

# Università degli Studi di Napoli Federico II

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica



## **ELABORATO DI AI SYSTEM ENGINEERING**

**HEALTHGATE: UN SISTEMA LLM PER IL SUPPORTO DECISIONALE E LA RIDUZIONE DEL  
SOVRAFFOLLAMENTO NEI PRONTO SOCCORSO**

*Prof.re Roberto Pietrantuono*

A.A. 2024-25

Studenti:

Campanella Alessandro M63001697

Castaldi Rita M63001667

Di Fraia Simone M63001678

# Indice

<b>BUSINESS IDEA .....</b>	<b>1</b>
<b>REQUIREMENT ANALYSIS.....</b>	<b>1</b>
STAKEHOLDERS .....	1
REQUISITI FUNZIONALI .....	1
REQUISITI NON FUNZIONALI .....	2
CASI D'USO .....	3
USE CASE DIAGRAM .....	7
<b>SYSTEM DESIGN .....</b>	<b>7</b>
ARCHITETTURA A MICROSERVIZI .....	7
MODELLAZIONE COMPLESSIVA DEL SISTEMA .....	8
SEQUENCE DIAGRAM .....	9
ARCHITETTURA LOGICA E ORGANIZZAZIONE DEI COMPONENTI .....	10
IMPLEMENTAZIONE .....	12
FRONTEND .....	12
MICROSERVICES .....	14
API GATEWAY .....	14
AUTHENTICATION SERVICE.....	20
INGESTION SERVICE .....	25
DECISION ENGINE SERVICE .....	31
AGGREGATOR SERVICE .....	39
REPORT MANAGEMENT SERVICE .....	42
<b>CONTINUOUS MONITORING E TESTING .....</b>	<b>44</b>
TESTING DEL SISTEMA.....	44
TESTING FUNZIONALE .....	44
TESTING DI INTEGRAZIONE .....	44
TESTING DEL DATABASE .....	45
TESTING DEL FRONTEND .....	45
TESTING DEL MODELLO DECISIONALE .....	45
CONTINUOUS MONITORING.....	47
LOCAL DEPLOYMENT CON DOCKER .....	47
RISULTATI DEL DEPLOYMENT .....	48
<b>CONCLUSIONI .....</b>	<b>49</b>

## Business Idea

Uno dei problemi più rilevanti del sistema sanitario odierno è rappresentato dall'elevato numero di accessi impropri alle strutture di emergenza. Molti pazienti si rivolgono al pronto soccorso anche in presenza di condizioni cliniche non gravi, spesso per mancanza di informazioni, per timore o per difficoltà ad accedere ad altre forme di assistenza. Questo fenomeno determina un sovraccarico del sistema, con conseguenti rallentamenti nella gestione dei casi realmente urgenti, incremento dei tempi di attesa e aumento dei costi complessivi, oltreché un aggravio del carico di lavoro per il personale sanitario. Tale fenomeno incide negativamente sia sulla qualità dell'assistenza fornita ai pazienti in condizioni critiche, sia sulla sostenibilità complessiva del sistema sanitario.

Dall'altro lato, in molte situazioni, i pazienti si trovano in condizioni di incertezza circa la gravità dei propri sintomi. Tale incertezza può portare da un lato a sottovalutare situazioni cliniche che richiederebbero invece un intervento tempestivo, dall'altro a ricorrere inutilmente al pronto soccorso per disturbi lievi, come citato precedentemente.

Un sistema in grado di analizzare i sintomi descritti dal paziente e di fornire un'indicazione chiara e immediata può contribuire a ridurre questa incertezza, aumentando la consapevolezza del cittadino e promuovendo un comportamento più appropriato e responsabile nell'uso delle strutture sanitarie.

## Requirement Analysis

### Stakeholders

Gli **stakeholder** sono tutte le figure che hanno un interesse o un impatto nell'uso e nello sviluppo del sistema. Nel nostro contesto, gli stakeholder sono due:

- I **Pazienti** che rappresentano i principali beneficiari, in quanto ricevono supporto decisionale e un orientamento sull'eventuale necessità di recarsi al pronto soccorso.
- **L'Operatore Sanitario** del pronto soccorso che può utilizzare i report clinici generati dal sistema per velocizzare la presa in carico del paziente.

### Requisiti Funzionali

Di seguito si riportano i requisiti funzionali, i quali descrivono le funzionalità e i servizi offerti dal sistema:

1. Il sistema deve essere in grado di consentire ai pazienti e al personale sanitario di potersi registrare sulla piattaforma.
2. Il sistema deve prevedere meccanismi di autenticazione sicura per i *pazienti* e per il *personale sanitario*.

3. Il sistema deve essere in grado di consentire ai *pazienti* di registrare i propri sintomi mediante input vocale.
4. Il sistema deve consentire ai *pazienti* di riportare i propri sintomi in formato testuale.
5. Il sistema deve essere in grado di elaborare le informazioni fornite dal paziente e determinare se sia necessario recarsi al Pronto Soccorso, basandosi su criteri clinici predefiniti.
6. Il sistema deve generare automaticamente un report clinico relativo alle condizioni del paziente.
7. Il sistema deve permettere all'*operatore sanitario* di reperire e consultare i report clinici dei pazienti.
8. Il sistema deve permettere agli utenti di accedere alla cronologia completa delle consultazioni effettuate in precedenza.
9. Il sistema deve permettere all'*operatore sanitario* di aggiornare i report clinici con informazioni relative alle cure somministrate e al piano terapeutico.

## Requisiti Non Funzionali

Di seguito si riportano i requisiti non funzionali che definiscono o limitano le proprietà del sistema:

1. Il sistema deve garantire la memorizzazione sicura dei dati sensibili dei pazienti.
2. Il sistema deve determinare la scelta relativa all'accesso al Pronto Soccorso in tempi relativamente bassi, garantendo reattività.
3. Il modello deve essere leggero (*lightweight*) per consentire l'esecuzione delle funzionalità anche su dispositivi mobili o con risorse limitate.
4. Il modello deve fornire spiegazioni precise e comprensibili riguardo le decisioni prese dal sistema.
5. Il sistema deve essere scalabile per supportare un numero crescente di utenti e dati clinici senza degrado delle prestazioni.
6. Il sistema deve essere facilmente manutenibile ed estendibile per consentire aggiornamenti di documenti clinici ufficiali o modifiche funzionali.
7. Il sistema deve essere robusto alla presenza di drift nei dati clinici o nelle abitudini degli utenti.

## Casi d'uso

Caso d'uso: Registrazione sintomi e classificazione tramite input vocale	
<b>Attore primario</b>	Paziente
<b>Attore secondario</b>	-
<b>Descrizione</b>	Il paziente registra i propri sintomi descrivendoli al microfono del dispositivo. Il sistema trascrive ed elabora l'input vocale salvando le informazioni.
<b>Pre-condizioni</b>	Il paziente è autenticato nel sistema
<b>Sequenza di eventi principali</b>	<ol style="list-style-type: none"> <li>1. Il paziente accede alla funzione "Registra sintomi vocali".</li> <li>2. Il sistema avvia la registrazione audio.</li> <li>3. Il paziente descrive i sintomi a voce.</li> <li>4. Il sistema converte il parlato in testo.</li> <li>5. Il sistema corregge errori grammaticali e normalizza nella semantica clinica.</li> <li>6. Il sistema memorizza i sintomi trascritti o corretti.</li> <li>7. Il sistema elabora automaticamente i sintomi e genera una valutazione clinica.</li> <li>8. Il sistema crea un report clinico sulle condizioni del paziente e lo archivia nel fascicolo elettronico.</li> <li>9. Il paziente riceve conferma dell'avvenuto salvataggio.</li> </ol>
<b>Post-condizioni</b>	I sintomi vengono trascritti, eventualmente corretti, valutati e salvati nel profilo del paziente.
<b>Sequenza di eventi alternativi</b>	-

Caso d'uso: Registrazione sintomi e classificazione tramite input testuale	
<b>Attore primario</b>	Paziente
<b>Attore secondario</b>	-
<b>Descrizione</b>	Il paziente registra i propri sintomi mediante un input testuale. Il sistema elabora l'input salvando le informazioni.
<b>Pre-condizioni</b>	Il paziente è autenticato nel sistema
<b>Sequenza di eventi principali</b>	<ol style="list-style-type: none"> <li>1. Il paziente accede alla funzione "Trascrivi sintomi".</li> <li>2. Il paziente descrive i sintomi mediante input testuale.</li> <li>3. Il paziente revisiona e conferma l'elaborazione del sistema.</li> </ol>

	<ol style="list-style-type: none"> <li>4. Il sistema memorizza i sintomi.</li> <li>5. Il sistema elabora automaticamente i sintomi e genera una valutazione clinica.</li> <li>6. Il sistema crea un report clinico sulle condizioni del paziente e lo archivia nel fascicolo elettronico.</li> <li>7. Il paziente riceve conferma dell'avvenuto salvataggio.</li> </ol>
<b>Post-condizioni</b>	I sintomi vengono valutati e salvati nel profilo del paziente
<b>Sequenza di eventi alternativi</b>	-

<b>Caso d'uso:</b>	<b>Registrazione</b>
<b>Attore primario</b>	Paziente, Operatore sanitario
<b>Attore secondario</b>	-
<b>Descrizione</b>	L'utente accede al sistema per potersi registrare.
<b>Pre-condizioni</b>	Nessuna
<b>Sequenza di eventi principali</b>	<ol style="list-style-type: none"> <li>1. L'utente seleziona il suo ruolo (Paziente / Operatore Sanitario).</li> <li>2. <b>IF</b> L'utente ha selezionato il ruolo di Paziente <ol style="list-style-type: none"> <li>2.1 L'utente inserisce Nome, Cognome, Sesso, Data di Nascita e Luogo di Nascita, Password.</li> </ol> </li> <li><b>ELSE</b> <ol style="list-style-type: none"> <li>2.2 L'utente inserisce Nome, Cognome, Numero di Iscrizione all'Albo professionale, Password.</li> </ol> </li> <li>3. Il sistema valida i dati</li> <li>4. <b>IF</b> I dati sono stati inseriti in maniera scorretta <ol style="list-style-type: none"> <li>4.1 Il sistema procede a richiedere nuovamente i dati all'utente</li> </ol> </li> <li>5. L'utente viene correttamente registrato</li> </ol>
<b>Post-condizioni</b>	L'utente viene correttamente registrato nella piattaforma
<b>Sequenza di eventi alternativi</b>	2.1/2.2/4.1

Caso d'uso:	Autenticazione sicura
Attore primario	Paziente, Operatore sanitario
Attore secondario	-
Descrizione	L'utente accede al sistema tramite credenziali.
Pre-condizioni	L'utente si è precedentemente registrato alla piattaforma.
Sequenza di eventi principali	<ol style="list-style-type: none"> <li>1. L'utente inserisce le credenziali.</li> <li>2. Il sistema valida i dati.</li> <li>3. <b>IF</b> I dati sono stati inseriti in maniera scorretta <ol style="list-style-type: none"> <li>3.1 Il sistema procede a richiedere nuovamente i dati all'utente</li> </ol> </li> <li>4. L'utente ottiene l'accesso in base al profilo.</li> </ol>
Post-condizioni	L'utente accede al sistema con il proprio ruolo.
Sequenza di eventi alternativi	-

Caso d'uso:	Consultazione cronologia personale
Attore primario	Paziente
Attore secondario	-
Descrizione	Il paziente consulta la cronologia dei propri referti.
Pre-condizioni	Il paziente è autenticato.
Sequenza di eventi principali	<ol style="list-style-type: none"> <li>1. Il paziente accede alla sezione "Cronologia".</li> <li>2. Il sistema mostra la lista completa dei report precedenti relativi all'utente.</li> <li>3. Il paziente può selezionare un report per leggerlo.</li> </ol>
Post-condizioni	La cronologia viene mostrata sul dispositivo.
Sequenza di eventi alternativi	-

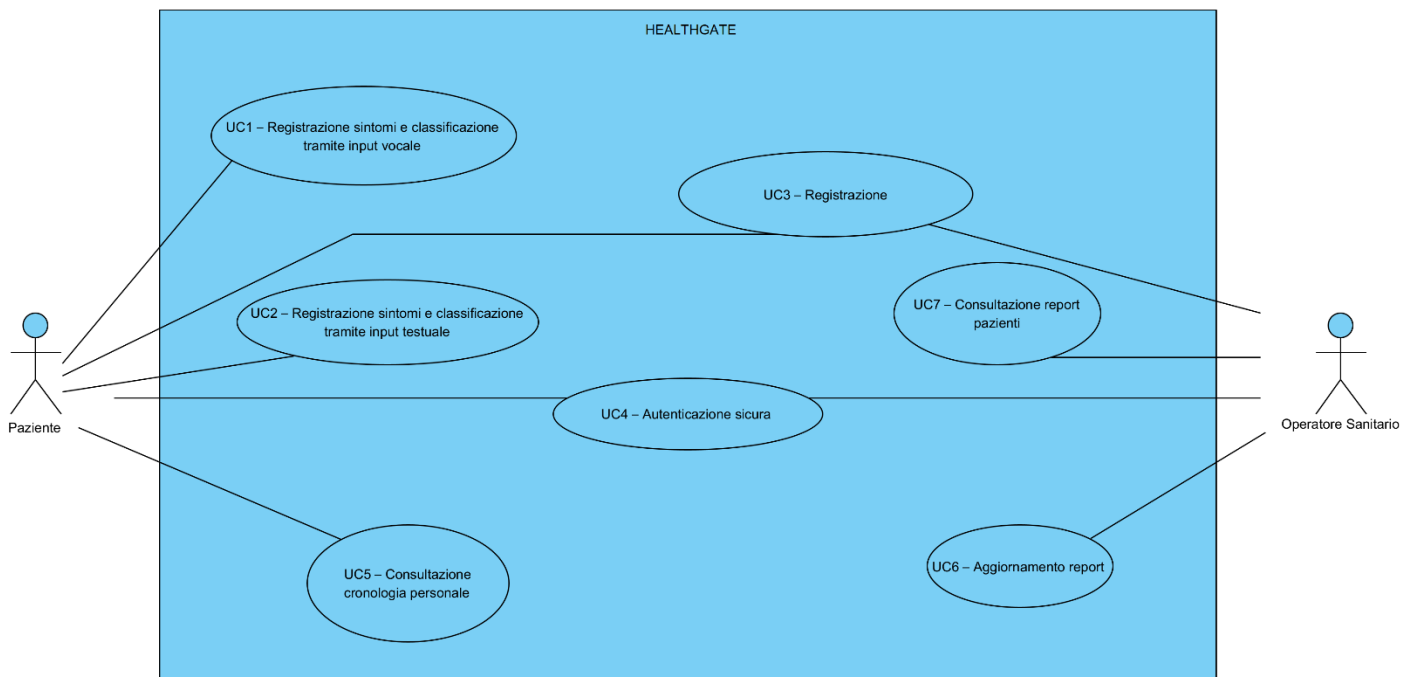
<b>Caso d'uso:</b>	<b>Consultazione report pazienti</b>
<b>Attore primario</b>	Operatore sanitario
<b>Attore secondario</b>	-
<b>Descrizione</b>	L'operatore sanitario consulta un report clinico.
<b>Pre-condizioni</b>	Report clinico già generato. Operatore sanitario autenticato e autorizzato.
<b>Sequenza di eventi principali</b>	<ol style="list-style-type: none"> <li>1. L'operatore sanitario seleziona un paziente dalla lista.</li> <li>2. L'operatore sanitario seleziona un report del paziente.</li> <li>3. Il sistema mostra a video il report selezionato dall'operatore sanitario.</li> </ol>
<b>Post-condizioni</b>	Il report viene mostrato sull'interfaccia
<b>Sequenza di eventi alternativi</b>	-

<b>Caso d'uso:</b>	<b>Aggiornamento report</b>
<b>Attore primario</b>	Operatore sanitario
<b>Attore secondario</b>	-
<b>Descrizione</b>	L'operatore sanitario aggiorna un report clinico aggiungendo informazioni relative a cure e piano terapeutico.
<b>Pre-condizioni</b>	Report clinico già generato. Operatore sanitario autenticato e autorizzato.
<b>Sequenza di eventi principali</b>	<ol style="list-style-type: none"> <li>1. L'operatore sanitario seleziona un report.</li> <li>2. Aggiunge note su cure somministrate e terapie.</li> <li>3. Salva le modifiche.</li> </ol>
<b>Post-condizioni</b>	Il report aggiornato viene salvato nel sistema.
<b>Sequenza di eventi alternativi</b>	-



## Use Case Diagram

Di seguito vi è il diagramma dei casi d'uso:



## System Design

### Architettura a microservizi

Il sistema è stato progettato secondo un'**architettura a microservizi**. Questa architettura consente modularità, scalabilità e una chiara separazione delle responsabilità tra i diversi componenti del sistema. L'utente interagisce tramite un'applicazione **mobile e web** e invia richieste tramite chiamate **API**, gestite da un **API Gateway** che si occupa di indirizzare la richiesta verso il microservizio appropriato.

