Contents

Manutenzione ed Evoluzione del sistema LLM-SmartContractScanner

Nicola Tortora, Gaspare Galasso

July 5, 2025

1 Introduzione

La crescente diffusione delle blockchain e degli smart contract ha portato alla necessità di strumenti avanzati per l'analisi automatica del codice, in particolare per l'individuazione di vulnerabilità che potrebbero compromettere la sicurezza e l'integrità dei sistemi decentralizzati. In questo contesto, il presente progetto propone un sistema di vulnerability assessment automatico che combina la capacità dei Large Language Models (LLM) con tecniche di Retrieval-Augmented Generation (RAG).

Questa prima versione del sistema è focalizzata esclusivamente su contratti scritti per la blockchain Algorand (Pyteal), ma la metodologia è stata progettata con un'architettura modulare e scalabile, che consentirà in futuro l'estensione ad altre piattaforme come Solana.

1.1 Panoramica

Il sistema ha come obiettivo l'analisi automatica di smart contract per l'identificazione delle vulnerabilità. Per ottenere una valutazione più accurata e ridurre il numero di falsi positivi, il sistema combina due approcci complementari: da un lato, l'analisi diretta del codice mediante modelli linguistici avanzati (LLM), dall'altro l'uso di un motore di retrieval semantico per il recupero di contratti simili, con vulnerabilità note, da un database vettoriale. Le informazioni dei contratti simili vengono poi fornite in contesto agli LLM per un analisi più accurata e risposte più corrette.

Il flusso generale prevede tre fasi principali:

- **Analisi Generale del Codice**: identificazione preliminare di vulnerabilità sospette tramite LLM.
- **Retrieval Semantico**: il codice sotto test viene convertito in un *embedding vector* e confrontato con una base di conoscenza di codice vulnerabile.
- **Analisi Dettagliata**: per ciascuna vulnerabilità sospetta, viene effettuata un'analisi mirata utilizzando anche il contesto derivato dal retrieval.

Ogni componente del sistema è stato progettato per operare in maniera modulare, facilitando la manutenzione, l'estensione e l'integrazione con strumenti futuri.

1.1.1 Architettura generale

Lo schema in Figura 1 spiega il funzionamento dei componenti principali (retrieval, analisi iniziale, analisi specifica).

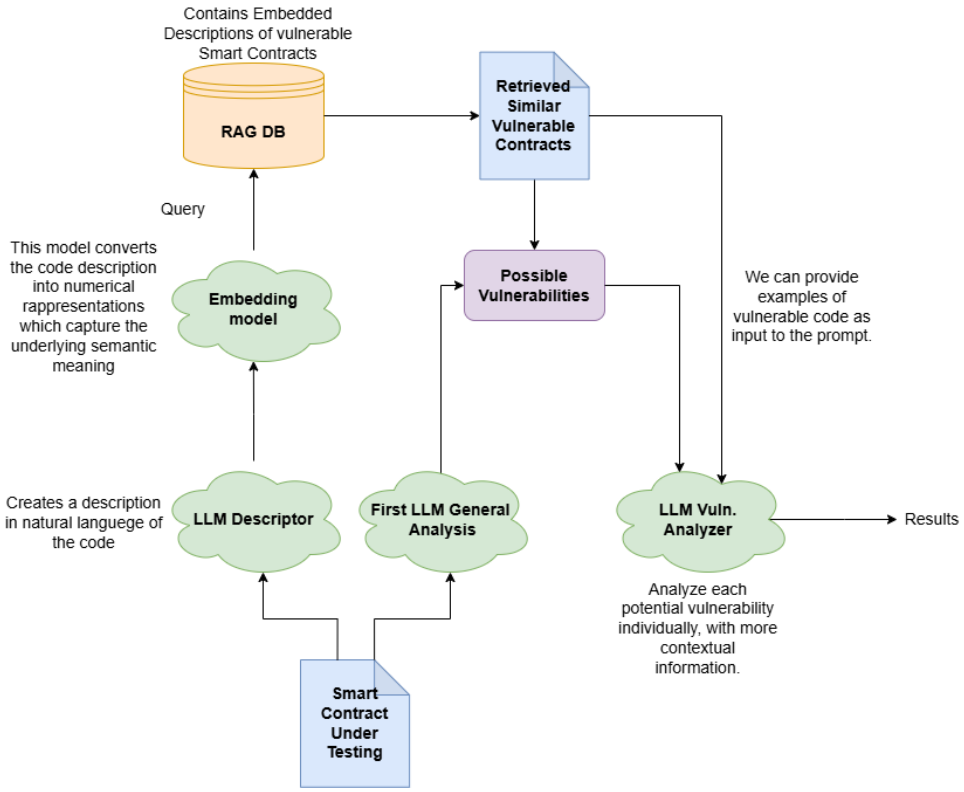

Figure 1: Architettura generale del sistema

1.1.2 Tecnologie Utilizzate

Linguaggi e Framework

- **Python**: linguaggio di programmazione principale per lo sviluppo di tutti i moduli del sistema.
- **FastAPI**: framework leggero per la realizzazione di API RESTful, utilizzate per esporre il servizio del database.

