Funzionalità principali» |
| v0.0.7 | 2025-08-07 | Giada Rossi | Mattia Piva, Marco Cola, Ruize Lin | Corretto e integrato paragrafo «Tecnologie per l'analisi del codice» |
| v0.0.6 | 2025-08-05 | Giada Rossi | Marco Cola, Ruize Lin | Integrate e corrette sezioni «Introduzione», «Tecnologie», «Funzionalità principali» e «Database» |

| | | | | |
|---|---|---|---|---|
| v0.0.5 | 2025-08-04 | Giada Rossi | Marco Cola, Ruize Lin | Aggiunto e strutturato il paragrafo «Database» |
| v0.0.4 | 2025-08-03 | Giada Rossi | Marco Cola, Ruize Lin | Aggiunto e strutturato il paragrafo «Funzionalità principali» |
| v0.0.3 | 2025-08-01 | Giada Rossi | Marco Cola, Ruize Lin | Aggiunto e integrato paragrafo «Tecnologie» |
| v0.0.2 | 2025-07-28 | Marco Cola | Mattia Piva, Ruize Lin | Aggiunta la sezione «Stato dei requisiti funzionali» |
| v0.0.1 | 2025-07-26 | Marco Cola | Ruize Lin | Prima Bozza |

# Indice

# 1. Introduzione

Questo documento descrive la *Specifica Tecnica*$_G$ del progetto Artificial QI, una *WebApp*$_G$ sviluppata per valutare l'affidabilità delle risposte generate da un *Large Language Model*$_G$ (LLM) rispetto ad un *dataset*$_G$ di domande. Vengono illustrate e motivate le scelte architetturali, i *design pattern*$_G$ e le tecnologie adottate dal team di sviluppo. La trattazione comprende l'*architettura logica*$_G$ e di *deployment*$_G$, la progettazione ad alto livello e di dettaglio dei principali componenti, supportata da diagrammi esplicativi volti a chiarire la struttura del sistema.

## 1.1. Glossario

All'interno della documentazione del progetto è disponibile il *Glossario v2.0.0*, documento che fornisce una definizione chiara e dettagliata di tutti i termini specifici utilizzati all'interno di questo documento. Tali termini sono contrassegnati da una «G» a pedice per facilitarne l'individuazione e il riferimento, in *questo modo*$_G$.

## 1.2. Riferimenti

### 1.2.1. Riferimenti normativi

- Norme_di_Progetto_v2.0.0: https://7commits.github.io/7Commits/docs/PB/Interni/Norme_di_Progetto_v2.0.0.pdf
- **Regolamento del progetto didattico** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/PD1.pdf
- **Capitolato C1** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~tullio/IS-1/2024/Progetto/C1.pdf

### 1.2.2. Riferimenti informativi

- Analisi dei requisiti v2.0.0: https://7commits.github.io/7Commits/docs/PB/Esterni/Analisi_dei_Requisiti_v2.0.0.pdf
- **Verbali Interni**
- **Verbali Esterni**
- **Progettazione e programmazione: Diagrammi delle classi (UML)** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~rcardin/swea/2023/Diagrammi%20delle%20Classi.pdf
- **Progettazione: I pattern architetturali** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~rcardin/swea/2022/Software%20Architecture%20Patterns.pdf
- **Progettazione: I pattern creazionali** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Creazionali.pdf
- **Progettazione: I pattern strutturali** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~rcardin/swea/2022/Design%20Pattern%20Strutturali.pdf
- **Progettazione: I pattern comportamentali** (ultimo accesso: 2025-08-28): https://drive.google.com/file/d/1cpi6rORMxFtC91nI6_sPrG1Xn-28z8eI/view
- **Progettazione software** (ultimo accesso: 2025-08-28): https://www.math.unipd.it/~tullio/IS-1/2024/Dispense/T06.pdf

# 2. Tecnologie

In questa sezione sono indicate tutte le tecnologie utilizzate dal gruppo nello sviluppo dell'applicazione.

## 2.1. Tecnologie per la codifica

### 2.1.1. Linguaggi

| Nome | Versione | Descrizione |
|---|---|---|
| *Python*$_G$ | >= 3.10.0 | Linguaggio di programmazione ad alto livello ampiamente utilizzato nelle applicazioni Web, nello sviluppo di software, nella *data science*$_G$ e nel *machine learning*$_G$ (ML). È caratterizzato da una sintassi chiara e leggibile e dispone di molte risorse e librerie online. |

### 2.1.2. Librerie python e framework

| Nome | Versione | Descrizione |
|---|---|---|
| *Streamlit*$_G$ | >= 1.28.0 | Framework Python *open source*$_G$ che permette di creare e condividere rapidamente applicazioni web interattive, *dashboard*$_G$ interattive, visualizzazione di dati e applicazioni di *machine learning*$_G$. Caratterizzato da semplicità d'uso e rapidità di sviluppo |
| *Pandas*$_G$ | >= 1.5.0 | Libreria Python che offre strutture dati e operazioni per manipolare e analizzare dati. |
| *Plotly*$_G$ | >= 5.0.0 | Libreria Python *open source*$_G$ per creare grafici interattivi e visualizzazioni di dati, sia in ambito web che in applicazioni di *data science*$_G$. Offre facilità d'uso e flessibilità. |
| *SQLAlchemy*$_G$ | >= 2.0.0 | Libreria Python che permette agli sviluppatori di interagire con i *database relazionali*$_G$ utilizzando il linguaggio Python, anziché scrivere direttamente *query*$_G$ *SQL*$_G$. |
| *PyMySQL*$_G$ | >= 1.0.0 | Libreria Python che consente ai programmi Python di comunicare con un *server*$_G$ *MySQL*$_G$ per eseguire operazioni come la creazione, la lettura, l'aggiornamento e l'eliminazione di dati. |
| *OpenAI*$_G$ | >= 1.0.0 | Libreria Python che permette di fare richieste a tutti gli *host*$_G$ che supportano lo standard di *OpenAI*$_G$ |
| *Cryptography*$_G$ | >= 42.0.0 | Libreria Python di crittografia per far eseguire *MySQL*$_G$ 8.0 in modo sicuro e stabile per migliorare la sicurezza senza causare problemi di esecuzione o performance. |

### 2.1.3. Strumenti

| Nome | Versione | Descrizione |
|---|---|---|
| *MySQL*$_G$ | >= 5.7.0 | Sistema di gestione di *database relazionali*$_G$ *open source*$_G$. Consente di archiviare i dati (domande/risposte e risultati di test e report) in tabelle, organizzate in righe e colonne, e di interagire con essi tramite il linguaggio SQL. |
| *Docker*$_G$ | >= 20.10.0 | Strumento di conteinerizzazione in cui ogni applicazione viene gestita ed eseguita come un container isolato dagli altri, consentendo |

| | | |
|---|---|---|
| | | loro di funzionare in modo coerente su qualsiasi ambiente, indipendentemente dal sistema operativo o dalle librerie sottostanti. |
| *Docker Compose*G | >= 2.0.0 | Strumento di conteinerizzazione per gestire in modo immediato i container. |
| *Git*G | | Sistema di controllo versione distribuito, essenziale per la gestione e il tracciamento delle modifiche nel codice sorgente di progetti software. |

## 2.2. Tecnologie per il testing

| Nome | Versione | Descrizione |
|---|---|---|
| *pytest*G | >= 7.0.0 | *Framework*G di test per Python |
| *pytest-cov*G | >= 4.0.0 | *Plugin*G per misurare la copertura del codice |
| *pytest-mock*G | >=3.14.1 | Plugin per la creazione di *mock*G nei test |

## 2.3. Tecnologie per l'analisi del codice

### 2.3.1. Analisi statica

L'*analisi statica*G si effettua esaminando il codice senza eseguirlo per individuare *bug*G, violazioni di stile, problemi di tipizzazione o potenziali vulnerabilità.

| Nome | Versione | Descrizione |
|---|---|---|
| *Flake8*G | >=7.3.0 | Strumento a riga di comando per controllare che il codice segua le linee guida di stile (*convenzioni PEP 8*G) e individuare potenziali problemi di codice ed errori, come quelli di sintassi del codice. |
| *MyPy*G | >=1.17.1 | Strumento di controllo statico che analizza il codice Python e ne verifica la correttezza dei tipi, segnalando eventuali incongruenze tra i tipi dichiarati e quelli effettivi (ad esempio, il tentativo di sommare una stringa e un intero). |

### 2.3.2. Analisi dinamica

L'*analisi dinamica*G consiste nell'esecuzione del codice per osservare il comportamento a runtime, verificare che funzioni correttamente, misurarne la copertura o le prestazioni.

| Nome | Versione | Descrizione |
|---|---|---|
| pytest | >= 7.0.0 | Framework di test per Python che semplifica la scrittura, l'esecuzione e la gestione dei test. I test coprono diverse funzioni di logica, come il calcolo delle statistiche sui risultati e l'importazione di set di domande. |

# 3. Architettura

Il progetto non è stato suddiviso in due insiemi di codice distinti *"Client - Server"*$_G$. È stato invece realizzato come un'applicazione monolitica sviluppata con *Streamlit*$_G$, in cui interfaccia utente, *logica di business*$_G$ e livello dati sono integrati all'interno di un unico progetto *Python*$_G$. Questa scelta è stata motivata dalla natura relativamente semplice dell'applicazione, che non richiedeva un'architettura distribuita. Per tale ragione si è preferito adottare un approccio monolitico, implementando tutte le componenti in un unico blocco tramite Streamlit.

## 3.1. Architettura generale

L'applicazione adotta un'architettura multi-layer organizzata in più componenti principali:

- **Interfaccia utente**;
- **Logica di controllo**;
- **Strato dati** (modello + database);
- **Integrazione con servizi esterni** (*API*$_G$ LLM).

Di seguito, una panoramica di ciascun livello e delle interazioni:

- **Interfaccia Utente (*UI*$_G$)**: Realizzata con *Streamlit*$_G$, offre pagine web interattive per ogni funzionalità (configurazione *API*$_G$, gestione domande, gestione set, esecuzione test, visualizzazione risultati). L'utente interagisce tramite form e pulsanti; ogni azione dell'utente (ad es. aggiungere una domanda o lanciare un test) attiva funzioni *Python*$_G$ lato server.
- **Controller (Logica Applicativa)**: Un insieme di moduli Python che fungono da tramite tra la *UI*$_G$ e il *database*$_G$. I controller implementano le operazioni chiave del sistema (es. creare un preset API, salvare una domanda, eseguire un test) orchestrando chiamate ai modelli e ad altre utilità. In pratica, quando l'utente compie un'azione sulla *UI*$_G$, la funzione Streamlit associata chiama una funzione di controller che incapsula la logica richiesta.
- **Modelli e Database**: I dati persistenti (domande, set di domande, risultati di test, preset API) sono gestiti tramite *SQLAlchemy*$_G$ e salvati in un database *MySQL*$_G$. Il progetto definisce modelli *ORM*$_G$ (Object-Relational Mapping) che mappano le entità principali a tabelle del database. Le entità chiave includono:
  - **Domanda (QuestionORM)**: rappresenta una domanda di test con testo, risposta attesa e categoria.
  - **Set di Domande (QuestionSetORM)**: gruppo logico di più domande identificato da un nome. C'è una relazione molti-a-molti tra set e domande, realizzata tramite una tabella di associazione (`question_set_questions`).
  - **Risultato di Test (TestResultORM)**: memorizza l'esito di un singolo test eseguito su un set, includendo un timestamp e un campo JSON contenente i dettagli (risposte generate, valutazioni e metriche aggregate).
  - **Preset API (APIPresetORM)**: conserva i parametri di configurazione per l'accesso a un provider LLM (nome del preset, tipo di provider, endpoint URL, chiave API, modello selezionato, temperatura e max token).
- **Integrazione LLM (API esterne)**: La piattaforma interagisce con servizi esterni di *AI*$_G$ (come OpenAI) per generare e valutare risposte. Questo avviene tramite chiamate *API*$_G$ *HTTP*$_G$ effettuate con la *libreria OpenAI*$_G$ Python. Le credenziali e gli *endpoint*$_G$ per questi servizi sono gestiti dall'utente tramite i *preset*$_G$ API e utilizzati internamente dai controller al momento opportuno.

#### 3.1.1. Flusso di controllo tipico

Quando l'utente richiede una certa funzionalità, ad esempio l'esecuzione di un test, la chiamata parte dal livello *UI*$_G$ (funzione Streamlit della pagina Esecuzione Test) che invoca la routine appropriata nel controller (ad es. `test_controller.run_test`). Il controller recupera i dati necessari dallo strato modelli/database (es. le domande del set selezionato, i parametri del modello LLM da usare) e utilizza servizi esterni se necessario (chiamate OpenAI per generare e valutare risposte). I risultati vengono quindi salvati nel database e restituiti alla UI, che li presenta sotto forma di grafici e tabelle. Per migliorare le prestazioni, il sistema fa largo uso di *caching in-memory*$_G$: le liste di domande, set, preset e risultati sono caricate dal database in strutture *pandas*$_G$ DataFrame e memorizzate in *cache LRU*$_G$ sul lato *server*$_G$. In questo modo, successive richieste di dati (ad esempio la visualizzazione di tutte le domande) possono essere soddisfatte senza interrogare nuovamente il database, a meno che non siano state apportate delle modifiche.