I principali microservizi che compongono l'architettura sono i seguenti:

#### 1. **API Gateway:**

Funziona come punto di ingresso unico del sistema. Gestisce, orchestra e astrae le chiamate API provenienti dai client, indirizzandole verso i microservizi di competenza. Consente inoltre di applicare politiche di sicurezza, logging e monitoraggio centralizzato.

#### 2. **Authentication Service:**

È responsabile della **gestione delle identità e delle credenziali** degli utenti (pazienti e operatori sanitari). Gestisce la **registrazione**, il **login** e la **protezione delle sessioni**, garantendo autenticazione sicura e controllo degli accessi.

### 3. Symptoms Ingestion Service:

Riceve l'input dell'utente, che può essere fornito **in formato testuale o audio**. In caso di input vocale, il servizio esegue la **trascrizione automatica** (speech-to-text) e normalizza il testo risultante, preparandolo per l'elaborazione successiva.

### 4. Decision Engine Service:

Si occupa di **analizzare e interpretare i sintomi** forniti dall'utente, applicando modelli basati su **Large Language Models (LLM)** integrati con un approccio di tipo **Retrieval-Augmented Generation (RAG)**. Il servizio determina quindi se sia necessario recarsi al **Pronto Soccorso**, fornendo al tempo stesso una **spiegazione chiara e motivata** della decisione, così da assicurare **trasparenza e tracciabilità** del processo decisionale.

### 5. Aggregator Service:

Funziona come livello intermedio di integrazione tra i diversi microservizi. Il suo compito principale è **aggregare e uniformare i dati provenienti da più sorgenti** (come i report clinici, i dati anagrafici dei pazienti e le decisioni diagnostiche) per fornire una **vista unificata e coerente** al Decision Engine. In questo modo, riduce la complessità delle chiamate multiple ai microservizi, migliora le prestazioni complessive e semplifica la comunicazione tra i componenti del sistema.

### 6. Report Service:

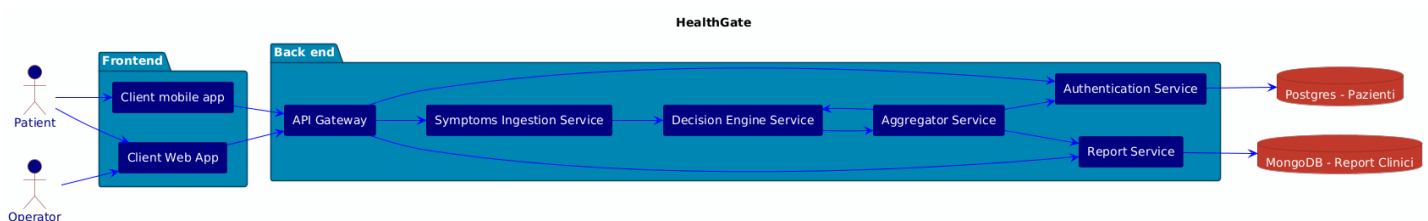
Si occupa della **generazione, archiviazione e consultazione dei referti clinici**. I report vengono creati automaticamente a partire dalle decisioni del motore diagnostico e possono essere visualizzati o aggiornati dagli operatori sanitari autorizzati.

## Modellazione complessiva del sistema

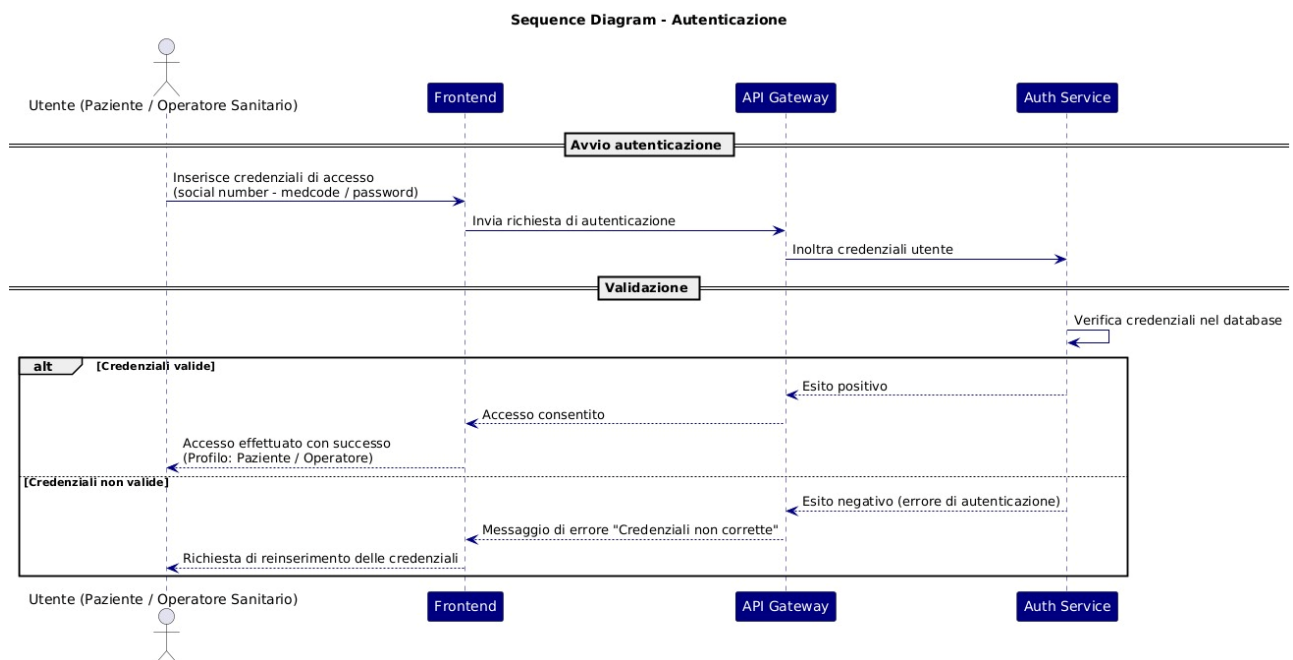
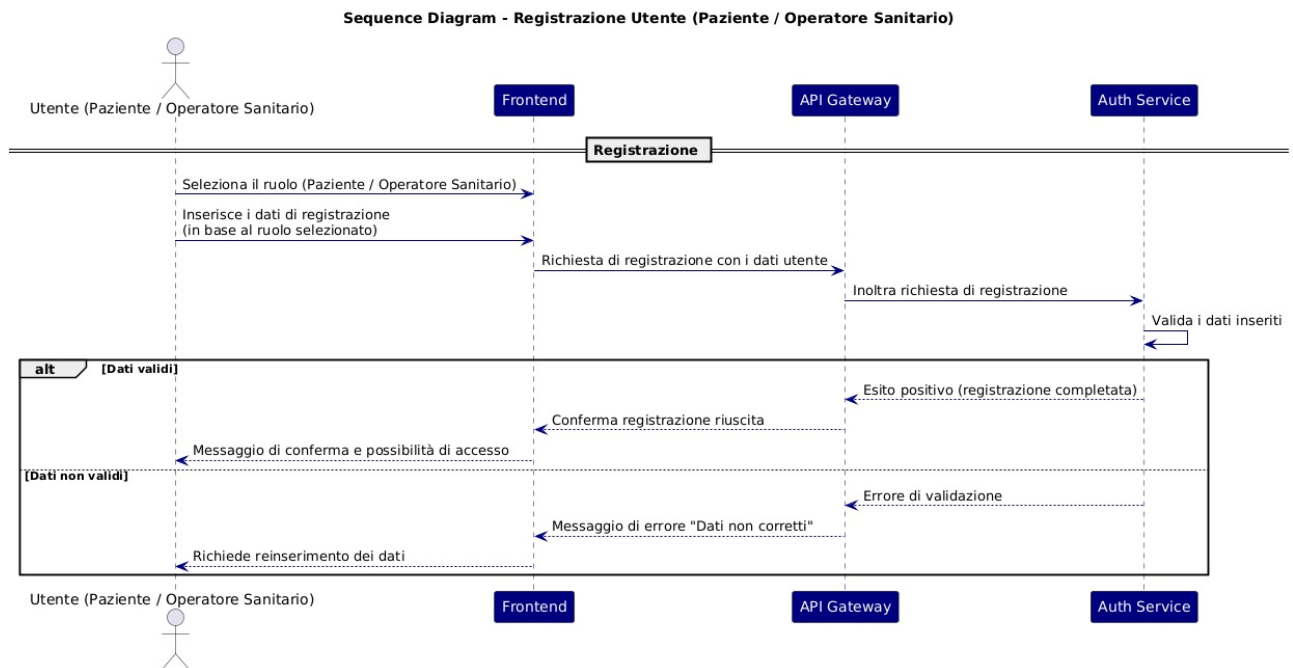
Per rappresentare in modo formale la struttura generale del sistema **HealthGate**, abbiamo realizzato un **diagramma di alto livello**. Questo diagramma mostra la **composizione logica dell'architettura a microservizi**, evidenziando i principali blocchi funzionali e le loro relazioni.

Il diagramma presenta:

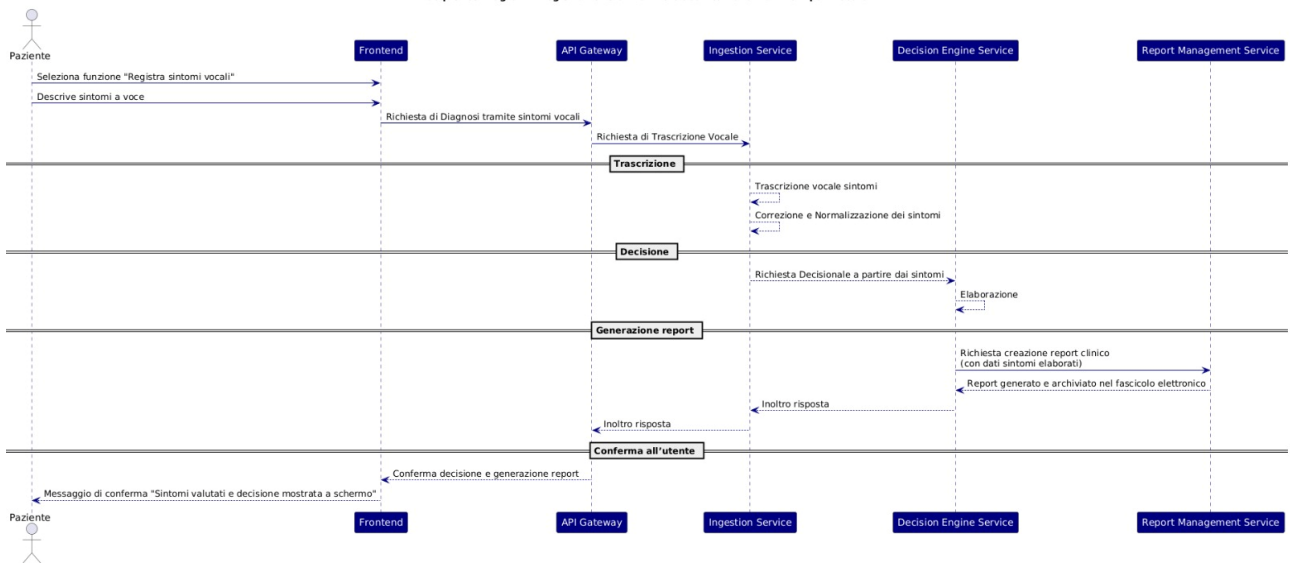
- Gli utenti (paziente, operatore) che attraverso il **frontend** interagiscono con l'unico punto di accesso del sistema, l'**API Gateway**;
- Il Gateway coordina i microservizi **Auth**, **Aggregator**, **Ingestion**, **Decision Engine** e **Report Management**;
- Auth e Report si interfacciano con i rispettivi database **PostgreSQL** e **MongoDB**.



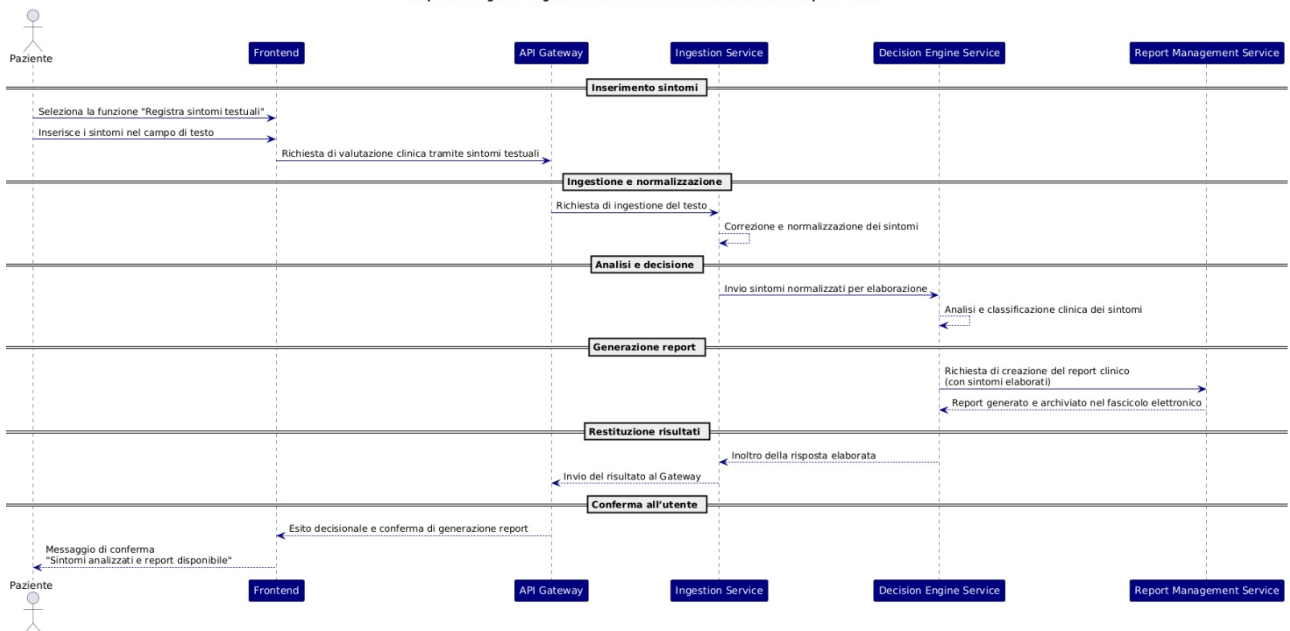
# Sequence Diagram



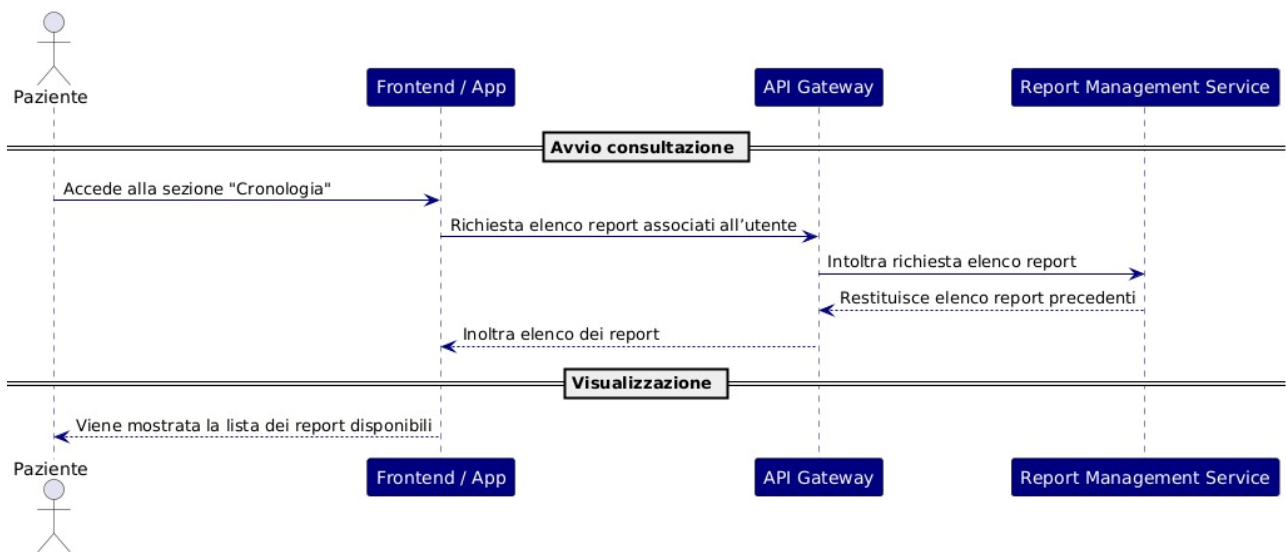
Sequence Diagram - Registrazione Sintomi e Classificazione tramite Input Vocale