Modelli e LLM

- **Large Language Models (LLM)**: modelli linguistici avanzati (come Deepseek-V3 o equivalenti open-source).
- **Modelli di embedding**: modelli come `BAAI/bge-codehkunlp/instructor-xl` impiegati per generare rappresentazioni vettoriali del codice in uno spazio semantico.

Database

- **Qdrant**: motore di ricerca vettoriale utilizzato per effettuare operazioni di retrieval semantico basate su similarità tra embedding.

Strumenti di Supporto

- **Docker**: utilizzato per il deploy del database vettoriale *Qdrant*. L'uso di container Docker consente l'avvio rapido del servizio,

1.2 Requisiti del Sistema

Il sistema proposto per l'analisi automatica di smart contract PyTeal è stato progettato secondo un insieme di requisiti funzionali e non funzionali che ne guidano l'implementazione, la valutazione e l'evoluzione futura. In questa sezione vengono elencati e descritti in dettaglio.

1.2.1 Requisiti Funzionali

I requisiti funzionali definiscono il comportamento atteso del sistema e le operazioni che ciascun modulo deve essere in grado di svolgere.

1. Interfaccia

- **RF1.1** – Il sistema deve accettare in input uno smart contract.

2. Analisi Generale

- **RF2.1** – Il sistema deve invocare un LLM per generare un'analisi generale del codice.
- **RF2.2** – Il sistema deve identificare un insieme di potenziali vulnerabilità nel codice.
- **RF2.3** – In input possono essere specificate il numero di potenziali vulnerabilità da restituire.

3 Analisi Dettagliata per Vulnerabilità

- **RF3.1** – Il sistema deve analizzare ogni vulnerabilità sospetta rilevata.
- **RF3.2** – Il sistema deve costruire un prompt specializzato con il codice e il contesto simile.
- **RF3.3** – Il sistema deve inviare il prompt a un LLM per un'analisi approfondita della vulnerabilità.
- **RF3.4** – Il sistema deve classificare la vulnerabilità come vera o falsa positiva sulla base della risposta.

4. Utilizzo dei LLM

- **RF4.1** – Il sistema deve supportare l'utilizzo di modelli linguistici di grandi dimensioni (LLM) per le analisi generali e dettagliate del codice.
- **RF4.2** – Il sistema deve supportare l'utilizzo di LLM tramite API esterne.
- **RF4.3** – Il sistema deve supportare l'utilizzo di modelli open-source ospitati su Hugging Face, eseguibili localmente.

5. Sistema di Retrieval

- **RF5.1** – Il sistema deve generare una descrizione in linguaggio naturale del codice tramite LLM.
- **RF5.2** – Il sistema deve convertire la descrizione testuale in un embedding vettoriale.
- **RF5.3** – Il sistema deve confrontare il vettore con un database vettoriale Qdrant contenente contratti vulnerabili.
- **RF5.4** – Il sistema deve recuperare i K contratti semanticamente più simili.
- **RF5.5** – Il sistema deve includere, alle potenziali vulnerabilità, anche quelle presenti nei contratti simili recuperati.
- **RF5.6** – In input possono essere specificati il numero di contratti vulnerabili da restituire.

6. Accesso al DataBase

- **RF6.1** Il sistema deve inviare vettore a Qdrant e recuperare i primi K elementi con punteggio di similarità più alto.
- **RF6.2** I contratti recuperati devono essere restituiti al modulo di analisi e le vulnerabilità estratte dai loro payload.

1.3 Design del Sistema

Il sistema è progettato secondo un'architettura modulare, che consente elevata manutenibilità, testabilità e possibilità di estensione.