## 3.2. Deployment

Dal punto di vista del *deployment*$_G$, l'architettura è containerizzata: viene utilizzato un container *Docker*$_G$ per l'app Streamlit e un container per il database *MySQL*$_G$, orchestrati con *Docker Compose*$_G$. Il file `docker-compose.yml` definisce i due servizi (app e database) e garantisce che il database sia disponibile all'avvio dell'applicazione. All'avvio, la logica di inizializzazione (in `utils.startup_utils` e `models.database`) si occupa di creare il database e le tabelle se non già presenti, utilizzando le credenziali fornite nel file di configurazione (`database.config`). Ciò assicura che l'ambiente sia pronto al primo utilizzo senza dover creare manualmente lo schema del database.

```
1   database:
2     image: mysql:8.0
3     container_name: database
4     volumes:
5       - database_data:/var/lib/mysql
6     environment:
7       MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'
8       MYSQL_ROOT_HOST: '%'
9     networks:
10      - llm-network
11
12  app:
13    build: .
14    container_name: llm-app
15    ports:
16      - '8501:8501'
17    volumes:
18      - .:/app
19    depends_on:
20      - database
21    command: streamlit run app.py --server.port 8501 --server.address 0.0.0.0
22    networks:
23      - llm-network
24
```

## 3.3. Design pattern

Il *design pattern*$_G$ (o pattern di progettazione) è una soluzione collaudata e riutilizzabile a un problema comune di progettazione software. Non è un blocco di codice pronto, ma un modello concettuale che descrive come organizzare classi, oggetti e interazioni per risolvere problemi ricorrenti. Serve a

standardizzare le soluzioni, fornendo un linguaggio comune tra sviluppatori, sfruttando approcci già sperimentati permette di migliorare la manutenibilità e aumentare la leggibilità del codice.

### 3.3.1. Design pattern architetturale - MVC pattern

Il nostro gruppo ha scelto il pattern architetturale *MVC*$_G$. *MVC (Model–View–Controller)*$_G$ «separa livello dati, interfaccia e logica di controllo, rendendo la struttura più chiara e le responsabilità più esplicite:

- **Model**: gestisce e memorizza i dati.
- **View**: presenta l'interfaccia e le interazioni.
- **Controller**: fa da ponte tra Model e View, coordinando il flusso richiesta/risposta.

I principali vantaggi sono:

- Divisione del lavoro chiara, che facilita la collaborazione tra più sviluppatori.
- Alta manutenibilità: le modifiche alla logica non impattano direttamente l'interfaccia.
- Buona estensibilità: è semplice aggiungere nuove viste o sostituire le sorgenti dati.

### 3.3.2. Design pattern creazionale - Singleton pattern

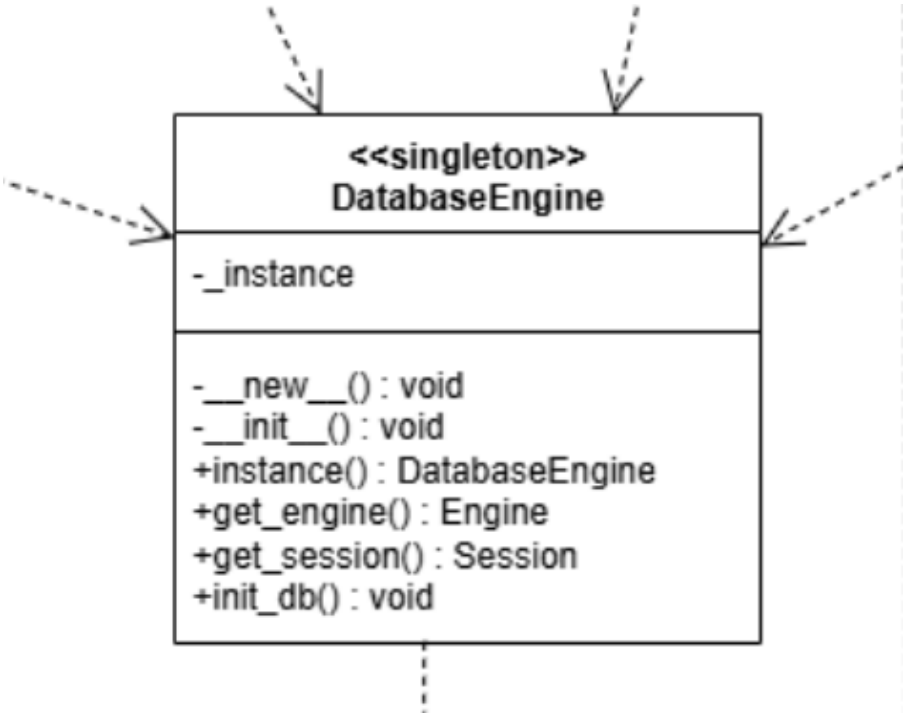


Figura 1: Singleton Pattern

Il pattern *Singleton*$_G$ è un design pattern che impedisce la creazione di più istanze di una classe e fornisce un punto di accesso globale. È particolarmente adatto per componenti che devono essere uniche, come gestori di configurazione, log o connessioni ai *database*$_G$. In ambienti *multi-thread*$_G$, senza sincronizzazione si potrebbero creare istanze multiple; per questo è essenziale un'implementazione thread-safe.

Nel nostro progetto, il database è al centro dell'applicazione e l'engine *SQLAlchemy*$_G$ è un oggetto pesante da istanziare. Avere più istanze di Engine sparpagliate nel codice porterebbe infatti a:

- creazione di più pool di connessioni, con spreco di risorse;
- configurazioni duplicate e potenzialmente incoerenti;

- maggiore complessità nella gestione delle transazioni e del pooling.

Il pattern Singleton consente di centralizzare la creazione e la gestione di Engine e della sessione, assicurando che tutte le parti dell'applicazione lavorino con la stessa connessione al database. Inoltre, abilita il *lazy loading*$_G$, permettendo la creazione dell'Engine solo al momento effettivamente necessario, con una conseguente riduzione dei tempi di avvio e dell'uso di memoria.

#### 3.3.2.1. Realizzazione del pattern, dichiarazione della classe e variabili statiche

La classe `DatabaseEngine` viene definita con un docstring che indica chiaramente che si tratta di un singleton thread-safe:

```python
class DatabaseEngine:
    """Singleton thread-safe che fornisce l'engine del database e le sessioni."""
    _instance = None
    _instance_lock = threading.Lock()
```

- `_instance` è la variabile di classe che conterrà l'unica istanza.
- `_instance_lock` è un threading.Lock usato per sincronizzare l'accesso in ambienti *multi-thread*$_G$.

Per evitare la creazione di nuove istanze mediante chiamata diretta, `__new__` e `__init__` lanciano un'eccezione se un'istanza esiste già:

```python
def __new__(cls, *args, **kwargs):
    if cls._instance is not None:
        raise RuntimeError("DatabaseEngine è un singleton; usa DatabaseEngine.instance()")
    return super().__new__(cls)

def __init__(self) -> None:
    if self.__class__._instance is not None:
        raise RuntimeError("DatabaseEngine è un singleton; usa DatabaseEngine.instance()")
    self._engine: Optional[Engine] = None
    self._session_factory: Optional[sessionmaker] = None
    self._engine_lock = threading.Lock()
    self._session_lock = threading.Lock()

```

Questa scelta forza gli utenti della classe a usare il metodo `instance()`.

La classe dispone anche un metodo `instance()` che implementa il double-checked locking:

- controlla se `_instance` è None, e in tal caso entra nel blocco protetto da `_instance_lock` per creare l'istanza:

```python
@classmethod
def instance(cls) -> "DatabaseEngine":
    if cls._instance is None:
        with cls._instance_lock:
            if cls._instance is None:
                cls._instance = cls()
    return cls._instance

```

In questo modo, in presenza di più thread solo il primo che entra nel blocco crea l'istanza; gli altri ottengono immediatamente quella già creata.

`DatabaseEngine` implementa anche metodi per caricare la configurazione , creare il *database*$_G$ se non esiste e costruire l'engine:

python
```python
def get_engine(self) -> Engine:
    if self._engine is None:
        with self._engine_lock:
            if self._engine is None:
                cfg = self._load_config()
                self._ensure_database(cfg)
                url = (
                    f"mysql+pymysql://{cfg['user']}:{cfg['password']}@"
                    f"{cfg['host']}:{cfg.get('port', 3306)}/{cfg['database']}"
                )
                self._engine = create_engine(
                    url,
                    pool_pre_ping=True,
                    pool_recycle=3600,
                )
    assert self._engine is not None
    return self._engine

```

- `_engine_lock` garantisce che l'engine venga creato solo una volta in modo *thread-safe*$_G$.
- `_load_config()` legge i parametri del database dal file `database.config` o `database.config.example`.

python
```python
    def _load_config(self) -> Mapping[str, str]:
        config = configparser.ConfigParser()
        root = Path(__file__).resolve().parent.parent
        cfg_path = root / "database.config"
        if not cfg_path.exists():
            cfg_path = root / "database.config.example"
        config.read(cfg_path)
        return config["mysql"]

```

- `_ensure_database()` crea il database se necessario.

python
```python
    def _ensure_database(self, cfg: Mapping[str, str]) -> None:
        """Crea il database di destinazione se non esiste già."""
        root_url = (
            f"mysql+pymysql://{cfg['user']}:{cfg['password']}@{cfg['host']}:
{cfg.get('port', 3306)}"
        )
        engine = create_engine(root_url, isolation_level="AUTOCOMMIT")
        try:
            with engine.begin() as conn:
                conn.execute(text(f"CREATE DATABASE IF NOT EXISTS `{cfg['database']}
`"))
        except Exception as exc:
            logger.exception(
                "Impossibile creare il database '%s' sull'host '%s' con l'utente
'%s'",
                cfg.get("database"),
```

```
14                 cfg.get("host"),
15                 cfg.get("user"),
16             )
17             raise RuntimeError(
18                 (
19                     f"Impossibile creare il database '{cfg.get('database')}' "
20                     f"sull'host '{cfg.get('host')}' per l'utente '{cfg.get('user')}'.
   "
21                     "Il server del database potrebbe non essere raggiungibile, le
   credenziali potrebbero essere errate "
22                     "o l'utente potrebbe non avere privilegi sufficienti."
23                 )
24             ) from exc
```

L'engine è memorizzato in `_engine` e riutilizzato per tutte le chiamate successive.

Per ottenere una sessione *SQLAlchemy*[G], il metodo `get_session()` crea un factory sessionmaker e lo riutilizza:

python

```python
def get_session(self) -> Session:
    if self._session_factory is None:
        with self._session_lock:
            if self._session_factory is None:
                engine = self.get_engine()
                self._session_factory = sessionmaker(bind=engine)
    assert self._session_factory is not None
    return self._session_factory()

```

La classe mette a disposizione un metodo `reset_instance()` per reimpostare l'engine quando necessario, ad esempio per consentire ai test di isolare lo stato globale.

python

```python
@classmethod
def reset_instance(cls) -> None:
    """Reimposta l'istanza singleton e svuota le risorse in cache."""
    with cls._instance_lock:
        if cls._instance is not None:
            with cls._instance._engine_lock:
                if cls._instance._engine is not None:
                    cls._instance._engine.dispose()
                    cls._instance._engine = None
            with cls._instance._session_lock:
                cls._instance._session_factory = None
        cls._instance = None


```

Questa funzione chiude l'engine e azzera l'istanza, così ogni test inizia con uno stato pulito.

Grazie a questo approccio, tutte le parti dell'applicazione condividono lo stesso engine e la stessa session factory.

### 3.3.3. Design pattern comportamentale - Template Method

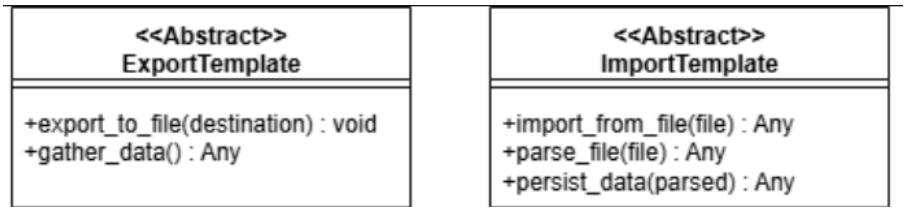


Figura 2: Template Method Pattern

Il *Template Method pattern*$_G$ è un *pattern comportamentale*$_G$ che permette di definire in una classe astratta la struttura generale di un algoritmo, delegando alle sottoclassi la responsabilità di implementare i singoli passi variabili. Esso stabilisce l'ordine in cui eseguire le operazioni, ma non ne definisce il contenuto specifico: i passi astratti devono essere implementati dalle sottoclassi, mentre quelli opzionali possono avere un'implementazione di default. In questo modo l'algoritmo rimane intatto e coerente, ma ogni sottoclasse può personalizzarne i dettagli.