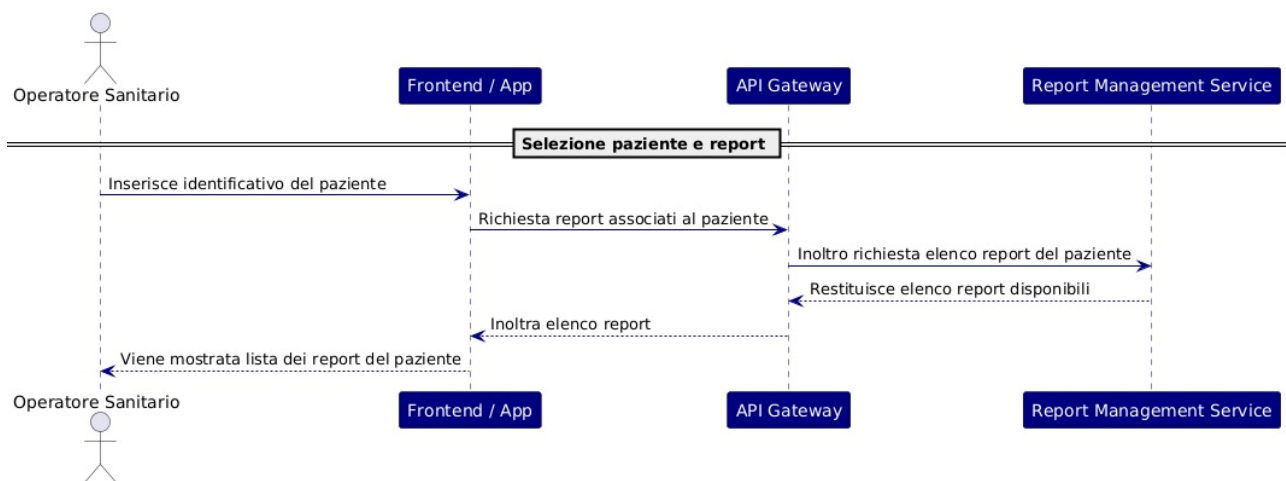
Sequence Diagram - Registrazione Sintomi e Classificazione tramite Input Testuale

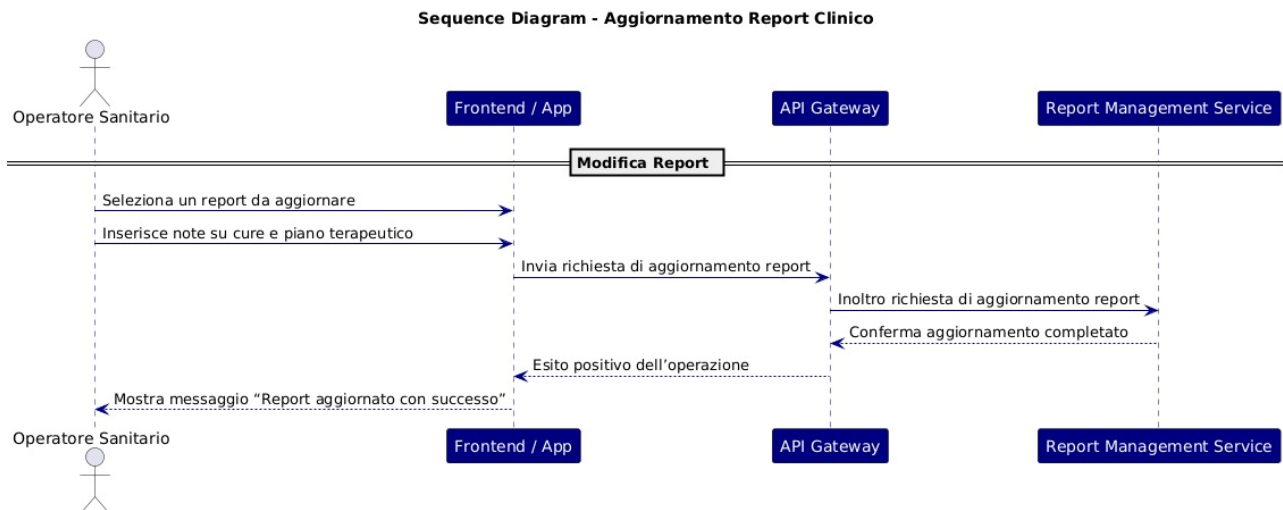


**Sequence Diagram - Consultazione Cronologia Personale**



**Sequence Diagram - Consultazione Report Pazienti**





## Architettura logica e organizzazione dei componenti

L'organizzazione del progetto *HealthGate* riflette la struttura logica dell'architettura a microservizi descritta in precedenza. Il sistema è suddiviso in due macro-componenti principali:

1. Il **Frontend**, che gestisce l'interazione con l'utente.
2. Il **Backend**, implementato attraverso una serie di **microservizi indipendenti**, ognuno con responsabilità e database propri.

Questa separazione consente di mantenere una chiara distinzione tra la presentazione e la logica applicativa, favorendo modularità, scalabilità e facilità nella manutenzione. Ogni microservizio può essere sviluppato, testato e distribuito in modo autonomo, senza interferire con gli altri componenti del sistema.

## Implementazione

Dal punto di vista implementativo, l'intero sistema è organizzato in due macrodirectory principali: *frontend/* e *microservices/*, ciascuna con un ruolo ben definito nell'architettura complessiva.

### Frontend

La componente **frontend** dell'applicazione *HealthGate* è sviluppata interamente in **Python** utilizzando **Streamlit**, una libreria pensata per la creazione rapida di interfacce web interattive. Questa scelta permette di realizzare un'interfaccia **moderna, reattiva e facilmente estendibile**, mantenendo al contempo un'integrazione diretta con i microservizi backend senza la necessità di framework complessi. Il frontend comunica **esclusivamente con l'API Gateway**, che si occupa di inoltrare le richieste ai microservizi appropriati (Autenticazione, Ingestion, Decision Engine, Report). Le risposte sono gestite in formato **JSON**, con messaggi di

stato e notifiche visualizzate direttamente all'utente tramite componenti interattive di Streamlit.

L'interfaccia è composta da più moduli indipendenti, ciascuno responsabile di una funzionalità specifica:

- *HealthGate.py*

È il modulo principale che gestisce il flusso logico dell'applicazione. Controlla la navigazione tra le schermate (home, login, registrazione, area paziente, area operatore) utilizzando lo stato della sessione Streamlit (`st.session_state.view`). In base alla vista corrente richiama dinamicamente i moduli corrispondenti (*main.py*, *login.py*, *signup.py*, *patient\_ui.py*, *operator\_ui.py*).

- *main.py*

Gestisce la **home page** dell'applicazione e l'inizializzazione dello stato globale tramite la funzione *initialize\_session\_state()* definita in *config\_css.py*. Mostra le tre principali opzioni di accesso: autenticazione come **paziente**, autenticazione come **operatore sanitario**, oppure **registrazione** di un nuovo utente. Include il caricamento dei **fogli di stile CSS personalizzati**, definiti in *config\_css.py*, che curano l'aspetto grafico (bottoni, header, card, colori, hover).

- *login.py*

Gestisce il **processo di autenticazione** di pazienti e operatori. Attraverso chiamate HTTP POST verso l'**API Gateway**, invia le credenziali e, in caso di successo, memorizza nella session state i dati dell'utente e il **token JWT**. Include una gestione dettagliata degli **errori backend**, traducendo i messaggi tecnici (Pydantic, FastAPI) in notifiche leggibili per l'utente.

- *signup.py*

Implementa il **processo di registrazione** per nuovi utenti, con due modalità distinte:

- **Paziente:** inserimento di dati anagrafici (nome, cognome, data e luogo di nascita, sesso, password);
- **Operatore sanitario:** registrazione tramite codice d'iscrizione all'albo, e-mail, numero di telefono e password.

Il modulo effettua **validazione locale dei campi**, controllando la coerenza delle password e la completezza dei dati prima di inviare la richiesta al backend tramite l'API Gateway.

- *patient\_ui.py*

Definisce la **schermata riservata al paziente**, articolata in due sezioni principali:

- **Registrazione dei sintomi:** L'utente può fornire i propri sintomi scegliendo una delle tre modalità: **Registrazione vocale** (tramite microfono del dispositivo), **Caricamento di un file audio** già esistente, **Trascrizione manuale** del testo. I file vengono salvati temporaneamente in locale e poi inviati al backend per l'elaborazione da parte dei microservizi *Symptoms Ingestion* e *Decision Engine*.

- **Visualizzazione dei report clinici:** Il paziente può consultare i referti generati, scaricarli in formato **PDF**. I dati sono recuperati tramite richieste GET verso l'API Gateway, che inoltra la chiamata al *Report Service*.

Entrambe le sezioni condividono un menu laterale per la navigazione interna e la possibilità di effettuare **logout sicuro** tramite dialog modale.

- *operator\_ui.py*

È il modulo dedicato agli operatori sanitari, che consente la ricerca e consultazione dei report clinici dei pazienti. Comprende funzionalità di:

- Ricerca per codice fiscale,
- filtraggio per data,
- download dei report in formato PDF.

In futuro, il modulo potrà essere esteso per permettere l'aggiornamento o validazione dei referti direttamente dal frontend.

## Microservices

### API Gateway

In un'architettura a microservizi, le singole componenti applicative sono progettate per essere **piccole, autonome e specializzate**, ciascuna con un proprio ciclo di vita, linguaggio di sviluppo e database. Questa modularità, se da un lato migliora la scalabilità e la manutenibilità del sistema, dall'altro introduce una complessità significativa nella gestione delle comunicazioni tra client e servizi. Dal punto di vista del client (applicazione web o mobile) il sistema appare come un insieme eterogeneo di servizi, ognuno con **endpoint, protocolli e politiche di autenticazione differenti**. Per evitare che il client debba conoscere i dettagli interni di ogni servizio, viene introdotto un **API Gateway**, che agisce come **strato di astrazione e coordinamento centrale** tra il mondo esterno e la rete di microservizi. L'**API Gateway** funge quindi da **punto di accesso unificato** per tutte le richieste. Si occupa di ricevere, validare, trasformare e indirizzare ogni richiesta verso il microservizio appropriato, gestendo la logica di comunicazione e di sicurezza in un unico livello. In questo modo, il client comunica sempre con un solo endpoint (il Gateway) mentre la complessità interna rimane completamente nascosta.

### Ruolo dell'API Gateway nelle architetture a microservizi

Il Gateway non è solo un router, ma un **livello di orchestrazione intelligente** che integra funzionalità di sicurezza, gestione del traffico e ottimizzazione delle prestazioni.

Le sue funzioni principali comprendono:

- 1. Instradamento e bilanciamento del traffico (Routing & Load Balancing):**



Il Gateway analizza ogni richiesta e la inoltra al microservizio corretto in base all'endpoint richiesto, al ruolo dell'utente o alle regole di routing definite. Può anche distribuire le richieste tra più istanze dello stesso servizio, migliorando la **scalabilità orizzontale**.

## 2. Autenticazione, autorizzazione e sicurezza centralizzate:

Tutte le richieste, ad eccezione di login e registrazione, passano attraverso la funzione `verify_jwt_with_role()`, che decodifica e valida il **token JWT** inviato dal client. La verifica include la scadenza del token, la firma crittografica e il controllo del **ruolo utente** (patient, operator), applicando politiche di accesso granulari a seconda del servizio richiesto.

## 3. Trasformazione dei messaggi e sicurezza dei microservizi:

L'API Gateway può adattare i formati dei messaggi (es. da JSON a XML o viceversa) o convertire protocolli, permettendo l'interoperabilità tra sistemi eterogenei. Dopo la validazione, il Gateway rimuove l'header Authorization e lo sostituisce con header interni (X-User-Id, X-User-Role, X-User-Expiry), così i microservizi ricevono solo informazioni contestuali, senza dover gestire direttamente i token JWT. Questo approccio riduce la superficie d'attacco e semplifica la logica di sicurezza dei servizi interni.

## 4. Fault Tolerance e resilienza:

Può implementare meccanismi di **circuit breaker**, **timeout management** e **retry policy** per evitare che un errore in un microservizio si propaghi all'intero sistema. In questo modo aumenta l'affidabilità complessiva dell'applicazione.

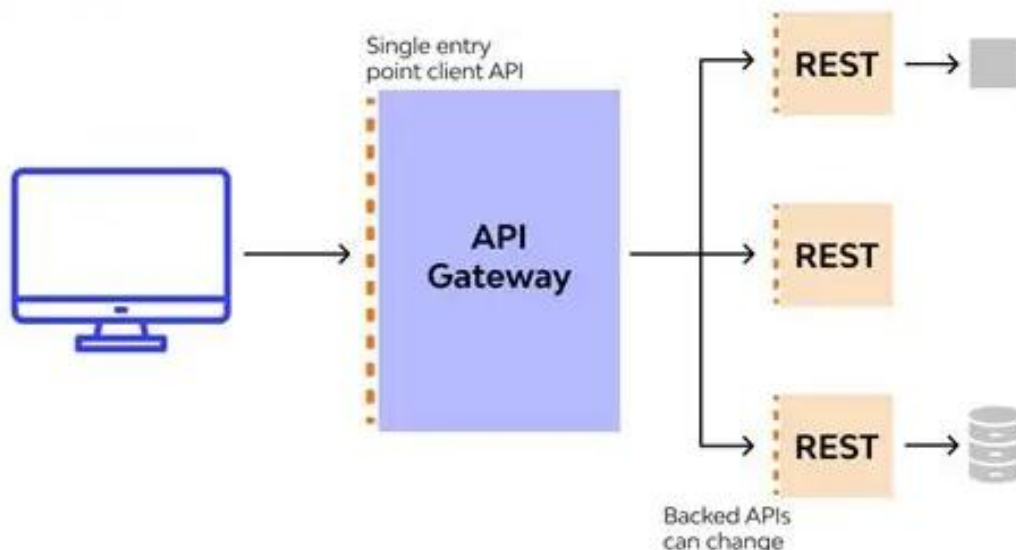
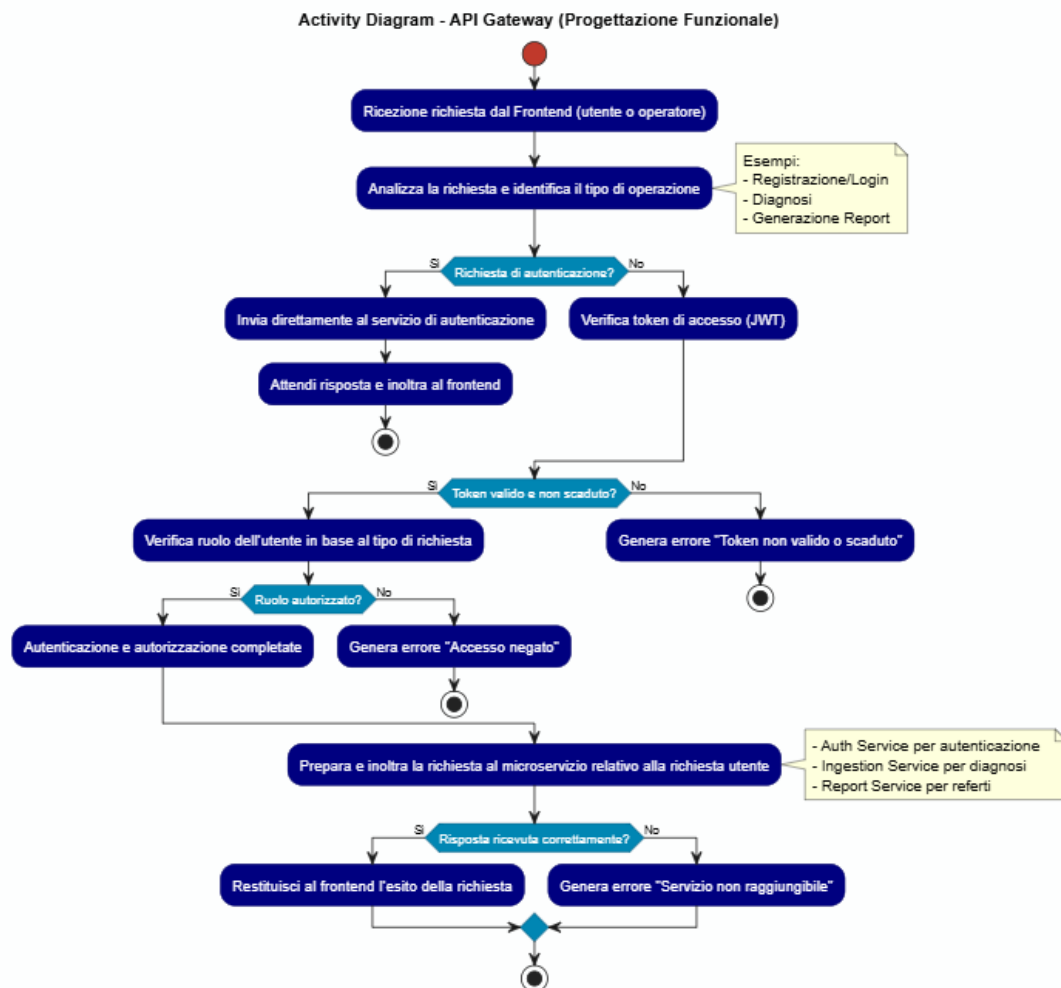


Figura: Principio di funzionamento di un API Gateway in un'architettura a microservizi.

## Activity Diagram dell'API Gateway

Per rappresentare in modo formale e sistematico il comportamento e la struttura dell'API Gateway all'interno dell'architettura HealthGate, è stato adottato il linguaggio **SysML (Systems Modeling Language)**. La modellazione SysML dell'API Gateway è utile per descrivere, attraverso l'Activity Diagram, il flusso operativo del router durante la gestione delle richieste e delle risposte.



## Descrizione implementativa dell'API Gateway e scelte progettuali:

L' **API Gateway** di *HealthGate* è stato realizzato in **Python** utilizzando il framework **FastAPI**, una scelta dettata da esigenze di **prestazioni**, **scalabilità** e **semplicità di integrazione asincrona** con gli altri servizi del sistema. Di seguito sono illustrate nel dettaglio le scelte progettuali più rilevanti.

### - Scelta del framework: FastAPI

L'utilizzo di **FastAPI** è motivato dalla necessità di un **gateway leggero e altamente performante**, in grado di gestire simultaneamente numerose richieste provenienti dal

frontend Streamlit. FastAPI supporta nativamente l'elaborazione **asincrona** tramite *async/await*, permettendo di gestire un elevato throughput con un ridotto consumo di risorse.

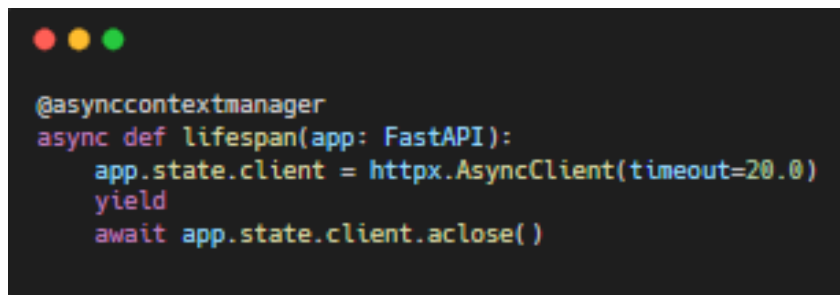
I principali vantaggi che ne hanno giustificato l'adozione sono:

1. **Esecuzione asincrona nativa:** riduce la latenza nelle chiamate multiple verso i microservizi, migliorando la responsività del sistema.
2. **Definizione chiara delle API:** tramite annotazioni tipizzate (*pydantic*, *typing*), semplifica la documentazione e la validazione automatica delle richieste.
3. **Semplice estensibilità:** la logica di sicurezza, routing e proxy può essere personalizzata con poche righe di codice, senza dipendenze complesse.
4. **Perfetta integrazione con uvicorn e httpx,** garantendo velocità e stabilità nella comunicazione tra i componenti.

In un sistema distribuito come *HealthGate*, dove i microservizi di autenticazione, ingestion e decision possono risiedere su host diversi, l'approccio asincrono di FastAPI rappresenta la soluzione ideale per evitare colli di bottiglia lato gateway.

#### - **Comunicazione asincrona e gestione del ciclo di vita**

Nel file `main.py`, la creazione del client HTTP asincrono avviene all'interno del contesto di vita dell'applicazione:



```
@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.client = httpx.AsyncClient(timeout=20.0)
    yield
    await app.state.client.aclose()
```

Questa scelta è dovuta a precise motivazioni architetturali:

- **Persistenza delle connessioni:** l'oggetto `AsyncClient` mantiene connessioni aperte verso i microservizi, evitando overhead di creazione socket per ogni richiesta.
- **Ottimizzazione delle prestazioni:** un singolo client riutilizzabile è più efficiente rispetto alla creazione di client ad ogni chiamata, garantendo tempi di risposta più bassi.

## - Sicurezza e gestione dei token JWT

Il Gateway implementa la verifica dei token JWT tramite la funzione:

```
# --- verifica token e controllo ruolo generico ---
async def verify_jwt_with_role(request: Request, required_role: str):
    """
    Verifica firma e scadenza del token e controlla che l'utente
    abbia il ruolo richiesto.
    """
    print(request.headers)

    auth_header = request.headers.get("Authorization")
    if not auth_header or not auth_header.startswith("Bearer "):
        raise HTTPException(status_code=401, detail="Token mancante")

    token = auth_header.split(" ")[1]

    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[JWT_ALGORITHM])
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail="Token scaduto")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail="Token non valido")

    # Controllo ruolo
    role = payload.get("scope")

    if role != required_role:
        raise HTTPException(status_code=403, detail="Permesso negato")

    request.state.user = {"user_id": payload.get("sub"), "role": role, "expiry": payload.get("exp")}
    return
```

Le motivazioni dietro questa scelta sono fortemente legate ai principi di **Single Responsibility** e **Zero Trust Security**:

- **Delegated Authentication:** l'autenticazione avviene esclusivamente nel Gateway, che funge da “guardiano” dell'intera architettura. I microservizi non si occupano della validazione dei token, riducendo complessità e duplicazione del codice.
- **Ruolo contestuale:** il Gateway non solo valida il token, ma controlla che l'utente possieda il ruolo corretto (es. patient, operator). Ciò consente un accesso differenziato alle rotte, applicando policy di sicurezza granulari.
- **Riduzione della superficie d'attacco:** centralizzando la verifica del token, si evita che i microservizi esponano direttamente endpoint vulnerabili.

In caso di token non valido, scaduto o assente, il Gateway restituisce risposte standardizzate (401 Unauthorized, 403 Forbidden), garantendo coerenza semantica nelle risposte HTTP.

## - Configurazione con il file. env

La configurazione del sistema è completamente externalizzata tramite il file. env, da cui config.py importa i parametri critici. Le motivazioni principali sono:

- **Separazione tra codice e configurazione:** nessun dato sensibile o percorso fisso è hardcoded, facilitando il deployment su ambienti diversi (sviluppo, test, produzione).

- **Sicurezza:** la chiave segreta JWT e gli endpoint dei servizi restano esterni al codice sorgente, riducendo il rischio di esposizione.
- **Scalabilità:** modificare un endpoint (es. spostare il Decision Engine su un nuovo nodo) richiede solo l'aggiornamento del file. env, senza toccare il codice.

#### - Routing dinamico e funzione di proxy

La funzione `proxy_request()` rappresenta il cuore operativo del Gateway. Essa costruisce dinamicamente l'URL di destinazione combinando la rotta richiesta con il microservizio corrispondente.

```
# Funzione di proxy verso i microservizi
async def proxy_request(request: Request, method: str, service_url: str, role: str = None):

    target_path = request.url.path

    if not target_path.startswith(("login", "signup")):
        # await verify_jwt(request)
        # await verify_jwt_with_role(request, role)

    if target_path.startswith("/diagnose"): # da vedere se ci piace
        target_path = "/ingestion" # localhost:8000/diagnose -> localhost:8002/ingestion

    # localhost:8000/ingestion -> localhost:8002/ingestion

    url = f"{service_url}{target_path}" # concateniamo con l'url del microservizio es. /localhost:8001

    """
    Esempio:
    # Richiesta originale al gateway:
    GET http://localhost:8000/users/42/profile

    # service_url (microservizio utenti):
    service_url = "http://localhost:8001"

    # Risultato finale:
    url = "http://localhost:8001/users/42/profile"
    """

    # è inutile ricavare il body se ho una get o head
    if method in ["post", "put", "patch", "delete"]:
        body = await request.body()
    else:
        body = None

    headers = dict(request.headers)

    # Se l'utente è autenticato, aggiungo info interne - Strategia microservizi si fidano del controllo gateway sul token
    if hasattr(request.state, "user"):
        headers["X-User-Id"] = str(request.state.user.get("user_id"))
        headers["X-User-Role"] = str(request.state.user.get("role", "user"))
        headers["X-User-Expiry"] = str(request.state.user.get("expiry", ""))
        headers.pop("Authorization", None) # il microservizio non ha bisogno del token

    # Richiesta al microservizio
    try:
        response = await app.state.client.request(method, url, content=body, headers=headers)
    except httpx.ReadTimeout:
        raise HTTPException(status_code=504, detail="Timeout durante la comunicazione")

    return Response(content=response.content, status_code=response.status_code, headers=response.headers)
```

Le scelte qui riflettono due principi cardine:

1. **Astrazione dei microservizi:** il client (frontend) non conosce mai gli indirizzi o le porte interne dei microservizi, interagendo solo con il Gateway.
2. **Flessibilità di estensione:** nuovi microservizi possono essere aggiunti semplicemente aggiornando la mappa `MICROSERVICES` in `config.py`, senza modificare il codice di routing.

## Authentication Service

In un'architettura a microservizi, la gestione dell'identità e dell'accesso rappresenta uno degli aspetti più delicati e critici del sistema, poiché ogni servizio opera in modo indipendente e deve garantire la sicurezza dei propri endpoint senza compromettere la coerenza. Per evitare ridondanze e vulnerabilità, viene introdotto un **Authentication Service** centralizzato, responsabile della **registrazione (signup)**, **autenticazione (login)** e **gestione dei token di accesso (JWT)** di tutti gli utenti del sistema. Il principio architetturale alla base è quello della **centralizzazione della sicurezza**, secondo cui le funzionalità di autenticazione vengono estratte dai singoli microservizi e delegate a un unico componente, che diventa la **fonte di verità** per tutto ciò che riguarda gli utenti, le credenziali e i ruoli.

### Ruolo e funzioni principali

L'Authentication Service ha tre responsabilità fondamentali:

1. **Identificazione e registrazione utenti (Signup):** gestisce la creazione di nuovi account per pazienti e operatori sanitari, assicurando che i dati inseriti siano validi e che le credenziali (password) siano trattate in modo sicuro tramite tecniche di hashing.
2. **Autenticazione e generazione token (Login):** verifica le credenziali fornite, e in caso di successo, genera un **JWT (JSON Web Token)** firmato digitalmente, contenente l'identificativo dell'utente, il suo ruolo (es. *patient* o *operator*), il timestamp di emissione e di scadenza. Ciò che viene utilizzato è l'**autenticazione stateless**, ovvero invece di mantenere sessioni utente sul server, ogni richiesta successiva al login trasporta autonomamente il token JWT che contiene tutte le informazioni necessarie per identificare l'utente. Questo approccio elimina la necessità di sessioni persistenti, consente di scalare facilmente il servizio, è compatibile con architetture containerizzate e distribuite come Docker.
3. **Autorizzazione e controllo ruoli:** il token JWT rilasciato dal servizio include informazioni sul ruolo, permettendo all'**API Gateway** e agli altri microservizi di verificare rapidamente i privilegi dell'utente senza dover interrogare nuovamente il database di autenticazione.

### Separazione dei ruoli e sicurezza dei dati

Nel sistema *HealthGate*, l'Authentication Service distingue due tipologie di utenti:

- **Pazienti:** utenti finali che descrivono i propri sintomi e ricevono una valutazione;
- **Operatori sanitari:** medici o personale autorizzato a consultare e aggiornare i report clinici.

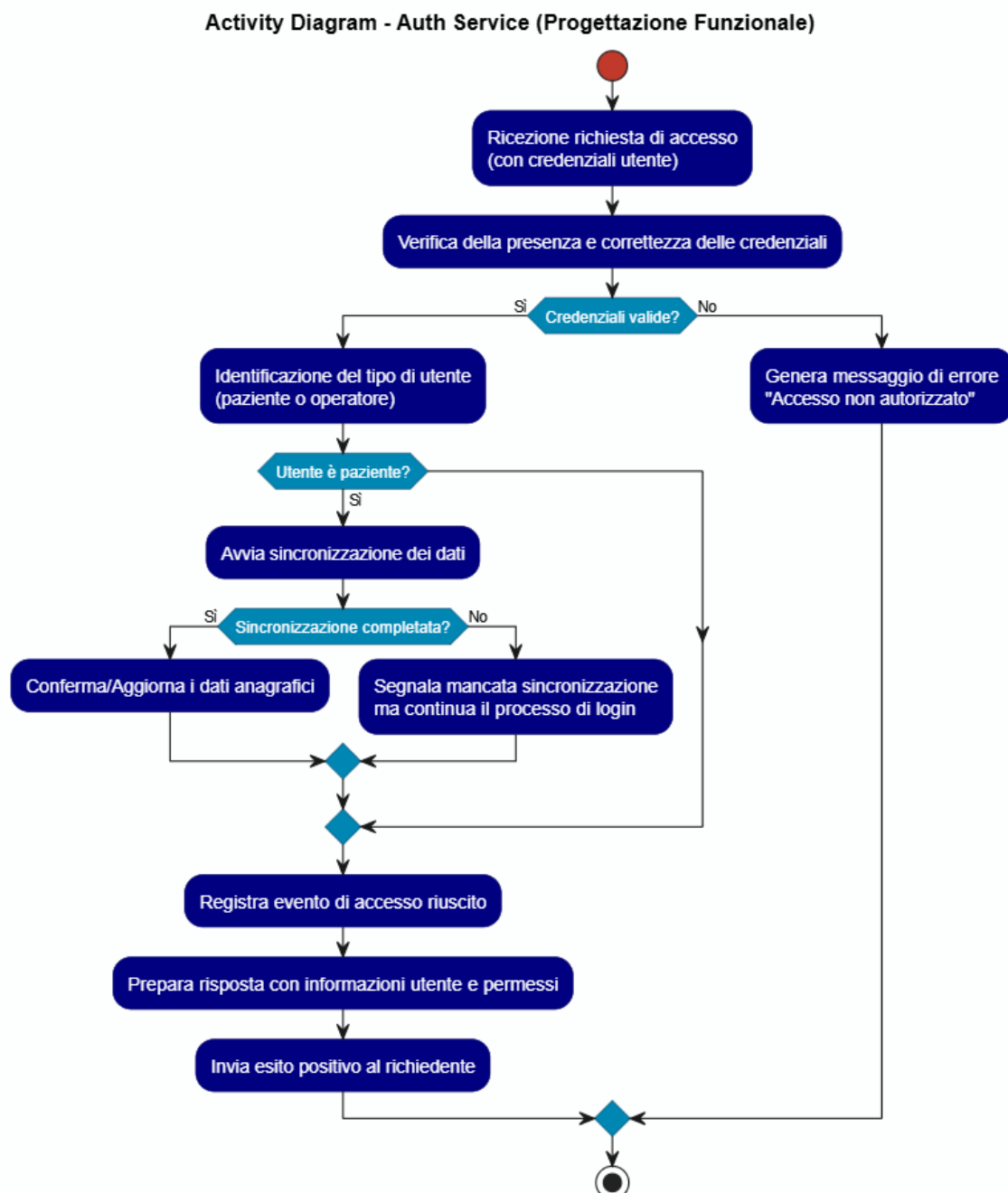
La distinzione è implementata a livello di *scope* nel payload JWT, che consente di applicare controlli di accesso granulari a valle (nell'API Gateway o nei microservizi).

Tutti i dati sensibili (in particolare le password) vengono gestiti secondo i principi della **sicurezza by design**:

- le password non vengono mai salvate in chiaro ma **hashate con l'algoritmo bcrypt**;
- il token JWT è firmato con una **chiave segreta privata** (SECRET\_KEY) e Il campo **exp** (expiration) definisce la durata del token in secondi a partire dal momento di creazione, secondo il valore impostato in JWT\_EXPIRE\_MINUTES nel file env. Le operazioni di login e registrazione sono soggette a validazioni rigorose dei campi.

### Activity Diagram dell'Authentication

Nel contesto di HealthGate, la modellazione SysML dell'**Authentication Service** è utile per descrivere, attraverso l'**Activity Diagram**, il **flusso operativo di autenticazione**, che comprende la validazione del token JWT e la gestione delle credenziali utente;



## Descrizione implementativa dell'authentication service e scelte progettuali

L'implementazione del microservizio è stata realizzata in **Python** con **FastAPI**, **SQLAlchemy** (**async**) e **PostgreSQL**.

**FastAPI:** è un framework web per Python, cioè uno strumento che semplifica la creazione di API RESTful, ovvero le interfacce con cui client e altri microservizi possono comunicare.

- È veloce (usa `async/await` per eseguire più operazioni contemporaneamente);
- Valida automaticamente i dati in ingresso grazie a `Pydantic`;
- Genera documentazione automatica delle API.

Nell'Authentication Service, FastAPI si occupa di:

- Gestire le richieste HTTP (POST /signup, POST /login),
- Passare i dati ricevuti ai moduli interni (DB, sicurezza, validazione),
- Restituire al client una risposta in formato JSON.

**SQLAlchemy:** È una libreria Python per interagire con i database in modo semplice e strutturato. Invece di scrivere query SQL a mano, possiamo lavorare con oggetti Python che rappresentano le tabelle del database. Questa tecnica si chiama ORM (Object Relational Mapping). La versione "async" che usiamo è progettata per funzionare bene con FastAPI: permette di comunicare con il database senza bloccare le altre richieste. In pratica:

- `create_patient ()` crea un nuovo record nella tabella patients;
- `find_operator_by_med_code ()` esegue una query per cercare un operatore.