Nel nostro progetto devono essere gestite più operazioni ripetitive: importare dati da file (*CSV*$_G$, *JSON*$_G$) per diverse entità (domande, set di domande, risultati di test) e esportare gli stessi dati su file.

Tutte queste operazioni seguono un flusso comune:

- **Importazione**: apertura del file, parsing dei dati, persistenza su database, gestione degli errori.
- **Esportazione**: recupero dei dati, serializzazione e scrittura su file.

Senza il Template pattern avremmo dovuto duplicare la logica di lettura/scrittura e la gestione degli errori in ogni classe. Il pattern permette invece di:

- Definire i passi comuni una sola volta nelle classi astratte (nel caso nostro `ImportTemplate` e `ExportTemplate`).
- Obbligare le classi figlie a fornire solo i passi specifici (come il parsing del file o la raccolta dei dati).
- Garantire che il flusso di importazione/esportazione rimanga invariato e correttamente gestito anche quando cambiano i tipi di dati.

#### 3.3.3.1. Realizzazione del pattern

La classe `ExportTemplate` si trova in `utils/export_template.py` ed è la base per tutte le esportazioni. Contiene il metodo astratto `gather_data`, che dovrà essere implementato dalle sottoclassi per raccogliere i dati da esportare, e il metodo finale `export_to_file`, che definisce la sequenza di passi per l'esportazione: invoca `gather_data`, poi usa `write_dataset` per scrivere su file. Poiché è contrassegnato con `@final`, le sottoclassi non possono sovrascrivere `export_to_file`; in questo modo l'algoritmo di esportazione resta invariato.

python

```python
class ExportTemplate(ABC):
    @abstractmethod
    def gather_data(self) -> Any:
        """Raccoglie i dati da esportare."""
        pass

    @final
    def export_to_file(self, destination: Union[str, IO[Any]]) -> None:
        """Esporta i dati raccolti su `destination`."""
```

```
        from utils.file_writer_utils import write_dataset
        data = self.gather_data()
        write_dataset(data, destination)


```

Analogamente `ImportTemplate`, in `utils/import_template.py`, definisce il flusso di importazione. Le sottoclassi devono implementare i metodi astratti `parse_file` (per leggere e interpretare i dati dal file) e `persist_data` (per salvare i dati nel database). Il metodo `import_from_file`, marcato `@final`, esegue il flusso standard: chiama `parse_file`, poi `persist_data`, gestisce eventuali eccezioni e restituisce il risultato

python

```python
class ImportTemplate(ABC):
    @final
    def import_from_file(self, file: IO[Any]) -> Any:
        try:
            parsed = self.parse_file(file)
            result = self.persist_data(parsed)
            return result
        except Exception as exc:
            logger.exception("Errore durante l'importazione: %s", exc)
            raise ValueError("Errore durante l'importazione") from exc

    @abstractmethod
    def parse_file(self, file: IO[Any]) -> Any:
        pass

    @abstractmethod
    def persist_data(self, parsed: Any) -> Any:
        pass

```

#### 3.3.3.2. Applicazione del pattern

Nel file `models/question.py` la classe `QuestionImporter` eredita sia `ImportTemplate` che `ExportTemplate`. Essa implementa i passi specifici per importare ed esportare domande:

python

```python
class QuestionImporter(ImportTemplate, ExportTemplate):
    """Importer per le domande basato su :class:`ImportTemplate`
  e :class:`ExportTemplate`."""

    def parse_file(self, file: IO[Any]) -> pd.DataFrame:  # type: ignore[override]
        """Legge le domande dal file usando ``read_questions``."""
        return read_questions(file)

    def persist_data(self, df: pd.DataFrame) -> Dict[str, Any]:  # type:
  ignore[override]
        """Persiste i dati tramite :meth:`Question._persist_entities`."""
        imported, warnings = Question._persist_entities(df)
        return {"success": True, "imported_count": imported, "warnings": warnings}

    def gather_data(self) -> pd.DataFrame:  # type: ignore[override]
        """Recupera tutte le domande dal database."""
        questions = Question.load_all()
        return pd.DataFrame([q.__dict__ for q in questions])
```

- `parse_file` utilizza la funzione `read_questions` per caricare le domande da un file *CSV*$_G$.

- `persist_data` chiama la funzione di classe `_persist_entities` per inserire nel *database*$_G$ solo le domande nuove e restituisce eventuali avvisi.
- `gather_data` recupera tutte le domande esistenti e le converte in un DataFrame pronto per l'esportazione.

Queste implementazioni seguono il contratto stabilito dalla classe base e consentono al metodo template di funzionare senza conoscere i dettagli di lettura/scrittura.

La classe `QuestionSetImporter`, definita in `models/question_set.py`, eredita anch'essa entrambe le classi template. Essa implementa:

```python
class QuestionSetImporter(ImportTemplate, ExportTemplate):
    """Importer per i set di domande basato su :class:`ImportTemplate`
e :class:`ExportTemplate`."""

    def parse_file(self, file: IO[Any]) -> List[Dict[str, Any]]:  # type:
ignore[override]
        """Legge i set di domande dal file usando ``read_question_sets``."""
        return read_question_sets(file)

    def persist_data(self, parsed: List[Dict[str, Any]]) -> PersistSetsResult:  #
type: ignore[override]
        """Persiste i dati tramite :meth:`QuestionSet._persist_entities`."""
        from controllers.question_controller import load_questions
        from controllers.question_set_controller import load_sets

        current_questions = load_questions()
        current_sets = load_sets()

        return QuestionSet._persist_entities(parsed, current_questions, current_sets)

    def gather_data(self) -> List[Dict[str, Any]]:  # type: ignore[override]
        """Recupera tutti i set di domande con i dettagli delle domande."""
        from models.question import Question

        sets = QuestionSet.load_all()
        questions = {
            q.id: {"id": q.id, "domanda": q.domanda, "risposta_attesa":
q.risposta_attesa, "categoria": q.categoria}
            for q in Question.load_all()
        }
        data: List[Dict[str, Any]] = []
        for s in sets:
            q_list = [questions.get(qid, {"id": qid}) for qid in s.questions]
            data.append({"name": s.name, "questions": q_list})
        return data

```

- `parse_file` per leggere i set di domande tramite `read_question_sets`.
- `persist_data` per creare i set nel database e gestire l'inserimento di nuove domande.
- `gather_data` per restituire tutti i set con i dettagli delle domande.

Il metodo template `import_from_file` di `ImportTemplate` viene riutilizzato per eseguire l'importazione in modo uniforme, mentre la logica specifica è incapsulata nei metodi implementati.

Anche `TestResultImporter` (in `models/test_result.py`) segue lo stesso schema:

```python
class TestResultImporter(ImportTemplate, ExportTemplate):
    """Importer per i risultati di test basato su :class:`ImportTemplate` e :class:`ExportTemplate`."""

    def parse_file(self, file: IO[Any]) -> pd.DataFrame:  # type: ignore[override]
        """Legge i risultati dal file usando ``read_test_results``."""
        return read_test_results(file)

    def persist_data(self, df: pd.DataFrame) -> Dict[str, Any]:  # type: ignore[override]
        """Persiste i dati tramite :meth:`TestResult._persist_entities`."""
        added_count = TestResult._persist_entities(df)
        if added_count > 0:
            TestResult.refresh_cache()
        message = (
            f"Importati {added_count} risultati."
            if added_count > 0
            else "Nessun nuovo risultato importato."
        )
        return {"success": True, "imported_count": added_count, "message": message}

    def gather_data(self) -> pd.DataFrame:  # type: ignore[override]
        """Recupera tutti i risultati dei test dal database."""
        return TestResult.load_all_df()
```

- `parse_file` usa `read_test_results` per creare un DataFrame dai risultati del test.
- `persist_data` salva i dati nel database e aggiorna eventuali cache.
- `gather_data` restituisce un DataFrame con tutti i risultati.

Il fatto che tutte queste classi implementino gli stessi metodi astratti consente di gestire diverse entità con un'unica interfaccia coerente, riducendo la duplicazione di codice e facilitando la manutenzione.

## 3.4. Diagramma delle classi

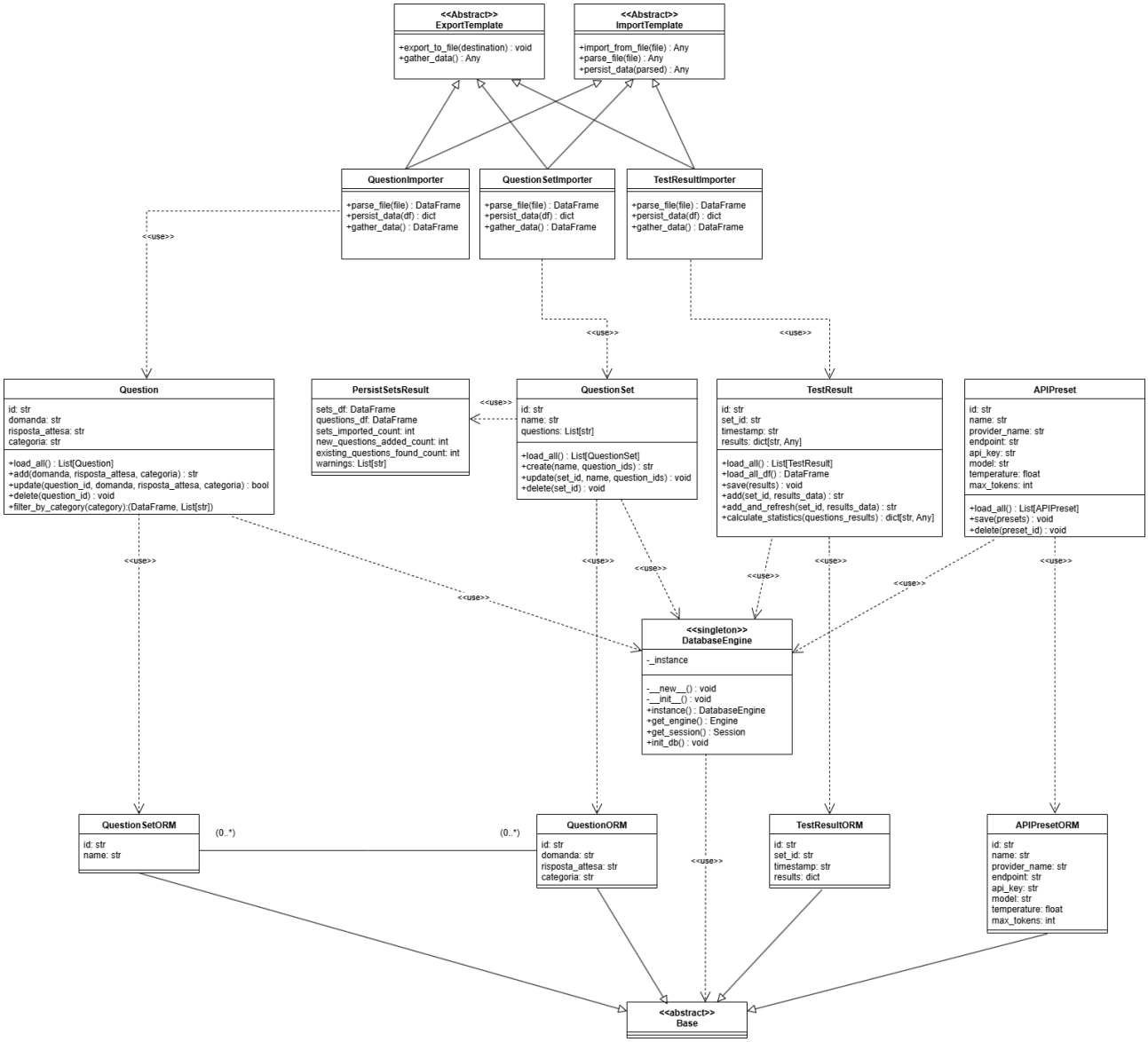


Figura 3: Diagramma delle classi

## 3.5. Database

Come già citato nella sezione *Tecnologie* del documento, il nostro prodotto utilizza *MySQL*$_G$ come *database relazionale*$_G$ principale per la gestione dei dati. Il database supporta SQL standard ed è facilmente integrabile con *Python*$_G$ tramite librerie (es. *PyMySQL*$_G$ o *SQLAlchemy*$_G$). Viene utilizzato per archiviare domande/risposte e risultati di test e report.

### 3.5.1. La scelta di MySQL

*MySQL*$_G$ è stato scelto per la sua stabilità e diffusione: viene largamente usato in contesti di produzione, compresi progetti aziendali. È *open-source*$_G$ e gratuito, perfetto per progetti accademici che richiedono pubblicazione su *GitHub*$_G$. È supportato da strumenti di backup, replica e recovery, utili se il progetto evolve in una demo pubblica. Anche se il capitolato lascia libertà nella scelta del sistema di archiviazione, MySQL rappresenta una scelta solida, scalabile e integrabile, che permette di affrontare sia la fase di sviluppo rapido sia quella di dimostrazione pubblica con affidabilità.