Tutto questo avviene in parallelo con altre richieste, grazie al supporto asincrono (`async/await`).

**PostgreSQL:** È il database relazionale (cioè con tabelle, colonne e chiavi) che usiamo per memorizzare i dati degli utenti.

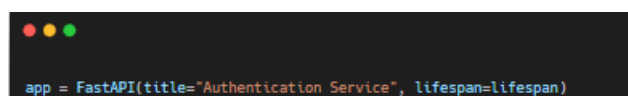
Nel nostro caso contiene due tabelle principali:

- patients → informazioni anagrafiche e password hashate dei pazienti;
- operators → dati identificativi degli operatori sanitari.

Ogni volta che un utente si registra o effettua il login, l'Authentication Service interagisce con PostgreSQL tramite SQLAlchemy.

## Struttura e inizializzazione del servizio

Nel file `main.py`, il servizio viene istanziato tramite:



```
app = FastAPI(title="Authentication Service", lifespan=lifespan)
```



Il ciclo di vita dell'applicazione (**lifespan**) inizializza automaticamente il database PostgreSQL all'avvio del servizio, sincronizzando le tabelle definite nei modelli SQLAlchemy. Questo assicura il corretto bootstrap in ambienti containerizzati Docker senza richiedere script manuali di migrazione e senza creare errori di configurazione.

```
# Definisce una funzione asincrona per inizializzare il database
async def init_db():
    # Apre una connessione asincrona al database in modalità transazionale
    async with engine.begin() as conn:
        # Esegue la funzione sincrona 'create_all' di SQLAlchemy
        # che crea tutte le tabelle definite nei modelli ereditati da Base
        # 'run_sync' permette di eseguire funzioni sincrone in contesto asincrono
        await conn.run_sync(Base.metadata.create_all)
```

## Gestione del database e modelli

Nel file *db\_ops.py*:

- viene creato un **motore asincrono SQLAlchemy** (*create\_async\_engine*) collegato al database PostgreSQL (*DATABASE\_URL*);
- vengono definiti i metodi CRUD asincroni per pazienti e operatori (*create\_patient*, *find\_operator\_by\_med\_code*, ecc.);
- viene gestita l'integrità referenziale con gestione delle eccezioni *IntegrityError* per evitare duplicati. Infatti, in caso di duplicati (es. un paziente già registrato), la funzione *create\_patient()* cattura l'eccezione e restituisce un errore http 400 con messaggio descrittivo. Questo permette di evitare crash del servizio e fornire un feedback chiaro al frontend.

```
try:
    await db.commit()
    await db.refresh(patient)
    return patient
except IntegrityError:
    await db.rollback()
    raise HTTPException(
        status_code=400,
        detail="Un paziente con gli stessi dati anagrafici esiste già."
    )
```

Nel file schemas.py sono definiti i modelli:

```
class Patient(Base):
    __tablename__ = "patients"
    id = Column(Integer, primary_key=True, autoincrement=True)
    social_sec_number = Column(String(16), nullable=False, unique=True, index=True)
    firstname = Column(String(30), nullable=False)
    lastname = Column(String(30), nullable=False)
    birth_date = Column(Date, nullable=False)
    sex = Column(String(1), nullable=False)
    birth_place = Column(String(100), nullable=False)
    hashed_password = Column(String(255), nullable=False)

class Operator(Base):
    __tablename__ = "operators"
    id = Column(Integer, primary_key=True, autoincrement=True)
    med_register_code = Column(String(20), unique=True)
    firstname = Column(String(30), nullable=False)
    lastname = Column(String(30), nullable=False)
    email = Column(String(255), nullable=False, unique=True)
    phone_number = Column(String(20), nullable=False, unique=True)
    hashed_password = Column(String(255), nullable=False)
```

La separazione tra **model (schema)** e **validazione (Pydantic)** migliora la manutenibilità e la robustezza dei dati in input. Le classi Pydantic definite in *validation.py* usano i **validatori asincroni** di Pydantic v2 (@field\_validator) per controllare formati e complessità dei dati in ingresso.

### Sicurezza e generazione token

Il file security.py contiene le funzioni di hashing e verifica delle password (bcrypt) e la generazione del token JWT:

```
def create_access_token(user_id: str, role: str):
    # Otteniamo il timestamp attuale
    now = int(time.time())

    # Definiamo il contenuto (payload) del JWT:
    payload = {
        "sub": str(user_id),          # "subject": identifica l'utente (es. user id)
        "iat": now,                  # "issued at": quando è stato generato il token
        "exp": now + JWT_EXPIRE_MINUTES, # "expiration": quando il token scadrà (ora + durata in sec.)
        "jti": str(uuid.uuid4()),      # "JWT ID": identificativo unico del token (utile per blacklist)
        "scope": role                 # "scope": ruolo dell'utente (Paziente o Operatore Sanitario)
    }

    # Firmiamo il token con la SECRET_KEY e algoritmo scelto (HS256 in questo caso)
    token = jwt.encode(payload, SECRET_KEY, algorithm=JWT_ALGORITHM)

    # Ritorniamo il token JWT pronto da mandare al client
    return token
```

Questa implementazione garantisce:

- **sicurezza crittografica** grazie alla firma HMAC-SHA256;
- **limitazione temporale del token (exp)**;
- **identificatore univoco (jti)** per prevenire riutilizzi;
- **ruolo integrato nel token**, utile per il controllo accessi a livello di API Gateway.

### Validazione degli input

Il file validation.py definisce i modelli Pydantic per la verifica dei dati di input nelle richieste REST:

- PatientSignupRequest, OperatorSignupRequest per la registrazione;
- PatientLoginRequest, OperatorLoginRequest per l'autenticazione.

Le validazioni includono:

- controllo di complessità della password (almeno una maiuscola e un numero);
- verifica della data di nascita (non futura);
- formattazione e-mail e numero di telefono per gli operatori.

Questa logica di validazione **previene attacchi di injection** e garantisce **pulizia dei dati** prima dell'inserimento nel database.

### Rotte REST e logica applicativa

Nel file `main.py`:

- le rotte `POST /signup/patient` e `POST /signup/operator` gestiscono la creazione dei nuovi utenti;
- le rotte `POST /login/patient` e `POST /login/operator` eseguono l'autenticazione e rilasciano un token JWT valido.

Durante il login:

1. Il sistema verifica le credenziali tramite `verify_password()`.
2. Se l'autenticazione ha esito positivo, viene generato un JWT con `create_access_token()`.
3. I dati dell'utente vengono restituiti in JSON, includendo il token e i metadati anagrafici.

Tutte le rotte FastAPI restituiscono oggetti Python convertiti automaticamente in JSON grazie al modello `response_model=dict`, evitando la necessità di serializzare manualmente i dati.

Questa progettazione separa chiaramente la **logica di business (db\_ops)** dalla **logica di presentazione (FastAPI routes)**, seguendo il principio **Separation of Concerns**.

### Ingestion Service

Il microservizio **Ingestion Service** rappresenta il punto di ingresso del sistema *HealthGate* per la **raccolta, elaborazione e normalizzazione dei sintomi** forniti dal paziente. Il suo compito è quello di **acquisire input multimodali** (audio o testo), **trascriverli, correggerli linguisticamente e prepararli per il modulo Decision Engine**, che effettuerà poi la valutazione clinica tramite modelli LLM. L'obiettivo architetturale principale del servizio è **ridurre la distanza semantica tra linguaggio naturale e linguaggio medico**, garantendo che i sintomi descritti in modo libero dal paziente vengano convertiti in una forma standardizzata, coerente e clinicamente significativa.

## Flusso logico del servizio

### 1. Input:

- File audio contenente la descrizione vocale dei sintomi, oppure
- Testo scritto manualmente dal paziente.

### 2. Elaborazione:

- Se l'input è audio, viene convertito in testo tramite il modello di **Speech-to-Text Whisper**.
- Successivamente, il testo viene passato a un **modello LLM di correzione linguistica** (Google Gemini 2.0 Flash) che elimina rumori, errori e ambiguità, restituendo una trascrizione clinicamente leggibile.

### 3. Output:

- Un JSON contenente il testo corretto e i metadati (timestamp, tipo di input, nome file).
- Questo output viene poi **inviato al Decision Engine** tramite un **adapter dedicato**.

## Pattern Adapter e decoupling

Il **pattern Adapter** è un principio architetturale appartenente alla famiglia dei *Structural Design Patterns*, il cui scopo è **armonizzare l'interazione tra componenti software che espongono interfacce incompatibili**. Nel contesto di *HealthGate*, questo pattern viene utilizzato per **collegare due microservizi indipendenti Ingestion e Decision Engine senza creare dipendenze dirette** tra di essi. Il microservizio **Ingestion** ha il compito di elaborare input testuali o vocali e restituire un **output semantico corretto**. Tuttavia, non deve conoscere:

1. la struttura interna del **Decision Engine** (modelli LLM, pipeline o logiche cliniche);
2. l'**endpoint HTTP** esatto a cui inviare la richiesta;
3. né il **formato dei dati** che il Decision Engine si aspetta.

Questo isolamento viene realizzato tramite un **adapter dedicato**, che agisce da **traduttore e intermediario** tra i due servizi:

1. Riceve l'output dell'**Ingestion** (corrected\_text o trascrizione).
2. Lo trasforma nel **formato JSON** richiesto dal Decision Engine.
3. Invia la richiesta al Decision Engine tramite HTTP asincrono (httpx.AsyncClient).
4. Restituisce la risposta al chiamante, senza che l'**Ingestion** debba conoscere la struttura della risposta.

In questo modo se in futuro cambia l'endpoint oppure cambia il formato richiesto, basta modificare esclusivamente l'adapter, senza toccare l'intero codice del servizio di ingestion.

Tra i vantaggi principali ritroviamo:

### 1. **Basso accoppiamento (Low Coupling)**

L'Ingestion non dipende più dai dettagli implementativi del Decision Engine. Questo garantisce **stabilità architetturale**: i due servizi possono evolvere in modo indipendente.

### 2. **Alta manutenibilità**

Tutta la logica di comunicazione è isolata in un unico punto (adapter.py), facilmente modificabile in caso di aggiornamenti futuri o migrazioni del Decision Engine.

### 3. **Chiarezza del dominio (Separation of Concerns)**

- a. L'Ingestion si occupa solo di elaborare e pulire i sintomi.
- b. L'Adapter si occupa solo di inviare le richieste al servizio successivo.
- c. Il Decision Engine si occupa solo della decisione clinica.

Questa separazione riduce la complessità cognitiva e migliora la leggibilità del codice.

### 4. **Flessibilità e testabilità**

L'uso dell'adapter permette di simulare facilmente la risposta del Decision Engine nei test, senza dover avviare realmente il microservizio decisionale. In fase di test, l'adapter può essere sostituito con un'implementazione di test che restituisce risposte predefinite.

## Activity Diagram dell'ingestion

Nel contesto di **HealthGate**, la **modellazione SysML dell'Ingestion Service** è utile per **descrivere, attraverso l'Activity Diagram, il flusso operativo di ingestion**, che comprende la gestione dell'input (file audio o testo), la trascrizione automatica, la correzione linguistica e la preparazione dei dati per l'elaborazione decisionale;

Activity Diagram - Ingestion Service (Progettazione Funzionale)



### Descrizione implementativa e scelte progettuali

L'Ingestion Service è stato implementato in **FastAPI** e fa uso di **modelli di deep learning** per la trascrizione (Whisper) e la correzione (Gemini 2.0 Flash).

### Gestione del ciclo di vita e caricamento modelli

Nel file main.py viene definito un contesto lifespan per caricare i modelli una sola volta all'avvio:

```

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Caricamento modello Whisper all'avvio del servizio
    global stt_model, correction_model
    # Creazione cartelle se non esistono
    os.makedirs("audio", exist_ok=True)
    os.makedirs("transcripts", exist_ok=True)
    stt_model = load_model_stt()
    correction_model = load_model_correction()
    print("Modello caricato correttamente.")
    yield
  
```

Caricare i modelli una volta sola migliora le prestazioni, evitando overhead di inizializzazione ad ogni richiesta.

## Funzione di trascrizione e correzione (ingest\_ops.py)

- *transcribe\_audio\_file ()*: usa il modello **Whisper** per convertire il parlato in testo.
- *correct\_transcription ()*: usa il modello **Gemini** per pulire e standardizzare la trascrizione, eliminando rumori e ambiguità, mantenendo però tutte le informazioni cliniche.

```
# Funzione per correggere la trascrizione clinica
def correct_transcription(llm, transcription: str) -> str:

    prompt = f"""
    Sei un assistente medico. Ti fornisco una trascrizione vocale di un paziente che potrebbe contenere:
    - errori ortografici
    - errori grammaticali
    - errori dovuti a rumori ambientali generici (es. traffico, clacson, sirene)
    - errori dovuti a rumori meccanici (es. ventilatori, condizionatori, macchinari generici)
    - errori dovuti a voci non rilevanti (es. persone che parlano in sottofondo, radio, televisione, musica accesa)
    - errori dovuti a rumori fisici legati al paziente o al microfono (es. colpi di tosse, starnuti, click, tocchi al microfono)

    Il tuo compito è:
    - Correggere tutti questi errori
    - Standardizzare i termini medici e riportarli in forma chiara (es. "dolore al petto" -> "dolore toracico").
    - Conservare tutte le informazioni cliniche senza aggiungerne di nuove.
    - Non modificare le quantità (es. farmaci, dosaggi, durata dei sintomi).
    - Restituire soltanto il testo corretto e leggibile, senza spiegazioni aggiuntive.

    ESEMPIO:
    [Input]
    "Ciao ho dolore stomaco nausea vomito sangue pressione bassa 90 60"
    [Output]
    "Il paziente riferisce dolore addominale, nausea e vomito ematico. Presenta ipotensione (PA 90/60 mmHg)."
```

La pulizia semantica delle trascrizioni è fondamentale per evitare che l'LLM del Decision Engine riceva input rumorosi o ambigui, migliorando l'accuratezza clinica del sistema.

## Pattern Adapter e invio al Decision Engine

Nel file adapter.py:

- la classe *DecisionClient* gestisce la comunicazione HTTP asincrona;

```

class DecisionClient:
    def __init__(self, timeout: float = 10.0):
        self.client = httpx.AsyncClient(timeout=timeout)

    async def request(self, headers, payload: Dict[str, Any]) -> Dict[str, Any]:
        try:
            logger.debug(f"[DecisionClient] Invio richiesta a {URL_SERVICE}{ROUTE_SERVICE} con payload: {payload}")

            resp = await self.client.post(f"{URL_SERVICE}{ROUTE_SERVICE}", headers=headers, json=payload)
            resp.raise_for_status()

            logger.debug(f"[DecisionClient] Risposta ricevuta: {resp.text}")
            return resp.json()

        except httpx.RequestError as e:
            # errore di connessione, DNS, timeout, ecc.
            logger.error(f"[DecisionClient] Errore di rete: {str(e)}")
            raise DecisionClientError(f"Errore di rete nel contattare Decision Engine: {str(e)}") from e

        except httpx.HTTPStatusError as e:
            # il server ha risposto ma con status 4xx o 5xx
            logger.error(f"[DecisionClient] Errore HTTP {e.response.status_code}: {e.response.text}")
            raise DecisionClientError(
                f"Decision Engine ha risposto con errore {e.response.status_code}: {e.response.text}"
            ) from e

        except Exception as e:
            # qualsiasi altro errore imprevisto
            logger.exception("[DecisionClient] Errore imprevisto")
            raise DecisionClientError(f"Errore imprevisto nel client Decision: {str(e)}") from e

    async def close(self):
        await self.client.aclose()

```

- la classe *DecisionAdapter* si occupa di costruire il payload corretto per il Decision Engine.

```

class DecisionAdapter:
    def __init__(self):
        self.client = DecisionClient()

    async def send(self, headers, ingestion_output: Dict[str, Any]) -> Dict[str, Any]:
        """
        L'ingestion_output contiene quello che il servizio di ingestion produce.
        Qui facciamo la 'traduzione' nel formato che il Decision Engine si aspetta.
        """

        corrected_text = ingestion_output.get("corrected_text")

        new_headers = {"X-User-Id": headers["X-User-Id"], "Authorization": headers["Authorization"]}
        payload = {"sintomi": corrected_text}

        try:
            response = await self.client.request(headers=new_headers, payload=payload)
            return response

        except DecisionClientError as e:
            # Log di alto livello - utile per capire dove è fallita la catena
            logger.error(f"[DecisionAdapter] Errore nel DecisionClient: {str(e)}")
            raise

        except Exception as e:
            logger.exception("[DecisionAdapter] Errore inatteso durante l'invio")
            raise DecisionClientError(f"Errore inatteso nel DecisionAdapter: {str(e)}") from e

```

Il

pattern Adapter **isola l'Ingestion Service dai dettagli del Decision Engine**. In caso di modifica dell'endpoint, del formato JSON o della logica di decisione, basterà aggiornare l'adapter, non l'intero servizio.

Endpoint principale (/ingestion)

Il file main.py definisce l'endpoint REST per la ricezione dei dati:



- accetta file audio (UploadFile) o testo libero (Form);
- elabora l'input tramite Whisper o Gemini;
- salva la trascrizione;
- invia l'output corretto al Decision Engine tramite l'adapter.

```
output_ingest = {
    "input_type": "audio" if file else "text",
    "filename": file.filename if file else None,
    "corrected_text": corrected_text,
    "timestamp": datetime.now().strftime('%Y-%m-%dT%H-%M-%S')
}
```

### Modulo di caricamento modelli (model.py)

Il caricamento modulare consente di sostituire facilmente i modelli senza modificare il resto del codice.

```
def load_model_stt(): # modello per lo speech-to-text
    return whisper.load_model(MODEL_NAME)

def load_model_correction(): # modello per la correzione delle trascrizioni
    return ChatGoogleGenerativeAI(model="gemini-2.0-flash")
```

### Decision Engine Service

Il **Decision Engine Service** costituisce il cuore logico dell'intero sistema *HealthGate*. È il componente responsabile di **analizzare i sintomi del paziente**, forniti dal servizio di Ingestion e dalla storia clinica, e **determinare se il soggetto debba recarsi o meno al pronto soccorso**, fornendo contestualmente una **motivazione esplicita** basata su evidenze cliniche.

L'architettura del servizio è costruita attorno al paradigma **Retrieval-Augmented Generation (RAG)**. Si tratta di un paradigma che integra i modelli linguistici di grandi dimensioni (**LLMs**) con meccanismi di **retrieval di conoscenza esterna**, colmando così una delle principali limitazioni dei modelli puramente generativi: la tendenza a produrre informazioni inventate (*allucinazioni*) o non aggiornate. In un sistema RAG, il modello non si affida esclusivamente al proprio addestramento pregresso, ma **consulta in tempo reale un archivio di conoscenza esterno** (testi, database, protocolli), combinando la capacità generativa con un processo di ricerca documentale mirata.

L'architettura di un RAG si compone di due componenti fondamentali:

- **Retrieval Component:**  
È responsabile del **recupero dei documenti più rilevanti** rispetto alla query in input (nel caso di *HealthGate*, i sintomi e la storia clinica del paziente). Nel caso di Retrieval basato su ricerca semantica, i documenti e le query vengono rappresentati come vettori

in uno spazio ad alta dimensionalità. La similarità tra query e documenti è calcolata tramite misure, come il **cosine similarity**, consentendo di individuare i testi concettualmente affini, anche in presenza di variazioni linguistiche.

- **Generation Component:**

È il modello generativo vero e proprio, che **sintetizza una risposta** coerente, motivata e linguisticamente fluida, utilizzando il **prompt**, la **query** dell'utente e i documenti recuperati nella fase precedente (che costituiscono un **Enhanced Context**).

In questo modo, il modello produce un testo supportato da evidenze, riducendo drasticamente la probabilità di allucinazioni o errori.

Nel caso del sistema HealthGate, il Rag esegue le quattro fasi tipiche nel modo seguente:

1. **Query Formulation:** il sistema costruisce la query semantica a partire dai sintomi (forniti dall'Ingestion service) e dalla storia clinica elaborata a partire dai Report Clinici precompilati precedentemente dal paziente.
2. **Document Retrieval:** vengono recuperati dal vector store i  $k$  documenti più rilevanti sulla base della similarità vettoriale.
3. **Context Encoding:** la query e i documenti selezionati vengono concatenati e forniti come contesto al modello generativo.
4. **Response Generation:** l'LLM elabora una risposta che integra il contenuto dei documenti recuperati con la propria capacità di ragionamento linguistico.

Prima della costruzione del sistema RAG, è stata necessaria una fase preliminare di **preprocessing dei documenti medici**, utilizzati per costruire la base di conoscenza utilizzata dal sistema. Questa procedura, implementata nel modulo **preprocessing\_docs**, comprende tre passaggi principali:

1. **Estrazione del testo da PDF medici ufficiali** (manuali di triage, linee guida, protocolli sanitari) tramite script di parsing ;
2. Procedura di Cleaning dei testi automatica.
3. Rappresetazione vettoriale e indicizzazione dei testi nel **vector store ChromaDB**, attraverso modelli di embedding.

Il risultato di questa pipeline è una **base di conoscenza clinica** normalizzata, segmentata in chunk e accessibile tramite query semantiche.

Per la costruzione del Sistema RAG è stato necessario:

- Un **retriever semantico** basato su **ChromaDB**, in cui i documenti clinici (preprocessati e resi in formato testuale) sono indicizzati sotto forma vettoriale.

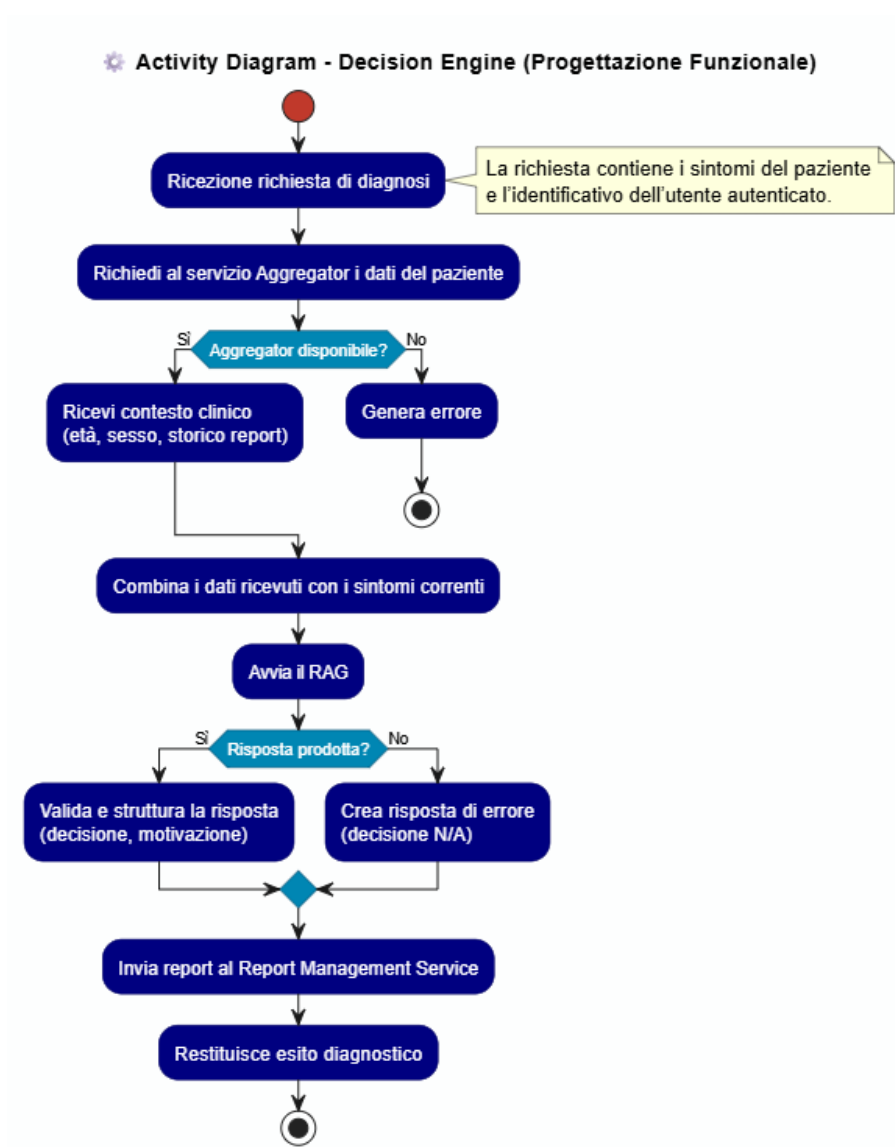
- Un modello di embedding come **a11-MiniLM-L6-v2** disponibile con la libreria di Hugging Face “SentenceTransformers”
- Un LLM come **Gemini 2.0 Flash**

### Graph Building

Il sistema RAG è stato implementato come un **grafo di esecuzione (LangGraph)**, dove ciascun nodo rappresenta uno step funzionale:

1. **Retrieve** per la ricerca dei documenti pertinenti;
2. **Generate** per la generazione della risposta tramite LLM;

### Activity Diagram del Decision Engine



## Descrizione implementativa e scelte progettuali

## Preprocessing e Pre-Training dei documenti clinici

## Estrazione e pulizia del testo (*estrazione\_testo.py*)

Il primo step consiste nell'estrazione e nella pulizia dei testi PDF attraverso un processo a due livelli:

- **Estrazione testo e regex cleaning:**

- I documenti PDF vengono letti con PdfReader (libreria *pypdf*).
- Il testo subisce un primo processing attraverso l'utilizzo di regole (regex) che rimuovono rumori tipici dei testi scansionati.

- **Post-Processing con LLM**

Il testo parzialmente pulito viene inviato sottoposto a un LLM (Gemini 2.0 Flash Lite) per subire un intervento ulteriore di cleaning, laddove non si è potuto intervenire con le regole regex:

```

def model_processing(llm, input_text:str) -> str:

    prompt = f """
    Sei un assistente di pulizia testi.
    Ti fornirò un blocco di testo, originariamente estratto da un PDF, che poi è stato pre-
    processato \\
    con alcune regole di pulizia base con regex (rimozione di trattini, unione di righe spezzate,
    eccetera).

    Tuttavia, il testo potrebbe ancora contenere una serie di errori.

    Il tuo compito è quello di:
    - CORREGGERE errori di spaziatura (es. "elem ento" invece di "elemento")
    - CORREGGERE errori dovuti a OCR imperfetto (es. "deU'infanzia" invece di "dell'infanzia")
    - CORREGGERE O RIMUOVERE errori dovuti a letture errate di OCR di tabelle o elenchi
    - RIMUOVERE presenza indesiderata di numeri di pagina, intestazioni, piè di pagina
    - RIMUOVERE presenza indesiderata di indici o sommari
    - RIMUOVERE testi relativi a intestazioni di sezioni o capitoli che non sono parte del
    contenuto principale \\
    (es. "Università Campus Biomedico di Roma Società Italiana Sistema 118
    Linee di indirizzo terapeutico - Medicina di emergenza territoriale 118")
    - RIMUOVERE testi estratti da copertine o pagine non rilevanti
    - CORREGGERE elenchi che non sono stati formattati correttamente
    - RIMUOVERE citazioni inutili (es. \\
    "La vita di una persona consiste in un insieme di avvenimenti di cui l'ultimo potrebbe
    anche cambiare il senso di tutto l'insieme" \\
    Italo Calvino)
    - RIMUOVERE bibliografie, sitografie e didascalie di immagini
    - CORREGGERE altri errori vari di formattazione

    Tieni in considerazione però che:
    - è IMPERATIVO NON riassumere, NON tradurre e NON aggiungere commenti.
    - il testo ottenuto sarà poi splittato in chunk dal RecursiveCharacterTextSplitter \\
    con chunk size = 1000 e chunk overlap = 150, \\
    quindi vai pure a capo dove necessario per separare paragrafi o sezioni.
    - DEVI RESTITUIRE soltanto il testo ripulito, pronto per essere salvato in un file .txt.
    - I chunk ottenuti saranno indicizzati per l'utilizzo di un RAG a supporto di un LLM \\
    che deve decidere se un dato paziente, che fornisce i sintomi, deve recarsi o meno al Pronto
    soccorso \\
    puoi quindi eliminare dei testi che sono completamente ESULI dall'ambito medico

    NOTA: non preoccuparti se, a valle di questi processamenti, il testo che ottieni è
    completamente vuoto, \\
    tu restituiscimelo lo stesso con ""

    NOTA 2: Per motivi dovuti a rate limiting, ti fornirò un testo che proviene dall'estrazione di
    più pagine

    Ecco il testo da ripulire:
    {input_text}

    """

    response = llm.invoke(prompt)
    return response.content

```

## Segmentazione e vettorializzazione dei documenti (*docs.py*)

Una volta ottenuti i testi puliti, questi vengono caricati e trasformati in embeddings per essere indicizzati nel *vector store*. Si procede a:

- Caricare tutti i file di testo della cartella (ottenuti dal pre-processing) con *TextLoader*.
- Ogni documento viene segmentato in chunk da massimo 1000 caratteri (con overlap di 150), grazie al *RecursiveCharacterTextSplitter*.

```

splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
chunks = splitter.split_documents(docs)

```

- I chunk vengono trasformati in embedding mediante il modello precedentemente citato

```
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
```

- A quel punto, gli embedding ottenuti saranno memorizzati nel vector store, dopo aver opportunamente avviato la connessione a Chroma.

```
client = chromadb.HttpClient(host=CHROMA_HOST, port=CHROMA_PORT, ssl=False)
vector_store = Chroma(client=client, collection_name="collection_name", embedding_function=embeddings)
vector_store.add_documents(documents=chunks)
```

### Architettura del servizio

Nel main del servizio viene definita l'App FastAPI, definendo come fatto per i precedenti servizi, la funzione *lifespan* che determina le operazioni da eseguire all'avvio. Nel caso del Decision Engine Service, è necessario caricare i modelli citati precedentemente e le risorse necessarie al corretto funzionamento. Il caricamento dei modelli avviene in modo asincrono in modo da parallelizzare le operazioni e velocizzare la procedura. Nello stesso main viene definito il principale endpoint esposto dal servizio **/llm/diagnose**.

Nel file *model.py* vengono definite le funzioni per il caricamento dei modelli, da richiamare all'interno della funzione *lifespan*. Si noti che l'approccio di definire le funzioni di caricamento, ha anche l'intento di voler disaccoppiare il codice seguente dalla tipologia di modello utilizzato, consentendo la possibilità futura di cambiare i modelli utilizzati senza ledere al funzionamento del codice.

```
def load_embedding_model(): # modello per lo speech-to-text
    return HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

def load_llm(): # modello per la correzione delle trascrizioni
    return ChatGoogleGenerativeAI(model="gemini-2.0-flash")

def load_vector_store(embedding_model):
    client = chromadb.HttpClient(host=CHROMA_HOST, port=CHROMA_PORT, ssl=False)
    return Chroma(
        client=client,
        collection_name="collection_name",
        embedding_function=embedding_model,
    )
```

Nel file *rag\_setup.py* viene implementato il sistema RAG descritto ad alto livello precedentemente. In particolare, vengono definiti gli step funzionali del grafo di esecuzione.