Il progetto richiede una gestione di archivi modificabili di domande e risposte, risultati dei test, archiviazione di test run diversi, eventualmente set multipli di domande; quindi operazioni Create, Read, Update, Delete devono essere gestibili facilmente. MySQL supporta in modo nativo queste operazioni tramite *framework*$_G$ web comuni (Streamlit) e interfacce grafiche come phpMyAdmin. Questo schema si adatta bene ad un modello relazionale, con tabelle collegate da chiavi esterne (foreign keys). MySQL è facile da installare anche localmente.

Il progetto richiede l'adozione di *pattern architetturali*$_G$ flessibili e manutenibili. L'uso di un database relazionale come MySQL permette una progettazione basata su modelli Entity-Relationship, utile per mantenere coerenza nei dati e facilitarne l'evoluzione. Può essere contenuto in un *Docker container*$_G$, semplificando il rilascio del *PoC*$_G$ e del *MVP*$_G$.

# 4. Funzionalità principali

L'applicazione offre un insieme di funzionalità chiave, accessibili tramite il menu di navigazione laterale. Di seguito vengono descritte le sezioni principali dell'app e le rispettive funzionalità:

## 4.1. Home

È la pagina introduttiva che descrive lo scopo dell'applicazione e guida l'utente attraverso i passi iniziali. Viene presentato un riassunto delle funzionalità con icone illustrative (Gestione Domande, Supporto Multi-Provider, Valutazione Automatizzata, Analisi Avanzata) insieme a un elenco puntato della sezione "Come iniziare". Questo aiuta l'utente a capire il flusso generale: si parte dalla configurazione delle *API*$_G$, per poi inserire domande e risposte attese, raggrupparle in un set e infine avviare i test, con la successiva visualizzazione dei risultati al termine dell'esecuzione.

## 4.2. Configurazione API

Consente di gestire i preset di connessione ai provider *LLM*$_G$. L'utente può creare uno o più preset specificando una serie di parametri necessari, come:

- nome;
- provider ed endpoint;
- chiave API di autenticazione;
- modello;
- parametri aggiuntivi di generazione (temperatura e limiti di uso token).

La pagina permette di salvare nuovi preset e modificare o eliminare quelli esistenti, applicando alcuni controlli di validazione, come ad esempio sul nome, che non deve duplicare quello di un preset già presente.

È inoltre presente la funzionalità «Test connessione API», dove l'utente, con un semplice click, può verificare che credenziali ed endpoint impostati funzionino correttamente. Il test avviene inviando in background, una richiesta semplice al servizio LLM, dove viene chiesto di restituire un messaggio di conferma che verrà controllato. Una volta controllato il messaggio, viene avvertito l'utente se la connessione ha avuto successo o meno, indicando gli eventuali errori riscontrati, che potrebbero essere problemi di rete, credenziali non corrette, limiti raggiunti o simili.

Una volta cliccato *Salva*, i preset vengono memorizzati nel *database*$_G$ (sotto la tabella `api_presets`) per essere richiamati dall'interfaccia ogni volta che ne è richiesto l'utilizzo.

## 4.3. Gestione domande

In questa sezione è possibile gestire le coppie di domande e risposte, potendole creare e organizzare in diversi modi. Le funzionalità disponibili vengono elencate di seguito.

### 4.3.1. Inserimento manuale di una nuova domanda

L'utente digita il testo della domanda insieme alla risposta attesa e può, opzionalmente, assegnare una categoria tematica. Il sistema genera di un ID univoco per la domanda (*UUID*$_G$) se non fornito e salva il record nel database. La lista di coppie domanda-risposta visualizzate viene aggiornata immediatamente svotando e ricaricaando la memoria cache.

### 4.3.2. Modifica/Eliminazione di domande esistenti

L'utende, attraverso l'interfaccia, può selezionare una coppia domanda-risposta e scegliere se aggiornarne i campi oppure cancellarla. Il controller corrispondente alla coppia scelta aggiorna il database

modificandola o rimuovendola e subito dopo viene aggiornata la cache. Inoltre, in caso di cancellazione, rimuove anche eventuali riferimenti presenti in set di domande (cancellando le associazioni many-to-many) prima di eliminare la domanda.

## 4.4. Importazione di domande da file

Il sistema supporta il caricamento di file *CSV*G o *JSON*G contententi coppie domande-risposte, purchè siano strutturati correttamente, seguendo il formato atteso (indicato anche nel *Manuale Utente v1.0.0*) che richiede siano popolati i campi domanda e risposta.

La funzione di import avviene seguendo questi passaggi:

- Lettura del file indicato, tramite utility di parsing specifiche per *CSV*G e *JSON*G.
- Normalizzazione dei dati inseriti, facendo controlli sui nomi per evitare duplicati:
  - Le domande importate che hanno ID già presenti nel database vengono saltate di default. Se il file dovesse contenere ID duplicati, viene considerata solo la prima occorrenza, mentre le successive verranno scartate avvisando l'utente con un warning.
- Inserimento della lista domande-risposte normalizzate:
  - L'inserimento avviene in blocco, raccogliendo informazioni sul numero di righe aggiunte e warning eventuali.
  - La logica è stata implementata nel metodo «`Question\_persist_entities`», che confronta gli ID del file importato con quelli esistenti per poi costruire la lista finale di record che andranno inseriti.
- Aggiornamento della cache e restituzione di un report, come detto precedentemente, con tutte le informazioni sul numero di righe importate ed eventuali warning.

## 4.5. Esportazione di domande

In modo analogo, l'interfaccia offre l'opzione di poter scaricare tutte le domande in formato *CSV*G o *JSON*G. Viene generato un *dataset*G completo con tutti i campi (ID, domanda, risposta attesa, categoria) e inviato come file all'utente, utile in caso di backup oppure per effettuare editing offline.

## 4.6. Gestione set di domande

Questa sezione consente di aggregare le domande in insiemi (set), al fine di definire test strutturati e organici. Le funzionalità disponibili vengono elencate di seguito.

### 4.6.1. Creazione di un nuovo set

L'utente assegna un nome al set e seleziona tramite interfaccia di multi-selezione le domande che ne faranno parte. Il controller `question_set_controller.create_set` genera un nuovo identificativo (ID) per il set e crea le opportune relazioni many-to-many tra il set e le domande selezionate. Al termine del salvataggio, la cache dei set viene aggiornata per riflettere le modifiche.

### 4.6.2. Modifica di un set esistente

Consente di rinominare un set e/o modificare l'elenco di domande associate. Il sistema aggiorna la riga corrispondente nel database e sostituisce le relazioni many-to-many in base agli ID forniti. Anche in questo caso, la *cache*G dei set viene ricaricata immediatamente.

#### 4.6.3. Eliminazione di un set

L'eliminazione rimuove il set dal *database*$_G$, senza eliminare le domande a esso collegate. Vengono rimossi unicamente i riferimenti nella tabella di relazione many-to-many. A operazione completata, la *cache*$_G$ viene aggiornata.

### 4.7. Importazione ed esportazione di set di domande

L'applicazione consente di importare ed esportare set di domande in formato *CSV*$_G$ o *JSON*$_G$.

#### 4.7.1. Importazione di set

Il formato CSV richiesto include colonne quali `name`, `id` (della domanda), `domanda`, `risposta_attesa` e `categoria`. L'importazione segue il seguente processo:

1. **Verifica nome set**: se il nome del set è già presente nel database, l'inserimento viene ignorato per evitare duplicati.
2. **Gestione delle domande**: per ogni domanda del set, il sistema verifica la presenza di un ID già esistente. Se l'ID non è presente e sono disponibili testo domanda e risposta attesa, la domanda viene creata automaticamente tramite il metodo `add_question_if_not_exists`, che garantisce l'assenza di duplicati.
3. **Creazione set e relazioni**: il set viene creato con un nuovo ID e popolato con riferimenti alle domande, siano esse nuove o già presenti.
4. **Gestione warning**: il sistema registra avvisi per:
   - set duplicati saltati;
   - domande senza ID né contenuti validi;
   - ID duplicati all'interno dello stesso file.
5. **Resoconto finale**: al termine viene generato un oggetto `PersistSetsResult` contenente il numero di set importati, di nuove domande create e di domande già presenti.
6. **Aggiornamento cache**: la cache dei set viene aggiornata per rendere immediatamente disponibili i nuovi dati.

#### 4.7.2. Esportazione di set

L'esportazione produce un file **JSON** con struttura annidata: ogni set include il proprio nome e la lista delle domande (complete di testo e risposta attesa) che contiene.

### 4.8. Filtri per categorie

La pagina **Gestione Set** offre la possibilità di filtrare i set mostrati in base alle categorie delle domande. Ad esempio, l'utente può voler visualizzare solo i set che contengono almeno una domanda di categoria "Geografia" e almeno una di "Matematica". Il controller `prepare_sets_for_view` applica filtri incrociati: per ogni set calcola la lista dettagliata delle domande (inserendo categoria e testo in `questions_detail`) e include il set nell'elenco solo se tutte le categorie selezionate sono rappresentate al suo interno. Questa funzionalità aiuta a focalizzarsi su set specifici quando il numero di set e la varietà di argomenti cresce.

### 4.9. Esecuzione test

È il cuore dell'applicazione, dove l'utente può avviare una sessione di test sui modelli *LLM*$_G$.

#### 4.9.1. Selezioni iniziali

- Selezione del **Set di Domande** (menu a tendina con i set disponibili).
- Selezione del **Preset API per la Generazione** delle risposte (es. "GPT-4 – OpenAI").
- Selezione del **Preset API per la Valutazione** delle risposte (può coincidere con quello di generazione o essere diverso). Se l'interfaccia non prevede due scelte separate, può essere riutilizzato lo stesso preset; internamente `run_test` accetta due configurazioni distinte: `gen_preset_config` ed `eval_preset_config`.
- Avvio del test tramite pulsante.

#### 4.9.2. Flusso di esecuzione (`test_controller.run_test`)

##### 4.9.2.1. Preparazione dei dati

- Caricamento di tutte le domande del set selezionato dal *database*$_G$ (tramite *cache*$_G$).
- Predisposizione di un dizionario per accesso rapido alle domande per ID.
- Recupero del nome del set.
- Preparazione delle configurazioni dettagliate (chiave, *endpoint*$_G$, modello, parametri) per i due preset selezionati.

##### 4.9.2.2. Generazione risposte con LLM

- Per ciascuna domanda: recupero del testo e della risposta attesa dal database.
- Invocazione di `generate_answer(question, gen_preset_config)`, che:
  - costruisce un prompt contestualizzato (ad esempio "Rispondi alla seguente domanda in modo preciso e accurato: …");
  - chiama l'*API*$_G$ del provider selezionato e restituisce la risposta testuale.
- Gestione errori: in caso di eccezione (es. timeout, chiave errata) il controller assegna un risultato di fallback:
  - risposta generata marcata come messaggio di errore (con descrizione);
  - evaluation impostata a punteggio 0 con spiegazione dell'errore. La pipeline prosegue senza interrompere l'intero test.

##### 4.9.2.3. Valutazione risposte con LLM

- Per ogni risposta generata (non in errore) viene invocata `evaluate_answer(question, expected_answer, actual_answer, eval_preset_config)`.
- Prompt di valutazione con: domanda originale, risposta attesa e risposta effettiva. Il modello agisce come "valutatore esperto" secondo tre criteri:
  - **Somiglianza semantica**;
  - **Correttezza fattuale**;
  - **Completezza** (punteggi 0–100) + punteggio complessivo (0–100) e breve spiegazione.
- Richiesta di output in *JSON*$_G$ con campi: `score`, `explanation`, `similarity`, `correctness`, `completeness`.

- Gestione errori: se il *JSON*$_G$ è invalido/mancante o l'*API*$_G$ fallisce, viene sollevata un'eccezione intercettata dal controller; l'evaluation viene sostituita con punteggio 0 e spiegazione contenente l'errore.

#### 4.9.2.4. Aggregazione risultati

- Per ogni domanda vengono salvati il testo, la risposta attesa, la risposta generata (o errore) e `evaluation` con punteggi e spiegazione.
- Il calcolo delle metriche globali avviene tramite `TestResult.calculate_statistics`:
  - `avg_score`: media aritmetica dei punteggi.
  - `radar_metrics`: medie di somiglianza, correttezza, completezza.
  - `per_question_scores`: lista di coppie domanda–punteggio per confronti/grafici.
- Preparazione di un oggetto risultato con metadati (nome set, timestamp, metodo usato — nel *MVP*$_G$: "LLM").

#### 4.9.2.5. Salvataggio e output

- Persistenza nel database (tabella `test_results`) tramite `TestResult.add_and_refresh`: creazione *UUID*$_G$, scrittura dei dati *JSON*$_G$ e aggiornamento *cache in-memory*$_G$.
- Ritorno alla UI di:
  - `result_id`
  - `avg_score`, `per_question_scores`, `radar_metrics`
  - `results` (dettaglio per domanda)
  - `results_df` (DataFrame aggiornato per le liste in UI)
- La *UI*$_G$ mostra immediatamente il punteggio medio e gli indicatori principali al termine dell'esecuzione.

## 4.10. Visualizzazione risultati

Sezione dedicata alla consultazione e all'analisi dei test salvati.

### 4.10.1. Filtri

- Filtro per **Set di Domande** (confronto sul nome del set associato al risultato).
- Filtro per **Modello LLM** usato in generazione (lettura del nome modello/preset nel *JSON*$_G$ risultati).

  I filtri sono combinabili (AND) tramite due menu a tendina: "Filtra per Set" e "Filtra per Modello LLM".

### 4.10.2. Selezione del test

- La lista dei risultati è ordinata in ordine cronologico decrescente.
- Ogni voce ha un'etichetta descrittiva con data/ora, nome del set, punteggio medio e metodo, generata da `prepare_select_options`.
- L'ID del risultato selezionato è mantenuto nello **state** dell'applicazione.

#### 4.10.3. Confronto tra due risultati

- Opzionalmente si seleziona un secondo risultato (escludendo quello già scelto).
- Le successive visualizzazioni mostrano i due test affiancati.

#### 4.10.4. Importazione/Esportazione risultati

- Esportazione in *JSON*$_G$ del singolo risultato selezionato (contenuti del record in `test_results`) o di **tutti i risultati** (lista completa).
- Importazione da file *JSON*$_G$ tramite `import_results_action`: inserimento dei nuovi record evitando duplicati tramite `TestResult._persist_entities`, aggiornando la *cache*$_G$ e mostrando messaggi di conferma/errore.

### 4.11. Visualizzazione dettagliata

Una volta scelto il risultato (e facoltativamente il confronto), la *UI*$_G$ presenta i seguenti risultati:

#### 4.11.1. Grafico a barre dei punteggi per domanda

- Istogramma con domande sull'asse X (etichette abbreviate) e punteggio percentuale sull'asse Y.
- In modalità confronto, barre raggruppate per domanda (test selezionato vs. test di confronto).

#### 4.11.2. Grafico radar (spider) delle metriche medie

- Grafico polare su tre assi: **Somiglianza**, **Correttezza**, **Completezza**.
- In confronto, due poligoni distinti per evidenziare differenze (es. modello A più corretto ma meno completo di B).

#### 4.11.3. Metriche numeriche aggregate

- Valori medi delle tre metriche mostrati sotto il radar, con indicatori (es. `st.metric`) per test selezionato ed eventuale confronto.

#### 4.11.4. Tabella comparativa per domanda

- Elenco delle domande con punteggio del test selezionato, punteggio del test di confronto e Delta.
- Ordinamento/scroll per individuare variazioni significative.

#### 4.11.5. Dettagli per ciascuna domanda

- Pannelli espandibili contenenti:
  - **Domanda** (testo completo).
  - **Risposta attesa**.
  - **Risposta generata/effettiva** del modello.
  - **Valutazione LLM**: punteggio complessivo (%) e **explanation**, oltre ai tre sottopunteggi (Somiglianza, Correttezza, Completezza).

In caso di errore nella generazione/valutazione, l'**explanation** riporta il messaggio d'errore (es. *timeout API*$_G$).

Complessivamente, queste funzionalità coprono l'intero ciclo di valutazione, dalla configurazione dell'ambiente di test, alla gestione dei dati di prova, fino all'esecuzione e all'analisi dei risultati, offrendo all'utente un controllo completo sul processo.

# 5. Moduli e componenti

Il codice del progetto è strutturato in moduli Python, organizzati secondo il principio della separazione delle responsabilità. Ciascun modulo raccoglie al suo interno funzionalità affini, così da favorire chiarezza, riusabilità e manutenibilità del software.

In questa sezione vengono descritti i principali componenti del sistema, specificandone il ruolo all'interno dell'applicazione, le responsabilità assegnate e alcune delle funzioni chiave implementate. Tale descrizione ha lo scopo di fornire una visione d'insieme dell'architettura del progetto e di agevolare la comprensione delle scelte progettuali adottate.

## 5.1. Interfaccia utente – Views

Le viste sono raccolte nel pacchetto `views/` e corrispondono alle pagine dell'applicazione *Streamlit*$_G$. Ogni file *Python*$_G$ in questa directory definisce una funzione `render()` decorata con `@register_page("Nome Pagina")`. Questo meccanismo popola un registro globale delle pagine.

Nel file principale `app.py`, dopo aver importato tutte le viste, viene costruito un menu di navigazione laterale che elenca i nomi registrati e richiama la funzione di rendering appropriata in base alla selezione dell'utente. In questo modo l'applicazione gestisce più pagine senza dipendere da componenti esterni.

Le principali viste e i loro contenuti (già introdotti nella sezione **Funzionalità**) sono:

- **Home** (`views/home.py`) – Pagina introduttiva e guida iniziale. Contiene elementi statici (testo, descrizioni, icone delle funzionalità) e una lista ordinata di passi consigliati per iniziare. Non include logica applicativa, se non la chiamata a `add_home_styles()` per applicare uno stile CSS personalizzato.
- **Configurazione API**(`views/api_configurazione.py`) – Mostra un modulo Streamlit per inserire i campi di un preset API (nome, endpoint, chiave, modello, temperatura, max token). I dati vengono inviati al controller `api_preset_controller.save_preset`, con gestione del feedatabaseack (messaggi di successo o errore). La pagina mostra inoltre la tabella dei preset salvati, con pulsanti di modifica ed eliminazione. Include un pulsante di test connessione, che richiama `api_preset_controller.test_api_connection` e visualizza l'esito direttamente nell'interfaccia.

  Questa vista fa ampio uso dello stato di sessione (`st.session_state`) per gestire selezioni e messaggi temporanei.
- **Gestione Domande** (`views/gestione_domande.py`) – Visualizza la tabella delle domande e offre widget per le operazioni *CRUD*$_G$:
  - Inserimento di una nuova domanda.
  - Modifica diretta in tabella (in-line editing) con pulsanti di salvataggio o cancellazione.
  - Esportazione (*CSV*$_G$/*JSON*$_G$) e importazione tramite caricamento file.

  La vista utilizza funzioni del `question_controller` come `load_questions`, `add_question`, `update_question`, `delete_question`. L'importazione è gestita da `import_questions_action` con gestione delle eccezioni per eventuali errori di formato.

  È inoltre disponibile un filtro per categoria, basato su `Question.filter_by_category`, che utilizza i dati in *cache*$_G$ per mostrare solo le domande pertinenti.
- **Gestione Set** (`views/gestione_set.py`) – Strutturata in modo simile alla gestione domande, ma focalizzata sui set. Mostra l'elenco dei set con conteggio domande e categorie, usando i dati preparati da `prepare_sets_for_view`. Funzionalità principali:

- Creazione di un nuovo set (nome + selezione multipla di domande).
- Modifica nome set e aggiunta/rimozione domande.
- Import/export dei set tramite `export_sets_action` e `question_set_controller`.
- Filtri per categoria, applicabili tramite caselle di selezione o menu a tendina.

- **Esecuzione Test** (`views/esecuzione_test.py`) – Interfaccia per avviare i test:
  - Selezione di un set di domande (`question_set_controller.load_sets()`).
  - Selezione dei preset di generazione e valutazione (`api_preset_controller.list_presets()`).
  - Pulsante **Esegui Test**, che richiama `test_controller.run_test` e salva i risultati nello `st.session_state`.

  Durante l'esecuzione l'interfaccia mostra un indicatore di caricamento (`st.spinner`). Al termine, vengono mostrati punteggio medio e link rapido a **Visualizzazione Risultati**. Nel nostro progetto i risultati vengono salvati direttamente nello stato globale (`st.session_state.results`).
- **Visualizzazione Risultati** (`views/visualizza_risultati.py`) – All'apertura carica i risultati (`get_results`) e i set di domande (`load_sets`). Funzionalità principali:
  - Filtri per set e modello, tramite `get_results(filter_set, filter_model)`.
  - Selezione di un risultato principale e di uno di confronto, con opzioni generate da `prepare_select_options`.
  - Esportazione e importazione dei risultati in *JSON*$_{\text{G}}$.
  - Visualizzazione grafica:
    - Grafico a barre con punteggi per domanda (via `plotly_express.bar`).
    - Grafico radar con metriche aggregate (`stats["radar_metrics"]`).
  - Visualizzazione delle metriche sintetiche (`st.metric`) e tabelle comparative in caso di confronto.
  - Dettaglio domanda per domanda, con testo, risposte e valutazioni, presentati in sezioni espandibili (`st.expander`).

## 5.2. Moduli di supporto UI

Oltre alle viste, due moduli completano la gestione dell'interfaccia:

- **views/style_utils.py** – Contiene funzioni per applicare stili CSS personalizzati (`add_global_styles()`, `add_page_header()`, `add_section_title()`).
- **views/session_state.py** – Definisce `initialize_session_state()`, che imposta valori di default nello stato di sessione (es. messaggi import/export, variabili condivise tra pagine). Questa funzione viene richiamata in `app.py` per garantire che tutte le chiavi di sessione siano presenti ed evitare errori

## 5.3. Controller (Logica di business)

I controller si trovano nel pacchetto `controllers/` e sono moduli funzionali (non classi) che espongono funzioni di utilità per la gestione dei dati. Implementano la logica richiesta dalle viste, combinando chiamate ai modelli e alle utilità di sistema.

Di seguito i principali controller e le loro responsabilità.

- **controllers/question_controller.py** – Gestisce le operazioni sulle domande.

Funzioni principali:

- `load_questions()` – restituisce tutte le domande come DataFrame, richiamando `utils.cache.get_questions`.
- `refresh_questions()` – invalida e ricarica la cache delle domande.
- `add_question(domanda, risposta_attesa, ...)` – aggiunge una nuova domanda al database tramite `Question.add`, poi aggiorna la *cache*$_G$.
- `add_question_if_not_exists(...)` – aggiunge una domanda solo se l'ID non è già presente. Usata soprattutto in fase di import set.
- `update_question(question_id, ...)` – aggiorna i campi di una domanda esistente (solo quelli forniti) e ricarica la *cache*$_G$.
- `delete_question(question_id)` – elimina una domanda dal *database*$_G$ e aggiorna la *cache*$_G$.
- `get_filtered_questions(category)` – filtra le domande per categoria con `Question.filter_by_category`.
- `import_questions_action(uploaded_file)` – gestisce l'import da file utente. Valida l'input e delega a `question_importer.import_from_file`. In caso di errore solleva eccezione, altrimenti ricarica la cache e restituisce il DataFrame aggiornato.
- `export_questions_action(destination)` – esporta le domande correnti tramite `question_importer.export_to_file`.

- **controllers/question_set_controller.py** – Gestisce i set di domande.

  Funzioni principali:

  - `load_sets()` – recupera tutti i set come DataFrame da cache.
  - `refresh_question_sets()` – ricarica la cache dei set.
  - `create_set(name, question_ids)` – crea un nuovo set e aggiorna la cache.
  - `update_set(set_id, name, question_ids)` – modifica un set esistente e restituisce il DataFrame aggiornato.
  - `delete_set(set_id)` – elimina un set e aggiorna la cache.
  - `export_sets_action(destination)` – esporta i set tramite `question_set_importer.export_to_file`.
  - `prepare_sets_for_view(selected_categories)` – prepara i dati per la visualizzazione nella UI:
    1. Carica domande e set dalla *cache*$_G$.
    2. Rielabora le domande con `utils.data_format_utils.format_questions_for_view`.
    3. Arricchisce i set con dettagli delle domande (`build_questions_detail`).
    4. Se presenti categorie selezionate, filtra i set con l'helper `has_categories`.
    5. Restituisce un dizionario con domande, set filtrati, set completi e categorie disponibili.

- **controllers/api_preset_controller.py** – Gestione dei preset API per i servizi LLM.

  Funzioni principali:

  - `load_presets()` – ottiene tutti i preset dalla *cache*$_G$.
  - `refresh_api_presets()` – aggiorna la cache dei preset.

- `list_presets(df=None)` – restituisce l'elenco dei preset come lista di dizionari, utile per popolare menu di selezione.
- `get_preset_by_id(preset_id)` / `get_preset_by_name(name)` – recuperano un preset specifico.
- `validate_preset(data, preset_id=None)` – valida i campi di un preset (es. nome univoco).
- `save_preset(data, preset_id=None)` – salva un nuovo preset o aggiorna uno esistente. Gestisce validazioni, normalizzazione campi e aggiornamento cache.
- `delete_preset(preset_id)` – elimina un preset dal database e aggiorna la cache.
- `test_api_connection(api_key, endpoint, model, temperature, max_tokens)` – verifica la connessione con l'*API*$_G$ *LLM*$_G$ tramite una chiamata di prova, restituendo esito positivo o dettagli di errore.

- **controllers/test_controller.py** – Logica per l'esecuzione dei test e la gestione dei risultati.