In particolare, abbiamo il retrieve step deputato alla ricerca dei documenti più affini sulla base della query costruita concatenando i sintomi attuali ai precedenti report clinici del paziente.

```

# Define application steps
def retrieve(state: State, vector_store):
    query_parts = [f"Sintomi attuali: {state['sintomi']}"]

    # Se ci sono report precedenti, aggiungili
    if state.get("reports"):
        reports_text = []
        for report in state["reports"]:
            r_text = (
                f"Report precedente - Sintomi: {report.get('sintomi', '')}. "
                f"Motivazione della visita: {report.get('motivazione', '')}. "
                f"Diagnosi del medico del pronto soccorso: {report.get('diagnosi', '')}. "
                f"Trattamento del medico del pronto soccorso: {report.get('trattamento', '')}. "
            )
            reports_text.append(r_text)

        query_parts.append(" ".join(reports_text))

    # Crea la query finale per la similarity search
    query = "\n".join(query_parts)

    retrieved_docs = vector_store.similarity_search(query, k=6)
    print("\n\n--- RETRIEVED DOCS ---\n\n")
    for doc in retrieved_docs:
        print("Content: " + doc.page_content + "\n")
        print("Source: " + doc.metadata['source'] + "\n")
        print("----- + \n")

    return {"context": retrieved_docs}

```

Segue poi il generate step, nella quale viene definito il prompt, che istruisce l'LLM sul tipo di input che riceve e sulle linee guida da seguire nella generazione della risposta, comprensive anche del formato richiesto alla risposta.

```

def generate(state: State, llm):
    docs_content = "\n\n".join(" " + doc.page_content + " " for doc in state["context"])

    print("\n\n--- DOCS CONTENT ---\n\n")
    print(docs_content)
    print("\n\n----\n\n")

    if len(state['reports']) != 0:
        report_text = "\n\n".join(
            f"- Data Report: {r.get('data', 'N/A')}\n"
            f"- Sintomi: {r.get('sintomi', 'N/A')}\n"
            f"- Motivazioni date dall'LLM sul recarsi o meno al pronto soccorso: {r.get('motivazione', 'N/A')}\n"
            f"- Diagnosi del medico del pronto soccorso: {r.get('diagnosi', 'N/A')}\n"
            f"- Trattamento del medico del pronto soccorso: {r.get('trattamento', 'N/A')}\n"
            for r in state["reports"]
        )
    else:
        report_text = "\n Non sono presenti report clinici precedenti associati a questo paziente \n"

    print("\n\n--- REPORT TEXT ---\n\n")
    print(report_text)
    print("\n\n----\n\n")

    messages = prompt.invoke({"sintomi": state["sintomi"], "age": state["age"], "sex": state["sex"], "report": report_text, "context": docs_content})
    response = llm.invoke(messages, temperature=0)
    return {"answer": response.content}

prompt_template = """
Sei un assistente sanitario virtuale.

Un paziente ti fornisce informazioni sui suoi sintomi.
Riceverai inoltre informazioni da un Retriever che ha accesso a linee guida ospedaliere ufficiali.
Inoltre sempre il retriever ti fornirà uno storico clinico del paziente, se presente.

IMPORTANTE:
- Le tue decisioni devono basarsi esclusivamente sui documenti forniti dal Retriever.
- Non utilizzare conoscenze esterne o supposizioni personali.

Il tuo compito è il seguente:
- Classificare se il paziente deve recarsi al pronto soccorso immediatamente o se non è necessario.
- Fornire una motivazione concisa basata sui documenti forniti.

Input:

- Sintomi attuali del paziente: {sintomi}
- Et  del paziente: {age}
- Sesso del paziente: {sex}
- Precedenti Report clinici del paziente: {report}
- Estratti da linee guida ufficiali relativi ai sintomi: {context}

Rispondi seguendo ESATTAMENTE con un dizionario come segue.

Answer: {
    "decisione": "Pronto soccorso necessario" o "Pronto soccorso non necessario",
    "motivazione": Breve spiegazione della decisione
}

```

La funzione `graph_building()` mette insieme i due step restituendo il riferimento al grafo successivamente invocabile all'atto della richiesta decisionale.

```
def graph_building(vector_store, llm):  
  
    def retrieve_step(state):  
        return retrieve(state, vector_store)  
  
    def generate_step(state):  
        return generate(state, llm)  
  
    graph_builder = StateGraph(State).add_sequence([retrieve_step, generate_step])  
    graph_builder.add_edge(START, "retrieve_step")  
    graph = graph_builder.compile()  
  
    return graph
```

### Endpoint del servizio

Come detto, nel main, viene definito l'endpoint principale del servizio. La richiesta (di tipo "Post") deve essere accompagnata da un opportuno payload contenente i sintomi correnti del paziente (il testo ottenuto a seguito del processing nell'Ingestion Service). A quel punto la funzione `diagnose`, associata all'endpoint, procederà nel ricavare dall'header l'id del paziente (Ricordando che è il gateway a inserire questi campi ulteriori dell'header).

Con l'id ricavato, verrà inoltrata una richiesta all'Aggregator Service, il cui obiettivo è ricavare il contesto clinico del paziente (funzionale alla decisione e alla componente di Retrieval). A quel punto viene invocato il grafo che restituirà un'opportuna risposta da cui estraiamo due campi fondamentali:

- **Decisione**, che restituisce un esito binario riguardante la necessità di andare o meno al pronto soccorso.
- **Motivazione**, che motiva la decisione presa.

A quel punto, è responsabilità del servizio stesso quella di costruire un report da inviare al Report Management Service (si rende necessario per disimpegnare l'utente stesso nel dover esplicitamente richiedere il salvataggio del report).

Dopo aver inviato la richiesta al Report, la risposta del Decision sarà inviata in modo da consentire all'utente di poter visualizzare l'esito decisionale nel frontend.



```

@app.post("/llm/diagnose", response_model=Dict)
async def diagnose(data: DiagnoseRequest, request: Request):
    """
    Endpoint che riceve un testo clinico e restituisce JSON strutturato
    dopo correzione, estrazione e validazione.
    """
    try:
        print("Ricevuta richiesta")
        user_id = request.headers.get("X-User-Id")
        resp = await client.get(f"{AGGREGATOR_SERVICE}/{AGGREGATOR_ROUTE}/{user_id}")
        resp.raise_for_status()

        resp = resp.json()
        print("apost")
        print(resp)
        response = graph.invoke({"sintomi": data.sintomi, "age": resp['age'], "sex": resp['sex'], "reports": resp['reports']})

        print("Ripost")

        # Verifica che 'answer' sia presente e non None
        raw_answer = response.get("answer")
        print(raw_answer)

        if not raw_answer:
            # fallback in caso di risposta vuota o assente
            return {"decisione": "N/A", "motivazione": "LLM non ha restituito dati"}

        answer_text = re.sub(r"```json\s*"```$", "", raw_answer.strip(), flags=re.MULTILINE)

        answer_json = json.loads(answer_text)

        # Costruisci il payload per /report
        report_payload = {
            "patient_id": resp['patient_id'],
            "social_sec_number": resp['social_sec_number'],
            "date": datetime.now().strftime("%Y-%m-%d"),
            "sintomi": data.sintomi,
            "motivazione": answer_json.get("motivazione", ""),
            "diagnosi": "",
            "trattamento": ""
        }

        print(report_payload)

        response = await client.post(f"{REPORT_SERVICE_URL}/{REPORT_ROUTE}", json=report_payload)

        return answer_json
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

```

## Aggregator Service

Il microservizio **Aggregator**, come si è avuto modo di vedere nel servizio precedente, rappresenta un componente fondamentale dell'architettura in quanto progettato per **unificare i dati clinici e anagrafici** provenienti da Database sotto la responsabilità di altri servizi (*Auth Service* e *Report Service*) e renderli disponibili al **Decision Engine Service**.

Il suo obiettivo è fornire un **“profilo del paziente”** (comprendente età, sesso, e storico clinico) per supportare i processi decisionali e garantire che le decisioni prese siano basate su informazioni aggiornate e coerenti.

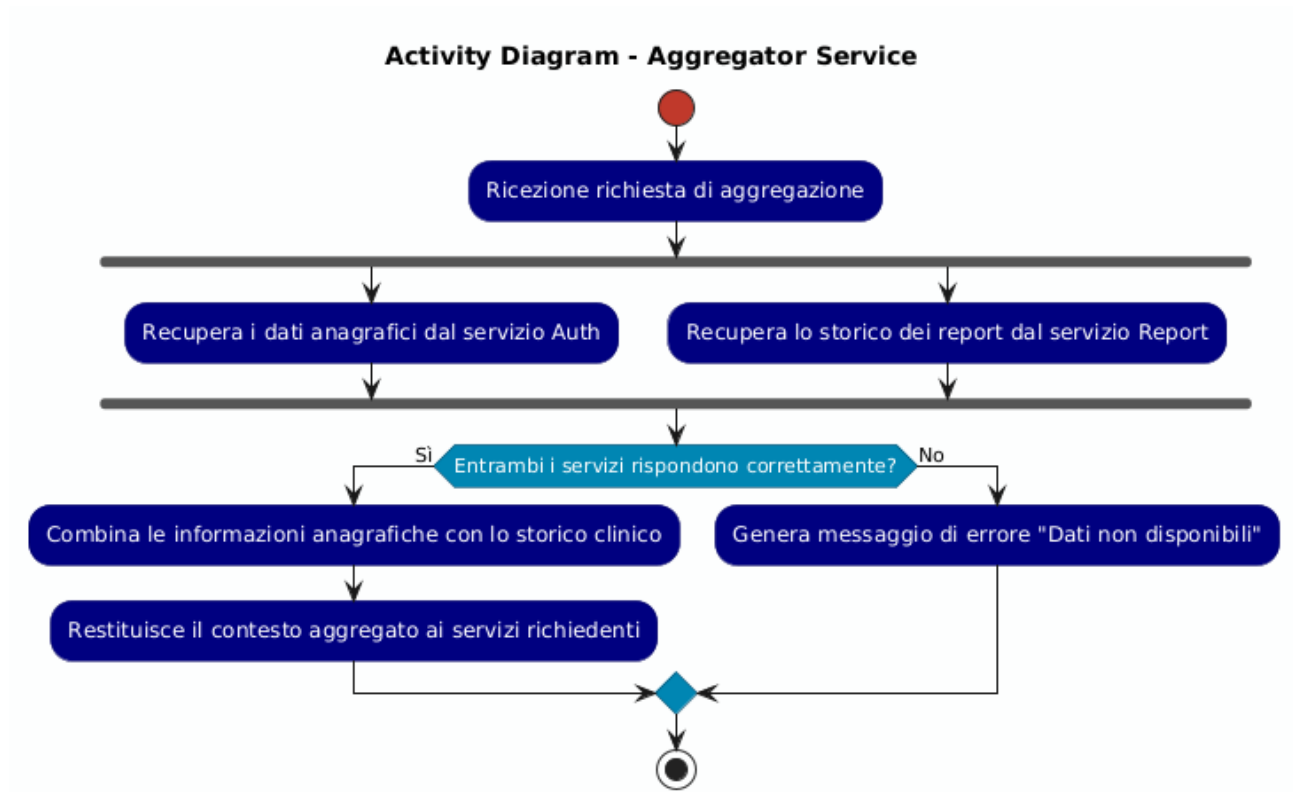
## Motivazione Architetture

In un'architettura a microservizi, ogni componente ha un dominio specifico di responsabilità. Nel caso di HealthGate:

- Il microservizio **Auth** gestisce le informazioni anagrafiche e di autenticazione.
- Il microservizio **Report** conserva la cronologia clinica e i referti medici.
- Il **Decision Engine**, per generare una valutazione corretta, deve accedere a entrambi.

Tuttavia, collegare direttamente il Decision Engine a più microservizi introdurrebbe **elevato accoppiamento**, riducendo la manutenibilità e aumentando la complessità del codice. Per questo motivo, è stato introdotto l'**Aggregator Service**, che funge da **strato di orchestrazione dei dati** (*Data Orchestration Layer*). Il suo compito è **interrogare i microservizi di origine, aggregare i dati ricevuti e restituirli in un formato unificato e coerente**.

## Activity Diagram dell'Aggregator



### Descrizione Implementativa

Nel file main, viene definito il servizio con il suo unico endpoint e il comportamento all'avvio.

```
@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.client = httpx.AsyncClient(timeout=20.0)
    app.state.today = date.today()
    yield # to be executed at shutdown
    await app.state.client.aclose()
```

Il suo unico endpoint **/aggregator** (da richiamare con una richiesta HTTP “GET”) riceve come input l'id dell'utente

- Interroga l'**Auth Service** per ottenere i dati anagrafici strettamente necessari ai fini decisionali come sesso e data di nascita.
- Interroga il **Report Service** per recuperare la lista dei referti clinici associati al paziente.

```
auth_resp = await app.state.client.get(f"{AUTH_SERVICE_URL}"
{ROUTE_AUTH_SERVICE}/{patient_id}")
report_resp = await app.state.client.get(f"{REPORT_SERVICE_URL}"
{ROUTE_REPORT_SERVICE}/{patient_id}")
```

Ricevuti i dati, li elabora in modo tale da calcolare dinamicamente l'età del paziente a partire dalla data di nascita e convertendola nel formato opportuno. A quel punto, combina i dati anagrafici e clinici in un unico dizionario JSON.

```
@app.get("/aggregator/{patient_id}")
async def get_patient_context(patient_id: int):
    auth_resp = await app.state.client.get(f"{AUTH_SERVICE_URL}/{ROUTE_AUTH_SERVICE}/{patient_id}")
    report_resp = await app.state.client.get(f"{REPORT_SERVICE_URL}/{ROUTE_REPORT_SERVICE}/{patient_id}")

    if auth_resp.status_code != 200 or report_resp.status_code != 200:
        raise HTTPException(status_code=500, detail="Failed to fetch data")

    auth_data = auth_resp.json()
    # ♦ Converti la data ISO in datetime.date

    # ♦ Converti la stringa ISO in datetime.date
    birth_date = datetime.fromisoformat(auth_data['birth_date']).date()

    age = app.state.today.year - birth_date.year - (
        (app.state.today.month, app.state.today.day) < (birth_date.month, birth_date.day)
    )

    print("Age:", age)

    report_data = report_resp.json()

    print(report_data)
    # ♦ Estrazione campi dai report
    reports_list = [
        {
            "data": r["date"],
            "motivazione": r["motivazione"],
            "diagnosi": r["diagnosi"],
            "sintomi": r["sintomi"],
            "trattamento": r["trattamento"]
        }
        for r in report_data
    ]

    # ♦ Risposta aggregata
    return {
        "patient_id": patient_id,
        "social_sec_number": auth_data['social_sec_number'],
        "age": age,
        "sex": auth_data["sex"],
        "reports": reports_list
    }
```

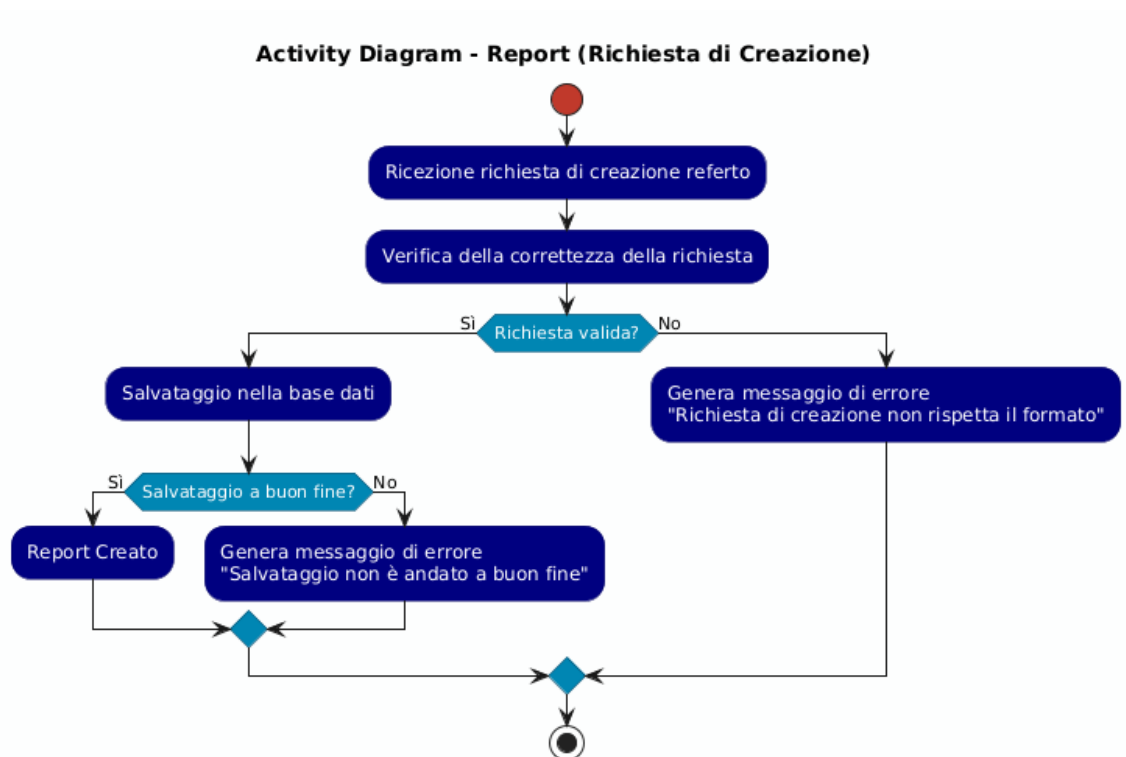
### Report Management Service

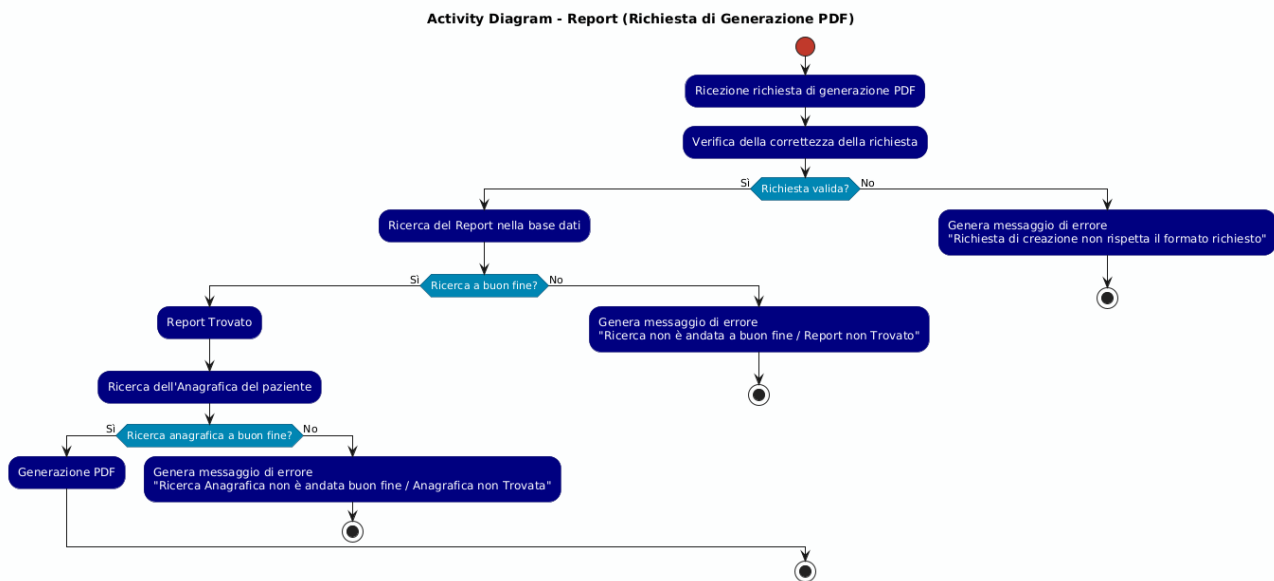
Il **Report Management Service** è l'ultimo servizio dell'architettura. È il componente incaricato di gestire l'intero ciclo di vita dei **referti clinici**, assicurando **persistenza, tracciabilità e accessibilità** dei dati medici.