  Funzioni principali:

  - `load_results()` – recupera i risultati di test da cache.
  - `refresh_results()` – ricarica la cache dei risultati.
  - `import_results_action(uploaded_file)` – importa risultati da file JSON.
  - `export_results_action(destination)` – esporta i risultati correnti.
  - `generate_answer(question, client_config)` – genera una risposta a una domanda tramite LLM.
  - `evaluate_answer(question, expected_answer, actual_answer, client_config)` – valuta una risposta generata rispetto a quella attesa.
  - `run_test(...)` – esegue un test completo:
    1. Carica le domande dal database.
    2. Genera e valuta risposte con LLM, gestendo eventuali errori.
    3. Salva i risultati e calcola statistiche aggregate (`TestResult.calculate_statistics`).
    4. Ritorna un dizionario con punteggi medi, dettagli per domanda e dataset aggiornato.

- **controllers/result_controller.py** – Funzioni di supporto per visualizzazione e filtro dei risultati.

  Funzioni principali:

  - `get_results(filter_set, filter_model)` – filtra i risultati per set e/o modello.
  - `list_set_names(results_df, question_sets_df)` – estrae i nomi dei set presenti nei risultati.
  - `list_model_names(results_df)` – elenca i modelli LLM utilizzati.
  - `prepare_select_options(results_df, question_sets_df)` – genera le opzioni da mostrare nella *UI*$_G$ per la selezione dei risultati, includendo timestamp, set, punteggio medio e modello.

  In sintesi, i controller forniscono un'*API*$_G$ ad alto livello per le view, astraendo i dettagli di accesso al database e di manipolazione dati.

  Vale la pena notare che molti controller sfruttano i meccanismi di caching: funzioni (`load_questions`, `load_sets`, `load_presets`, ecc.) recuperano i dati direttamente dalla cache, mentre le scritture passano sempre dai modelli, seguite da un `refresh` per garantire coerenza. Questo approccio semplifica la logica delle viste e migliora l'efficienza complessiva.

## 5.4. Modelli e accesso al database

La cartella `models/` rappresenta lo strato dati dell'applicazione. Qui si trovano sia le classi *ORM*$_G$ che mappano le tabelle del *database*$_G$, sia classi di supporto con metodi statici per la gestione dei dati, oltre a componenti per importazione ed esportazione basati su template generici.

Di seguito i principali file:

- `models/orm_models.py` – Contiene le definizioni delle classi *ORM*$_G$ di *SQLAlchemy*$_G$ che rappresentano le tabelle del *database*$_G$.

  In particolare:

  - `QuestionORM`, `QuestionSetORM`, `TestResultORM` e `APIPresetORM` sono sottoclassi della `Base Declarative` (definita in `models.database.Base`). Definiscono colonne e relazioni: ad esempio `QuestionSetORM` gestisce la relazione molti-a-molti con le domande tramite la tabella di associazione `question_set_questions`. `TestResultORM` utilizza una colonna *JSON*$_G$ per i risultati dettagliati. Queste classi vengono usate soprattutto dai modelli applicativi di livello superiore, mentre i controller le richiamano solo raramente in modo diretto.

- `models/database.py` – Gestisce la configurazione del database e implementa un pattern *Singleton*$_G$ per engine e sessioni:

  - La classe `DatabaseEngine` è esposta come singleton *thread-safe*$_G$, gestito tramite lock. Al primo accesso, `DatabaseEngine.instance()` inizializza l'engine SQLAlchemy collegandosi a MySQL con le credenziali lette da file di configurazione. La funzione privata `_load_config()` legge `database.config` (o `database.config.example` se il primo manca) con `configparser`. Durante l'inizializzazione, viene richiamato `_ensure_database(cfg)`, che crea il database fisico se non esiste già (con un comando **CREATE DATABASE IF NOT EXISTS** eseguito da un engine temporaneo root). Questo consente di partire da zero senza dover creare manualmente lo schema. Una volta garantita l'esistenza del database, viene costruito l'engine principale usando la stringa di connessione `mysql+pymysql://`, che indica l'uso del driver *PyMySQL*$_G$.

    La classe fornisce i seguenti metodi:

    - `get_engine()` – restituisce l'engine *SQLAlchemy*$_G$ (creato una sola volta e poi riutilizzato).
    - `get_session()` – restituisce una nuova sessione *ORM*$_G$ (la **session factory** viene creata al primo utilizzo). Le sessioni si usano tipicamente in contesto **with**, così da gestire automaticamente *commit*$_G$ e rollback.
    - `init_database()` – inizializza le tabelle invocando `Base.metadata.create_all(engine)`. È richiamato dallo script `initialize_database.py` all'avvio se necessario, oppure implicitamente al primo inserimento.
    - `reset_instance()` – usato nei test per distruggere e ricreare il singleton, così da isolare correttamente i casi di prova.

- `models/question.py` – Definisce la classe `Question` (dataclass) e la logica di gestione delle domande:

  - La dataclass `Question` ha i campi `id`, `domanda`, `risposta_attesa` e `categoria`, che rappresentano una domanda nell'ambito applicativo.
  - Metodi statici principali:
    - `load_all()` – legge tutte le domande dal database tramite *SQLAlchemy*$_G$ e restituisce una lista di oggetti `Question`.

- `add(domanda, risposta_attesa, categoria, question_id=None)` – inserisce una nuova domanda. Se non viene fornito un ID, ne genera uno con **uuid4**. Crea un oggetto `QuestionORM`, lo aggiunge alla sessione e fa **commit**, restituendo l'ID.
- `update(question_id, ...)` – aggiorna i campi di una domanda esistente. Recupera l'oggetto `QuestionORM` dal database; se trovato, aggiorna i campi e salva. Restituisce **True** se la domanda esiste, **False** altrimenti.
- `delete(question_id)` – elimina una domanda in modo sicuro: prima rimuove le associazioni nella tabella `question_set_questions`, poi cancella la domanda stessa ed esegue il commit. Così si evitano riferimenti orfani nelle relazioni molti-a-molti.
- `_persist_entities(df)` – gestisce l'importazione di domande da un **DataFrame**. Rimuove duplicati interni, controlla gli ID già presenti nel database e separa righe nuove da quelle esistenti. Aggiunge avvisi per gli ID scartati e inserisce i nuovi record con `bulk_insert_mappings`. Restituisce il numero di record importati e la lista di avvisi.
- `filter_by_category(category)` – filtra le domande per categoria, se specificata, altrimenti restituisce tutte. Usa la cache (`utils.cache.get_questions`) e formattazione standardizzata (`utils.data_format_utils.format_questions_for_view`), restituendo la coppia `(DataFrame filtrato, lista di categorie)`.

▸ `QuestionImporter` – in fondo al file troviamo questa classe, che estende `ImportTemplate` e `ExportTemplate`.

Implementa:

- `parse_file(file)` – legge un file (*CSV*$_G$ o *JSON*$_G$) e restituisce un **DataFrame** di domande.
- `persist_data(df)` – inserisce i dati richiamando `Question\_persist_entities(df)`. Restituisce un dizionario con stato, numero di record importati e eventuali avvisi.
- `gather_data()` – recupera tutte le domande tramite `Question.load_all()` e le converte in **DataFrame** (per l'export).
- È disponibile un'istanza globale `question_importer = QuestionImporter()`, utilizzata dai controller.

- `models/question.py` – Definisce la classe `Question` (dataclass) e la logica di gestione delle domande:
  - ▸ La dataclass `Question` ha i campi `id`, `domanda`, `risposta_attesa` e `categoria`, che rappresentano una domanda nell'ambito applicativo.
  - ▸ Metodi statici principali:
    - `load_all()` – legge tutte le domande dal database tramite *SQLAlchemy*$_G$ e restituisce una lista di oggetti `Question`.
    - `add(domanda, risposta_attesa, categoria, question_id=None)` – inserisce una nuova domanda. Se non viene fornito un ID, ne genera uno con **uuid4**. Crea un oggetto `QuestionORM`, lo aggiunge alla sessione e fa **commit**, restituendo l'ID.
    - `update(question_id, ...)` – aggiorna i campi di una domanda esistente. Recupera l'oggetto `QuestionORM` dal database; se trovato, aggiorna i campi e salva. Restituisce **True** se la domanda esisteva, **False** altrimenti.
    - `delete(question_id)` – elimina una domanda in modo sicuro: prima rimuove le associazioni nella tabella **question__set__questions**, poi cancella la domanda stessa ed esegue il commit. Così si evitano riferimenti orfani nelle relazioni molti-a-molti.

  - `_persist_entities(df)` – gestisce l'importazione di domande da un **DataFrame**. Rimuove duplicati interni, controlla gli ID già presenti nel database e separa righe nuove da quelle esistenti. Aggiunge avvisi per gli ID scartati e inserisce i nuovi record con **bulk_insert_mappings**. Restituisce il numero di record importati e la lista di avvisi.
  - `filter_by_category(category)` – filtra le domande per categoria, se specificata, altrimenti restituisce tutte. Usa la cache (`utils.cache.get_questions`) e formattazione standardizzata (`utils.data_format_utils.format_questions_for_view`), restituendo la coppia `(DataFrame filtrato, lista di categorie)`.

- `QuestionImporter` – in fondo al file troviamo questa classe, che estende `ImportTemplate` e `ExportTemplate`.

  Implementa:

  - `parse_file(file)` – legge un file (*CSV*$_G$ o *JSON*$_G$) e restituisce un **DataFrame** di domande.
  - `persist_data(df)` – inserisce i dati richiamando `Question._persist_entities(df)`. Restituisce un dizionario con stato, numero di record importati e eventuali avvisi.
  - `gather_data()` – recupera tutte le domande tramite `Question.load_all()` e le converte in **DataFrame** (per l'export).
  - È disponibile un'istanza globale `question_importer = QuestionImporter()`, utilizzata dai controller.

- `models/question_set.py` – Implementa la classe `QuestionSet` e la logica di gestione dei set di domande:

  - Dataclass `QuestionSet`: contiene `id`, `name` e `questions` (lista di ID domanda).
  - Metodi statici principali:
    - `load_all()` – legge tutti i set dal database e li converte in oggetti `QuestionSet`.
    - `create(name, question_ids)` – crea un nuovo set con ID generato via *UUID*$_G$, recupera gli oggetti domanda corrispondenti, li associa al nuovo `QuestionSetORM`, salva e restituisce l'ID.
    - `update(set_id, name, question_ids)` – aggiorna un set esistente. Se presente, aggiorna il nome e, se specificata, la lista di domande. *SQLAlchemy*$_G$ sincronizza automaticamente la tabella di associazione.
    - `delete(set_id)` – elimina il set indicato (le righe nella tabella di associazione vengono rimosse grazie al **cascade**).
    - `_resolve_question_ids(questions_in_set_data, current_questions_df)` – funzione di supporto usata in fase di import. Gestisce domande definite con ID o con dati completi.
      - Se l'elemento è un dizionario con testo e risposta, prova a usare l'ID indicato: se non esiste, aggiunge la domanda tramite `add_question_if_not_exists`; altrimenti lo considera già presente.
      - Se l'elemento è solo un ID, verifica la sua esistenza: se non c'è e mancano dettagli, produce un avviso e lo scarta.

      Restituisce la lista finale di ID, il **DataFrame** aggiornato e i conteggi di nuove domande, esistenti trovate e avvisi.
    - `_persist_entities(sets_data, current_questions_df, current_sets_df)` – esegue l'importazione dei set da una lista di dati già interpretata.

Per ogni set:

- verifica la validità (nome non vuoto, domande presenti);
- scarta eventuali duplicati di nome con un avviso;
- risolve gli ID delle domande tramite `_resolve_question_ids`;
- crea il set e aggiorna i contatori di importazione;
- alla fine restituisce un oggetto `PersistSetsResult` contenente set e domande aggiornati, contatori complessivi e lista avvisi.

- `models/test_result.py` – Contiene la classe `TestResult` e la logica per la gestione dei risultati dei test:
  - Dataclass `TestResult`: campi `id`, `set_id`, `timestamp`, `results` (dict). Il campo `_test` = False evita che pytest interpreti la classe come test case a causa del nome.
  - Metodi principali:
    - `load_all()` – legge tutti i `TestResultORM` dal database e li converte in oggetti `TestResult`.
    - `load_all_df()` – costruisce un **DataFrame** completo dei risultati (simile a `get_questions` in cache), decorato con `@lru_cache(maxsize=1)` per caching globale.
    - `refresh_cache()` – pulisce la cache di `load_all_df` e ricostruisce il **DataFrame** aggiornato.
    - `_persist_entities(imported_df)` – importa risultati da **DataFrame** evitando duplicati:
      - ottiene gli `existing_ids` dal database;
      - filtra le righe nuove tramite `filter_new_rows`;
      - se ci sono righe nuove, le unisce a quelle esistenti, le converte in `TestResult` e salva col metodo `save(results_list)`;
      - restituisce il numero di nuovi record aggiunti.
    - `save(results: List[TestResult])` – sincronizza una lista completa di risultati con il database:
      - elimina i record esistenti non più presenti nella lista;
      - aggiorna i record esistenti o aggiunge quelli nuovi;
      - commit finale.

      Questa strategia permette di fondere correttamente i dati importati con il database.
    - `add(set_id, results_data)` – aggiunge un singolo risultato generando un nuovo UUID, salva su database e restituisce l'ID.
    - `add_and_refresh(set_id, results_data)` – combina `add` e `refresh_cache` per rendere subito disponibile il nuovo risultato.
    - `calculate_statistics(questions_results: Dict[str, Dict])` – calcola metriche aggregate per un insieme di risultati:
      - se il dizionario è vuoto, restituisce 0% per tutto;
      - altrimenti, calcola punteggi medi, metriche per domanda e metriche radar (similarity, correctness, completeness);
      - restituisce un dizionario con **avg_score**, `per_question_scores` e `radar_metrics`.

Questo metodo viene usato in `test_controller.run_test` per arricchire i risultati prima del salvataggio.

- `TestResultImporter`: gestisce import/export dei risultati:
  - `parse_file(file)` – legge il file JSON e restituisce un **DataFrame**.
  - `persist_data(df)` – richiama `_persist_entities(df)` e, se vengono aggiunti risultati, aggiorna la cache. Restituisce un messaggio con il numero di risultati importati.
  - `gather_data()` – restituisce il **DataFrame** completo di tutti i risultati, pronto per l'export.
  - Istanza globale: `test_result_importer = TestResultImporter()`.

- `models/api_preset.py` – Implementa la classe `APIPreset` per i preset API:
  - Dataclass `APIPreset`: campi `id`, `name`, `provider_name`, `endpoint`, `api_key`, `model`, `temperature`, `max_tokens`.
  - Metodi principali:
    - `load_all()` – legge tutti i preset dal database e li converte in `APIPreset`.
    - `save(presets: List[APIPreset])` – sincronizza la lista di preset con il database:
      - elimina preset non più presenti;
      - aggiorna quelli esistenti o aggiunge nuovi;
      - commit finale.
    - `delete(preset_id)` – elimina il preset con l'ID indicato se esiste.

      Non è previsto il cascata perché i preset non hanno relazioni figlie; vengono solo referenziati nei `TestResult.results`.
  - Non è presente un importer per i preset, perché la gestione di chiavi *API*$_G$ e parametri sensibili è manuale e non parte del *MVP*$_G$.

I modelli forniscono un'interfaccia coerente per persistere e leggere i dati dal *database*$_G$.

Segue uno schema comune:

- `load_all()` per leggere tutti i record di un tipo.
- Metodi **save/sync** per scritture in blocco (`TestResult`, `APIPreset`).
- Metodi `add/update/delete` per scritture singole (`Question`, `QuestionSet`).
- Classi Importer/Exporter generiche per le entità principali, che riutilizzano metodi interni e utility di lettura/scrittura file.

## 5.5. Utilità (Utils)

La cartella `utils/` contiene vari moduli di utilità generale utilizzati trasversalmente:

- `utils/cache.py` – Gestisce il caching in memoria con `functools.lru_cache`:
  - Funzioni `get_questions`, `get_question_sets`, `get_api_presets` decorate con `@lru_cache(maxsize=1)` caricano i dati tramite i rispettivi `Model.load_all()`, li convertono in lista di dict (`dataclasses.asdict`) e poi in `DataFrame` pandas con colonne ordinate. Mantengono solo l'ultima versione caricata; successive chiamate restituiscono la copia cache fino a invalidazione.

- Funzioni `refresh_questions`, `refresh_question_sets`, `refresh_api_presets` puliscono la cache (`cache_clear()`) e richiamano subito la funzione `get_` corrispondente, restituendo il `DataFrame` aggiornato.
- Funzioni `get_results` e `refresh_results` richiamano i metodi di `TestResult`, che già gestiscono il caching interno.

L'uso di questo modulo permette accesso centralizzato e performante ai dati in sola lettura: la prima chiamata popola la cache, mentre le successive restituiscono i dati in memoria. Dopo operazioni di scrittura (add/update/delete/import) si richiamano i metodi `refresh_*` per mantenere coerenza.

- `utils/openai_client.py` – Incapsula la creazione di client per API OpenAI e compatibili:
  - Costanti `DEFAULT_MODEL = "gpt-4o"` e `DEFAULT_ENDPOINT = "https://api.openai.com/v1"`.
  - Classe `ClientCreationError` – eccezione personalizzata se la creazione del client fallisce.
  - Funzione `get_openai_client(api_key, base_url=None)` – crea un client OpenAI impostando chiave e endpoint; se `base_url` è `None` o `"custom"` usa `DEFAULT_ENDPOINT`. Ritorna il client o lancia `ClientCreationError` se la chiave manca.
  - Funzione `get_available_models_for_endpoint(provider_name, endpoint_url=None, api_key=None)` – elenca i modelli disponibili per un provider/endpoint:
    - Per provider `"Personalizzato"` prova a creare il client con le credenziali fornite e recupera i modelli disponibili, filtrando embedding e includendo parole chiave come chat, instruct, gpt, claude, grok, o nomi complessi. Se la lista resta vuota, restituisce `[DEFAULT_MODEL]`. Gestisce eccezioni restituendo messaggi di errore nella lista.
    - Per altri provider (es. `"OpenAI"`, `"Anthropic"`) restituisce `[DEFAULT_MODEL]` come placeholder.

  Questo modulo è usato dai controller per testare preset o ottenere client per richieste di completamento, anche su endpoint custom.
- `utils/file_reader_utils.py` – Funzioni per leggere file CSV/JSON:
  - `read_questions(file)` – legge CSV/JSON di domande, restituisce un `DataFrame` con colonne `id, domanda, risposta_attesa, categoria`.
  - `read_question_sets(file)` – legge set di domande, restituisce `List[Dict]` con struttura `{"name": ..., "questions": [...]}`.
  - `read_test_results(file)` – legge file JSON di risultati, restituisce un `DataFrame` compatibile con `TestResult._persist_entities`.
  - `filter_new_rows(df, existing_ids)` – filtra righe con ID non presenti in `existing_ids` e restituisce `(new_rows_df, count)`.
- `utils/file_writer_utils.py` – Scrittura di dataset su file:
  - `write_dataset(data, destination)` – se `destination` è stringa apre il file e scrive; se file object scrive direttamente. Se `data` è `DataFrame` sceglie formato CSV/JSON in base all'estensione, altrimenti scrive JSON da `list/dict`.
- `utils/data_format_utils.py` – Helper per formattazione dati:
  - `format_questions_for_view(questions_df)` – arricchisce il `DataFrame` domande con colonne normalizzate e costruisce mapping `question_map` e lista di categorie distinte.
  - `build_questions_detail(question_map, q_ids)` – costruisce lista di dizionari con dettagli delle domande corrispondenti a una lista di ID, usata per anteprime set e filtri categorie.

- `utils/startup_utils.py` – Funzioni da eseguire all'avvio:
  - `setup_logging()` – configura logging (livello, formato, output su console/file). Usata in `app.py` e `initialize_database.py`.
  - Altre utilità di startup possono includere caricamento variabili d'ambiente o configurazioni extra.
- `utils/import_template.py` – Template base per import:
  - Classe astratta `ImportTemplate` impone l'implementazione di `parse_file(file)` e `persist_data(parsed)` nelle sottoclassi (`QuestionImporter`, `QuestionSetImporter`, `TestResultImporter`).
  - Fornisce `import_from_file(file)` che esegue il flusso completo parse → persist, loggando errori e lanciando `ValueError` se qualcosa va storto.
- `utils/export_template.py` – Template base per export:
  - Classe `ExportTemplate` impone `gather_data()` nelle sottoclassi e offre `export_to_file(destination)`, che chiama `utils.file_writer_utils.write_dataset` per scrivere i dati su file.
  - Le sottoclassi implementano `gather_data` per raccogliere dal *Database*$_{\text{G}}$ entità in formato compatibile con *JSON*$_{\text{G}}$/*CSV*$_{\text{G}}$.

In generale, l'architettura di modelli e utilità segue un design modulare e riutilizzabile, dove operazioni comuni di import/export usano template generici e le operazioni ripetitive (caricamento record, salvataggio batch) sono centralizzate nei metodi statici dei modelli.

# 6. Flusso di dati e interazione API

In questa sezione descriviamo come i dati fluiscono attraverso il sistema – dalla persistenza nel *database*$_G$, al caching, fino all'uso nelle interfacce – e come avviene l'interazione con le *API*$_G$ esterne (*OpenAI*$_G$ e simili) per la generazione e valutazione.

## 6.1. Persistenza e caching dei dati locali

Quando l'utente inserisce o modifica dati (domande, set, preset), i controller invocano i metodi del modello che eseguono l'operazione sul database *MySQL*$_G$ (tramite *SQLAlchemy*$_G$) e subito dopo aggiornano la rispettiva cache in-memory. Ad esempio, se viene aggiunta una domanda, `Question.add(...)` inserisce la riga nella tabella questions e quindi question_controller.add_question richiama `refresh_questions()` che pulisce la *cache LRU*$_G$ e ricarica tutte le domande aggiornate. Così, la nuova domanda comparirà immediatamente nell'elenco in *UI*$_G$ senza bisogno di refresh manuali o *query*$_G$ extra al database. Lo stesso avviene per set e preset: ogni volta che si chiamano `create_set`, `update_set`, `save_preset`, `delete_preset`, ecc., i controller invocano il refresh della rispettiva cache subito dopo l'operazione sul database. Questo modello di cache aside assicura coerenza forte tra database e memoria: i dati in memoria vengono considerati fonte di verità solo finché non cambia qualcosa; appena cambia, vengono ricaricati dalla fonte persistente.

## 6.2. Lettura dei dati per la UI

Le view raramente accede direttamente al database; invece chiede ai controller i DataFrame preparati. Questi DataFrame provengono dalla cache (popolata come detto tramite i modelli). Ciò minimizza i tempi di risposta e alleggerisce il carico sul database, specialmente in contesti come la visualizzazione di liste (domande, set, risultati) dove potenzialmente l'utente potrebbe applicare filtri o refresh frequenti. Un diagramma semplificato del flusso potrebbe essere:

```
UI (Streamlit) --> Controller (get_x) --> Cache (DataFrame in mem)
                                      --> [se scaduta] Modello.load_all() -> database
UI (azione CRUD) --> Controller (op) --> Modello (scrive su database) -> database
                                    --> Cache.refresh() (ricarica in mem)
```

In altre parole, la prima lettura passa per il database, le successive sono servite dalla cache fino a invalidazione.

## 6.3. Flusso di esecuzione di un test LLM

Questa è la parte più articolata, che coinvolge sia dati locali (domande, risultati) sia chiamate esterne:

1. L'utente dalla *UI*$_G$ lancia un test selezionando set e preset. La view chiama

```python
test_controller.run_test(set_id, set_name, question_ids, gen_config, eval_config).
```

2. Il controller preleva localmente tutte le domande pertinenti (cache di domande) e costruisce `= {}` vuoto.
3. Per ogni domanda:
   - Prende il testo e la risposta attesa dal database (cache) – questi dati risiedono localmente.

- Contatta il servizio esterno di generazione (ad es. *endpoint OpenAI*$_G$) usando i parametri del preset di generazione. Avviene una chiamata *HTTP*$_G$ POST /v1/chat/completions (nel caso di OpenAI) con payload JSON contenente il prompt utente costruito e i parametri (model, temperature,

max_tokens). Questa richiesta transita attraverso internet (o rete locale se endpoint custom) al *server*$_G$ del provider, il quale esegue il modello *LLM*$_G$ e restituisce una risposta.

- La risposta del provider viene ricevuta dal nostro client Python (openai_client), tipicamente anch'essa in JSON (con l'oggetto choices contenente il messaggio del modello). La funzione `_generate_answer_` estrae il testo dal JSON e lo passa al controller.
- Subito dopo, viene contattato il servizio di valutazione (che potrebbe essere lo stesso provider ma potenzialmente anche un altro) con un'altra richiesta *API*$_G$: il prompt di valutazione (inclusivo di domanda, risposta attesa e risposta generata) viene inviato ad esempio a OpenAI ChatCompletions. Qui avviene un altro round di inferenza su *server*$_G$ remoto: il modello riceve il prompt con le istruzioni di valutare e produrre JSON con punteggi. Il risultato, idealmente un blob JSON testuale, viene restituito e il nostro codice `evaluate_answer` esegue un parse (usando json.loads in Python) per ottenere i valori strutturati. In caso di errori (ad es. il modello non rispetta il formato e risponde con testo libero), viene lanciata un'eccezione e gestita come detto inserendo un errore nei dati.
- Non appena per una domanda si hanno risposta generata ed esito valutazione, questi vengono accorpati in un risultato parziale e inseriti nel dict results locale.

4. Terminato il loop, il controller calcola i valori aggregati (questa parte è locale, pura computazione in Python sui dati raccolti) e prepara il dict finale contenente tutto (dati grezzi e metriche aggregate).
5. Il controller chiama il modello `_TestResult.add_and_refresh_` per salvare il risultato nel database: qui c'è un'unica transazione al database che fa un INSERT del record nella tabella test_results con i dati JSON (notare: i campi JSON includono possibili stringhe lunghe – risposte LLM e spiegazioni – e numeri, quindi l'oggetto JSON può essere di dimensioni consistenti, ma MySQL con colonna JSON li gestisce).
6. Dopo il commit sul database, `refresh_cache` aggiorna la cache dei risultati per includere subito il nuovo test.
7. Il controller ritorna alla *UI*$_G$ i dati; la UI può decidere di mostrare il punteggio medio immediatamente o di reindirizzare l'utente alla pagina di visualizzazione per dettagli.