La sua funzione è centrale per garantire che ogni decisione clinica venga **archiviata** in modo sicuro e sia **consultabile** da pazienti e operatori sanitari.

Il servizio, in questione, riceve un report dal Decision Engine, lo memorizza nel database dedicato e lo rende disponibile alla consultazione e al download in formato **PDF**.

### Activity Diagram del Report Management





## Architettura Interna del Servizio

L'architettura interna è composta da quattro moduli principali:

- **Main.py:** nel main viene definito il servizio e gli endpoint che espone verso l'esterno. Essi sono elencati di seguito:
  - **POST /report** → Consente la creazione di un nuovo referto;
  - **GET /reports/id/{patient\_id}** → Consente il recupero di tutti i referti di un paziente associato all'identificativo fornito;
  - **GET /reports/ssn/{social\_sec\_number}** → Consente il recupero di tutti i referti di un paziente associato al codice fiscale fornito;
  - **GET /report/{report\_id}** → Consente il recupero del referto associativo all'identificativo fornito;
  - **PUT /report/{report\_id}** → Consente l'aggiornamento del referto associato all'identificativo fornito;
  - **GET /report/pdf/{report\_id}** → Consente di generare un file in formato pdf relativo al referto associato all'identificativo fornito.
- **Database Layer:** in questo file sono definite le funzioni il cui scopo è quello di gestire la connessione al Database utilizzato per la gestione dei report (in questo caso **MongoDB**) e di realizzare le funzionalità CRUD.
- **Report\_ops.py:** Contiene le funzionalità necessarie per recuperare l'anagrafica di un paziente (le informazioni non accessibili al decision engine) necessarie all'atto di generazione del file PDF relativo a un referto.
- **Presentation Layer (pdf\_gen.py e scheda.html):** Si occupa della generazione del referto in formato **PDF**. Si utilizza la libreria **Jinja2** per compilare un template HTML con i dati del referto, e **xhtml2pdf** per convertirlo in un documento PDF compatto.

## Continuous monitoring e testing

La fase di continuous monitoring e testing rappresenta l'ultima parte del ciclo di sviluppo del prototipo e ha l'obiettivo di verificare la stabilità, l'affidabilità e il corretto funzionamento del sistema, nonché di gestire la distribuzione locale dell'intera architettura a microservizi. In questa fase vengono eseguite attività di testing funzionale e di integrazione, oltre al monitoraggio continuo delle prestazioni del sistema durante l'esecuzione in ambiente containerizzato. L'uso di strumenti di containerizzazione come **Docker** consente di simulare un ambiente di produzione reale in modo controllato, garantendo isolamento, portabilità e ripetibilità dei test.

Gli obiettivi principali di questa fase sono:

- Verificare il corretto funzionamento dei microservizi e delle loro interazioni;
- Testare la comunicazione attraverso l'API Gateway e la gestione dei token di autenticazione;
- Controllare la persistenza e la coerenza dei dati nei database PostgreSQL e MongoDB;
- Monitorare lo stato e le performance del sistema in esecuzione locale;
- Eseguire il deployment dell'intero ambiente in modo automatizzato e riproducibile tramite Docker Compose.

## Testing del sistema

Il testing è stato condotto a diversi livelli, per garantire la copertura delle funzionalità principali del sistema:

### Testing funzionale

Sono stati testati gli endpoint REST di ciascun microservizio utilizzando strumenti come Invoke-RestMethod, per assicurare la correttezza delle risposte e la gestione degli errori. Esempi di test eseguiti includono:

- Autenticazione di un utente tramite POST /login;
- Recupero dei report associati a un paziente tramite GET /reports/id/{patient\_id};
- Generazione e download del file PDF tramite l'endpoint del Report Management Service;
- Verifica del corretto instradamento delle richieste attraverso l'API Gateway.

### Testing di integrazione

È stata verificata la comunicazione tra i microservizi, assicurando che i dati scambiati attraverso il Gateway risultassero coerenti e sincronizzati. Durante i test di integrazione, è

stata prestata particolare attenzione alla sequenza delle chiamate API e al corretto passaggio dei token JWT, garantendo che le richieste non autorizzate venissero bloccate.

## Testing del database

Sono stati eseguiti test di connessione e scrittura su entrambi i database:

- In PostgreSQL, per verificare la corretta creazione e autenticazione degli utenti;
- In MongoDB, per assicurare l’inserimento, la lettura e l’aggiornamento dei documenti relativi ai report clinici.

I test hanno confermato la stabilità delle connessioni e l’integrità dei dati durante le operazioni CRUD (Create, Read, Update, Delete).

## Testing del frontend

Infine, sono stati eseguiti test di interazione tramite l’interfaccia utente. È stato verificato che il frontend in React fosse in grado di comunicare correttamente con l’API Gateway, mostrando in modo dinamico i dati restituiti dai microservizi e gestendo correttamente le sessioni di autenticazione.

## Testing del modello decisionale

Oltre ai test funzionali e di integrazione, è stata realizzata una fase di **testing** del modello di ragionamento (Decision Engine). L’obiettivo di questa fase è validare la **coerenza**, **robustezza** e **affidabilità clinica** delle decisioni generate dal modello RAG (Retrieval-Augmented Generation), che costituisce il nucleo logico del sistema. Per garantire un’analisi completa sono stati implementati diversi livelli di verifica automatizzata:

- **Operational Testing:** Simula casi clinici reali per verificare che il sistema produca decisioni corrette (“Pronto soccorso necessario” o “non necessario”) in base ai sintomi e alla storia clinica del paziente. Il test è eseguito tramite il file *tester.py*, che confronta la motivazione generata dal modello con quella attesa utilizzando **similarità semantica**. La metrica principale è la *Operational Accuracy*, calcolata in base al numero di casi in cui la decisione e la motivazione coincidono con l’atteso.

```
# soglia minima di similarità per considerare il test superato
SIMILARITY_THRESHOLD = 0.75

def semantic_similarity(model, text1: str, text2: str) -> float:
    """
    Restituisce la similarità semantica (cosine similarity) tra due testi.
    Valori vicini a 1 indicano alta somiglianza.
    """
    embeddings = model.encode([text1, text2], convert_to_tensor=True)
    score = util.cos_sim(embeddings[0], embeddings[1])
    return float(score.item())
```

- **LLM Testing:** Nel file `test_with_llm.py` è stato introdotto un approccio innovativo basato sul paradigma **LLM as Evaluator**, dove un secondo modello LLM valuta le risposte del Decision Engine.

```
def llm_evaluator(llm, sintomi, age, sex, reports, decision, motivation):

    if len(reports) != 0:
        report_text = "\n".join(
            f"- Data Report: {r.get('data', 'N/A')}\n"
            f"- Sintomi: {r.get('sintomi', 'N/A')}\n"
            f"- Motivazioni date dall'LLM sul recarsi o meno al pronto soccorso: {r.get('motivazione', 'N/A')}\n"
            f"- Diagnosi del medico del pronto soccorso: {r.get('diagnosi', 'N/A')}\n"
            f"- Trattamento del medico del pronto soccorso: {r.get('trattamento', 'N/A')}\n"
            for r in reports
        )
    else:
        report_text = "\n Non sono presenti report clinici precedenti associati a questo paziente \n"

    prompt_template = """
    Sei un assistente sanitario virtuale incaricato di valutare risposte cliniche generate da un modello AI.
    Il compito del modello in questione è quello di prendere dei sintomi raccolti da un paziente, oltre alle sue informazioni
    personali \\\
    e ai precedenti report clinici del paziente, e decidere se è necessario che si debba recare o meno al pronto soccorso \\\
    oltrechè dare motivazione della decisione presa.

    Gli input che ti verranno forniti sono questi (in parte sono gli stessi dati al modello di AI):
    - Sintomi attuali del paziente: {sintomi}
    - Et : {age}, Sesso: {sex}
    - Precedenti report clinici: {reports}
    - Decisione presa dal modello ("Pronto soccorso necessario" o "Pronto soccorso non necessario"): {decision}
    - Risposta generata dal modello: {motivation}

    Il tuo compito   il seguente:
    1. Valutare se la decisione presa   corretta.
    2. Valuta se la motivazione data   corretta e sufficiente rispetto ai dati clinici forniti.
    3. Fornisci a quel punto:
        - "output": "True" se la decisione presa   corretta / "False" altrimenti.
        - "score": punteggio da 0 a 1 a seconda della qualit  della motivazione fornita in relazione alle linea guida (1 = ottima, 0
    = scarsa)
        - "commento": breve spiegazione del punteggio e dell'output assegnati.

    Regole:
    - Non inventare nuove informazioni o diagnosi.
    - Rispondi solo in formato JSON.

    Esempio di output atteso DA RISPETTARE OBBLIGATORIAMENTE:
    {{
        "output": True
        "score": 0.9,
        "commento": "La risposta del modello coglie correttamente l'infezione respiratoria e menziona i report precedenti rilevanti."
    }}

    """

    prompt = PromptTemplate(
        template=prompt_template,
        input_variables=["sintomi", "age", "sex", "reports", "model_output"]
    )

    formatted_prompt = prompt.format(
        sintomi=sintomi,
        age=age,
        sex=sex,
        reports=report_text,
        decision = decision,
        motivation = motivation
    )

    # Chiamata all'LLM
    response = llm.invoke(formatted_prompt, temperature=0)
    return response
```



L'evaluator riceve in input:

- i sintomi e i report clinici del paziente,
- la decisione e la motivazione fornite dal modello, e restituisce una valutazione in formato JSON contenente:

```
{
  "output": true,
  "score": 0.9,
  "commento": "La decisione è coerente con i sintomi e le linee guida."
}
```

In questo modo si ottiene una metrica oggettiva di *qualità motivazionale* (da 0 a 1) e di *correttezza decisionale*.

- **Robustness Testing:** Implementato in main.py, verifica la stabilità del modello rispetto a **piccole variazioni linguistiche** dei sintomi (“dolore toracico” ovvero “pressione al petto con respiro corto”). Il test misura la *Robustness Score*, ovvero la percentuale di decisioni coerenti tra casi semanticamente simili.

```
def robustness_testing(graph):
    sintomi_base = "dolore toracico e difficoltà respiratoria"
    perturbazioni = [
        "dolore al petto e respiro corto",
        "leggero dolore toracico e affanno",
        "pressione al petto con respiro affannoso",
    ]

    base_state = {"sintomi": sintomi_base, "age": 50, "sex": "M"}
    base_decision = json.loads(graph.invoke(base_state)["answer"])[0]["decisione"]

    consistenti = 0
    for s in perturbazioni:
        decision = json.loads(graph.invoke({"sintomi": s, "age": 50, "sex": "M"})["answer"])[0]["decisione"]
        if decision == base_decision:
            consistenti += 1

    robustness_score = consistenti / len(perturbazioni)
    return round(robustness_score, 2)
```

## Continuous monitoring

In prospettiva futura, il sistema potrebbe essere integrato con strumenti di monitoraggio avanzato come Prometheus (monitoraggio di basso livello), Grafana (visualizzatore dati) o ELK Stack (Elasticsearch, Logstash, Kibana, per la raccolta, l'analisi e la visualizzazione dei log), per consentire una raccolta e visualizzazione centralizzata dei dati di performance e diagnostica.

## Local Deployment con Docker

L'intero sistema HealthGate è stato distribuito in locale tramite **Docker Compose**, che consente di gestire più container in modo coordinato. Ogni microservizio è descritto nel file `docker-compose.yml`, dove sono specificati:

- l'immagine Docker di riferimento;

- le porte esposte;
- le variabili d'ambiente;
- i volumi per la persistenza dei dati;
- la rete condivisa microservices-network che consente la comunicazione interna tra i container.

L'ambiente viene avviato con il comando:

```
docker-compose up -d --build
```

Questo comando:

1. Esegue la build delle immagini per ogni microservizio;
2. Crea e avvia automaticamente i container;
3. Collega i microservizi tra loro sulla rete Docker dedicata;
4. Inizializza i database e li rende accessibili ai rispettivi servizi.

Al termine del processo di deployment, i servizi risultano accessibili ai seguenti indirizzi:

- API Gateway: <http://localhost:8010>
- Authentication Service: <http://localhost:8001>
- Decision Engine Service: <http://localhost:8002>
- Ingestion Service: <http://localhost:8003>
- Report Management Service: <http://localhost:8004>
- Aggregator Service: <http://localhost:8005>
- Frontend: <http://localhost:8501>

## Risultati del deployment

Il deployment locale ha permesso di eseguire l'intera infrastruttura in modo stabile e isolato, con i seguenti risultati:

- Tutti i microservizi comunicano correttamente tra loro tramite la rete Docker condivisa;
- I database PostgreSQL e MongoDB risultano accessibili e sincronizzati con i rispettivi servizi;
- Il frontend interagisce correttamente con l'API Gateway;
- È possibile simulare scenari reali di utilizzo del sistema direttamente in ambiente locale.

L'ambiente containerizzato riproduce quindi in modo fedele la struttura di un sistema distribuito, offrendo la possibilità di eseguire test continui, aggiornamenti incrementali e scalabilità orizzontale dei servizi.

# Conclusioni

Il progetto HealthGate è un esempio dell'uso delle moderne tecniche di design del software basate sull'architettura a microservizi. Lo sviluppo è stato effettuato in modo incrementale ed sperimentale, tipico degli ambienti guidati dall'innovazione, nel tentativo di costruire un sistema flessibile, scalabile e facilmente estensibile.

Partendo dalla fase di Analisi dei Requisiti, sono state catturate le principali esigenze degli stakeholder e tradotte in requisiti funzionali e non funzionali. Questo ha costituito la base per la successiva fase di Design, in cui sono state determinate le decisioni architettoniche, la struttura logica dei componenti e i metodi di comunicazione tra i servizi.

L'uso della modellazione SysML ha aiutato a rappresentare graficamente la complessità del sistema, migliorando così la comprensione delle interazioni e facilitando la tracciabilità dei requisiti.

Nella fase di Sviluppo del Prototipo, è stato realizzato un prototipo funzionante per tradurre le decisioni di design in un artefatto concreto. L'integrazione di FastAPI per i microservizi e l'uso dei database PostgreSQL e MongoDB hanno fornito una gestione dei dati efficiente e organizzata. Il frontend creato in Streamlit ha reso l'interfaccia utente interattiva e user-friendly, mentre il deploy di JWT ha fornito un livello appropriato di autenticazione utente e protezione del sistema.

Tutti i microservizi sono stati containerizzati con Docker, rendendo l'ambiente di sviluppo portatile, isolato e facilmente replicabile.

La successiva fase di Continuous Monitoring & Local Deployment ha permesso di verificare la solidità del sistema, la coerenza dei flussi informativi e la stabilità delle connessioni tra i servizi. Attraverso Docker Compose è stato possibile avviare e coordinare l'intera infrastruttura, eseguendo il sistema in locale in condizioni simili a quelle di un ambiente di produzione.

I test funzionali e di integrazione hanno confermato la corretta comunicazione tra i microservizi, la gestione sicura dell'autenticazione e la persistenza dei dati nei database.

Dal punto di vista metodologico, l'approccio seguito ha dimostrato l'efficacia del modello iterativo e modulare nella realizzazione di sistemi complessi. Ogni componente, essendo indipendente, può essere aggiornato, sostituito o scalato senza compromettere il funzionamento globale del sistema, garantendo così manutenibilità e affidabilità a lungo termine.

In prospettiva futura, il progetto HealthGate potrebbe essere ulteriormente evoluto mediante:

- l'integrazione di un sistema di monitoraggio avanzato (es. Prometheus o Grafana);
- la container orchestration tramite Kubernetes per la distribuzione su larga scala;
- l'adozione di pipeline di Continuous Integration / Continuous Deployment (CI/CD);
- l'integrazione di moduli di intelligenza artificiale e analisi predittiva nel Decision Engine Service.

In conclusione, il percorso progettuale ha permesso di ottenere un prototipo solido, coerente e funzionante, che dimostra la validità dell'architettura proposta e costituisce una base concreta per futuri sviluppi orientati alla produzione e alla sperimentazione reale in ambito sanitario digitale.