Una caratteristica importante è la gestione delle eccezioni e casi limite in questo flusso:

- Se la chiamata per generare risposte fallisce (es. *API*$_G$ non risponde), il sistema non interrompe l'intero test. Invece, segna solo quella domanda come errore (score 0) e prosegue con le altre. Questo è fondamentale per robustezza, soprattutto quando si fanno batch di 10-20 chiamate *LLM*$_G$: uno o due fallimenti non devono invalidare tutto.
- Analogamente, se la valutazione automatica fallisce, assegna 0 a quella domanda ma il test continua.
- Se addirittura tutto il processo `run_test` lanciasse un'eccezione imprevista (magari un bug interno), viene catturata al termine e si ritorna semplicemente un dict vuoto, evitando crash dell'app. La UI potrebbe notificare l'utente che il test non è riuscito.

## 6.4. Gestione dati binari e file

Importante notare che l'app trasferisce alcuni dati tramite file upload/download:

- Le funzioni di import leggono file *CSV/JSON*$_G$ forniti dall'utente: questi file vengono caricati tramite Streamlit (`st.file_uploader`) in un stream in-memory (`uploaded_file`). Il controller import prende questo file-like object e lo passa all'importer `import_from_file`, il quale lo legge con pandas o JSON a seconda del formato. Durante la lettura, vengono interpretati i dati e generati

eventuali errori (ad es. CSV con colonne errate può lanciare errori *Pandas*$_G$, che vengono catturati dall'ImportTemplate e convertiti in ValueError).

- Le funzioni di export generano stringhe JSON e tramite *Streamlit*$_G$ st.download_button ne permettono il download. Non si appoggiano su un servizio esterno per questo; l'operazione avviene tutta lato server e il file viene trasferito all'utente tramite la funzionalità di Streamlit.

## 6.5. Transazioni database

Il codice utilizza il contesto with session: offerto da *SQLAlchemy*$_G$. Questo significa che ogni operazione di scrittura (`add/update/delete`) è incapsulata in una transazione che viene committata alla fine del blocco with. Se un'eccezione avviene dentro, SQLAlchemy effettua rollback automatico quando esce dal contesto senza commit. Dato che i modelli in molti casi catturano le eccezioni (per poi magari rilanciare ValueError), la transazione comunque viene gestita. Un caso particolare è in APIPreset.save e `TestResult.save` dove si fanno più operazioni: qui è importante che tutte le operazioni di sync vengano completate prima del commit, in modo che il database rimanga consistente con la lista in ingresso. L'uso di sessione in with garantisce che se qualcosa va storto a metà, niente venga scritto parzialmente.

## 6.6. Concurrency e thread-safety

Streamlit di base esegue l'app in un solo thread per utente, quindi non dovrebbero esserci problemi di scritture concorrenti. Tuttavia, DatabaseEngine è implementato come *thread-safe*$_G$ (lock sul singleton e sui lock engine/session) per precauzione e anche per poter essere eventualmente usato in contesti multi-thread (Streamlit in alcune configurazioni avanzate può far girare più sessioni utenti parallele). Anche le *cache LRU*$_G$ sono thread-safe in Python (il lock GIL le protegge durante l'update), e in ogni caso sono invalidate immediatamente prima di refresh per evitare condizioni di lettura sporcate.

## 6.7. Interazione con servizi LLM esterni

Come visto finora, avviene attraverso il modulo `openai_client` che incapsula la chiave *API*$_G$ e *endpoint*$_G$. Da un punto di vista di sicurezza e performance:

- Le *chiavi API*$_G$ inserite dall'utente vengono conservate nel *database*$_G$ (colonna api_key in chiaro) e richiamate al bisogno. Idealmente, per maggiore sicurezza, si potrebbero cifrare con la libreria Cryptography prima di salvarle e decifrare al volo; il codice attuale però non implementa ciò.
- Le chiamate ai modelli sono potenzialmente lente (centinaia di ms o alcuni secondi). L'app non fa altro nel frattempo se non attendere. Non è implementato un sistema asincrono: ogni chiamata è sincrona bloccante. Dato che Streamlit esegue codice server-blocking per singola sessione utente, questo significa che durante l'esecuzione di un test l'interfaccia di quell'utente rimane occupata finché non termina (si potrebbe mostrare un spinner). Tuttavia, il design *MVP*$_G$ privilegia la semplicità: in un ambiente con volumi maggiori, si potrebbe voler eseguire le valutazioni in parallelo (ad esempio multi-threading sulle domande) o delegare a servizi di coda, ma questo andrebbe oltre lo scope attuale.
- La formattazione richiesta all'LLM valutatore (risposta in JSON) è un meccanismo critico: in genere i modelli di linguaggio potrebbero non seguire perfettamente le istruzioni di formato. Per mitigare, nel prompt viene mostrato un esempio di risposta JSON in modo da guidare il modello. Nonostante ciò, il codice è pronto a gestire un JSONDecodeError qualora il modello restituisse output non JSON (lo logga e lancia eccezione). In produzione, potrebbe essere utile affinare il prompt o usare funzioni di correzione (regex per estrarre numeri se JSON malformato), ma per l'MVP si è scelto di far fallire esplicitamente così da informare l'utente dello scenario.

## 6.8. Notifiche all'utente

Il flusso di dati verso l'utente finale avviene tramite componenti Streamlit:

- Messaggi di successo/errore sono mostrati usando `st.success, st.error, st.warning` in varie pagine (ad es. dopo import, dopo test connessione API). Questo è integrato con lo `st.session_state` per garantire che i messaggi compaiano una sola volta (vengono reset dopo visualizzazione).
- I risultati e grafici sono aggiornati reattivamente perché Streamlit ricalcola la pagina ad ogni interazione. Ad esempio, quando `st.session_state.results` viene modificato in **Visualizza Risultati** (per import, o per esecuzione nuovo test), la pagina si rerenderizza mostrando i nuovi dati.

# 7. Stato dei requisiti funzionali

Vengono di seguito riportati i requisiti funzionali individuati durante la fase di analisi. Per ognuno di essi vengono forniti:

- **Codice**: identificativo;
- **Descrizione**;
- **Tipo**: priorità;
- **Stato**: soddisfatto/non soddisfatto.

Per maggiori dettagli su Codice e Tipo si rimanda alla sezione *Requisiti* del documento *Analisi dei Requisiti v2.0.0*.

| Codice | Descrizione | Tipo | Stato |
|---|---|---|---|
| RFO01 | L'utente deve poter visualizzare la lista delle domande | Obbligatorio | Soddisfatto |
| RFO02 | L'utente deve poter visualizzare una singola domanda | Obbligatorio | Soddisfatto |
| RFO03 | L'utente deve poter modificare una singola domanda | Obbligatorio | Soddisfatto |
| RFO04 | L'utente deve poter modificare una risposta attesa | Obbligatorio | Soddisfatto |
| RFO05 | L'utente deve poter eliminare una singola domanda | Obbligatorio | Soddisfatto |
| RFD06 | L'utente vuole poter annullare l'operazione di eliminazione di una singola domanda | Desiderabile | Soddisfatto |
| RFD07 | L'utente deve poter filtrare le domande in base alla categoria di appartenenza | Desiderabile | Soddisfatto |
| RFO08 | L'utente deve essere avvisato nel caso in cui l'aggiunta manuale della domanda non sia corretta e/o completa | Obbligatorio | Soddisfatto |
| RFO09 | L'utente deve poter aggiungere manualmente una domanda | Obbligatorio | Soddisfatto |
| RFO10 | L'utente deve poter aggiungere domande e relative risposte attese tramite un file CSV | Obbligatorio | Soddisfatto |
| RFO11 | L'utente deve poter aggiungere domande e relative risposte attese tramite un file JSON | Obbligatorio | Soddisfatto |
| RFO12 | L'utente vuole poter essere notificato nel caso in cui il file di importazione delle domande non sia corretto | Obbligatorio | Soddisfatto |
| RFO13 | L'utente deve poter visualizzare la lista dei set di domande salvati nel sistema | Obbligatorio | Soddisfatto |
| RFO14 | L'utente deve poter visualizzare un singolo set di domande | Obbligatorio | Soddisfatto |
| RFO15 | L'utente deve poter modificare un set di domande | Obbligatorio | Soddisfatto |
| RFO16 | L'utente deve poter eliminare un set di domande | Obbligatorio | Soddisfatto |
| RFD17 | L'utente vuole poter annullare l'operazione di eliminazione di un set di domande | Desiderabile | Soddisfatto |
| RFD18 | L'utente deve poter filtrare i set di domande in base alla categoria di appartenenza | Desiderabile | Soddisfatto |
| RFP19 | L'utente deve poter aggiornare manualmente un set di domande | Opzionale | Soddisfatto |
| RFO20 | L'utente vuole poter essere notificato nel caso in cui non inserisca correttamente il nome del set di domande | Obbligatorio | Soddisfatto |

| | | | |
|---|---|---|---|
| RFO21 | L'utente deve poter aggiungere un set di domande da file JSON | Obbligatorio | Soddisfatto |
| RFO22 | L'utente deve poter aggiungere un set di domande da file CSV | Obbligatorio | Soddisfatto |
| RFO23 | L'utente vuole poter essere notificato nel caso in cui il file contenente il set da aggiungere non sia valido | Obbligatorio | Soddisfatto |
| RFO24 | L'utente deve poter accedere alla sezione di configurazione dell'API e creare un preset | Obbligatorio | Soddisfatto |
| RFO25 | L'utente deve poter essere notificato nel caso in cui il nome del preset non sia stato inserito | Obbligatorio | Soddisfatto |
| RFP26 | L'utente deve poter effettuare un test di connessione dopo aver compilato i campi necessari per la creazione di un preset API | Opzionale | Soddisfatto |
| RFP27 | L'utente deve poter essere notificato nel caso in cui l'esito del test sia negativo | Opzionale | Soddisfatto |
| RFO28 | L'utente deve poter visualizzare la lista dei preset che sono stati configurati | Obbligatorio | Soddisfatto |
| RFO29 | L'utente deve poter modificare un preset precedentemente configurato | Obbligatorio | Soddisfatto |
| RFO30 | L'utente deve poter eliminare un preset precedentemente configurato | Obbligatorio | Soddisfatto |
| RFO31 | Il sistema deve poter inviare domande a un LLM esterno tramite API e riceverne le risposte. | Obbligatorio | Soddisfatto |
| RFO32 | L'utente deve poter visualizzare tutti i preset salvati all'interno della sezione di esecuzione test | Obbligatorio | Soddisfatto |
| RFO33 | L'utente deve poter visualizzare tutti i set di domande salvati all'interno della sezione di esecuzione test | Obbligatorio | Soddisfatto |
| RFO34 | L'utente deve poter visualizzare il punteggio o il giudizio qualitativo assegnato dal sistema in seguito al confronto tra la risposta attesa e la risposta generata dall'LLM | Obbligatorio | Soddisfatto |
| RFO35 | L'utente deve poter visualizzare i risultati ottenuti dal confronto tra risposte attese e risposte dell'LLM | Obbligatorio | Soddisfatto |
| RFO36 | L'utente deve poter visualizzare i risultati ottenuti dal confronto tra risposte attese e risposte dell'LLM relativi ad una singola domanda | Obbligatorio | Soddisfatto |
| RFO37 | L'utente deve poter visualizzare le metriche applicate ad una singola risposta | Obbligatorio | Soddisfatto |
| RFO38 | L'utente deve poter salvare e consultare i risultati del test (risposte, valutazioni e metadati) | Obbligatorio | Soddisfatto |
| RFP39 | L'utente deve poter scaricare i risultati di un test | Opzionale | Soddisfatto |
| RFP40 | L'utente deve poter caricare nel sistema un file di risultati di test effettuati precedentemente | Opzionale | Soddisfatto |
| RFO41 | L'utente vuole poter essere notificato nel caso in cui il caricamento del file contenete i risultati non vada a buon fine | Obbligatorio | Soddisfatto |
| RFO42 | L'utente deve poter visualizzare i test effettuati precedentemente | Obbligatorio | Soddisfatto |

| | | | |
|---|---|---|---|
| RFP43 | L'utente deve poter visualizzare tutti i dettagli di un determinato test effettuati precedentemente | Opzionale | Soddisfatto |
| RFD44 | L'utente deve poter filtrare lo storico dei test in base a diversi criteri | Desiderabile | Soddisfatto |
| RFP45 | L'utente deve poter confrontare due diverse esecuzioni di test | Opzionale | Soddisfatto |
| RFP46 | L'utente deve poter confrontare due diverse risposte LLM per una singola domanda specifica | Opzionale | Soddisfatto |