1.3.1 Architettura

L'architettura del sistema si articola in cinque macro-componenti:

- **User Interface (UI)**: permette all'utente di avviare l'analisi di un contratto smart.
- **Analysis Modules**: include i moduli `ModelAnalysis` e `VulnAnalysis` per l'analisi iniziale e approfondita.
- **Retrieval Module**: si occupa della descrizione testuale, embedding e retrieval semantico.
- **LLM API**: interfaccia di comunicazione con i modelli linguistici.
- **Vector DB (Qdrant)**: database vettoriale contenente contratti vulnerabili noti.

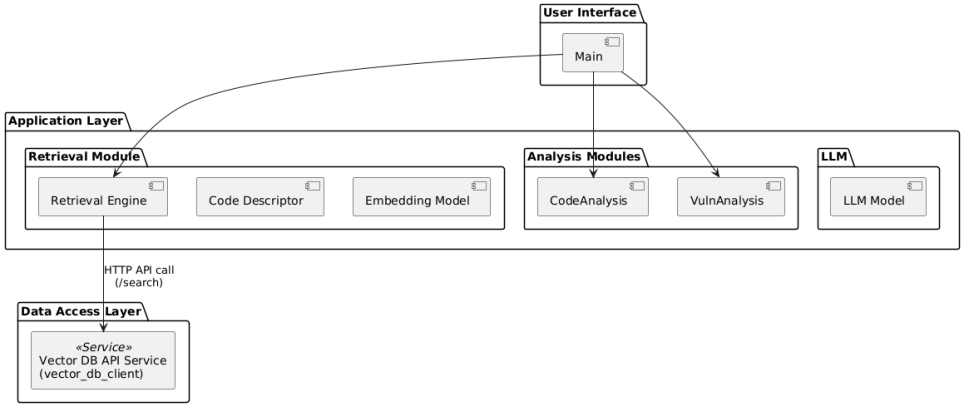

Figure 2: Architettura del sistema

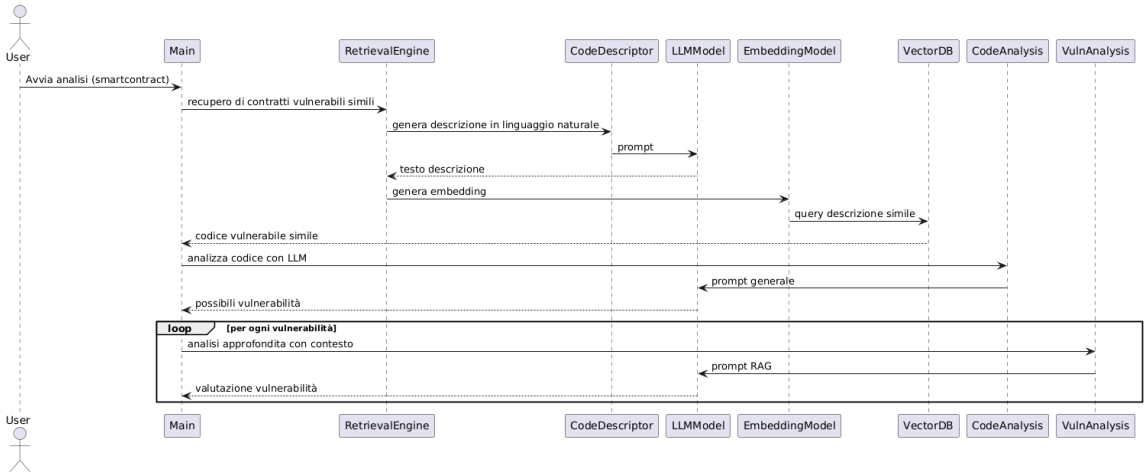

Figure 3: Diagramma di sequenza delle interazioni tra i moduli

1.3.2 Sequence Diagram

La Figura mostra il diagramma di sequenza del sistema, evidenziando le principali interazioni tra i componenti durante l'esecuzione dell'analisi.

1. L'utente carica uno smartcontract tramite la UI.
2. Il modulo `RetrievalModule` genera una descrizione naturale tramite LLM, calcola l'embedding e recupera contratti simili da `VectorDB`.
3. Il modulo `CodeAnalysis` analizza il codice e restituisce vulnerabilità sospette.
4. Per ogni vulnerabilità, `VulnAnalysis` costruisce un prompt RAG e chiede al LLM di classificare il rischio.

1.3.3 Class Diagram

La Figura descrive la struttura ad oggetti del sistema, evidenziando le relazioni tra le classi.

- **RetrivalEngine**: È il motore principale del sistema di retrieval. Coordina la generazione della descrizione testuale del codice, la sua conversione in embedding e l'invocazione del servizio remoto per trovare i contratti simili.

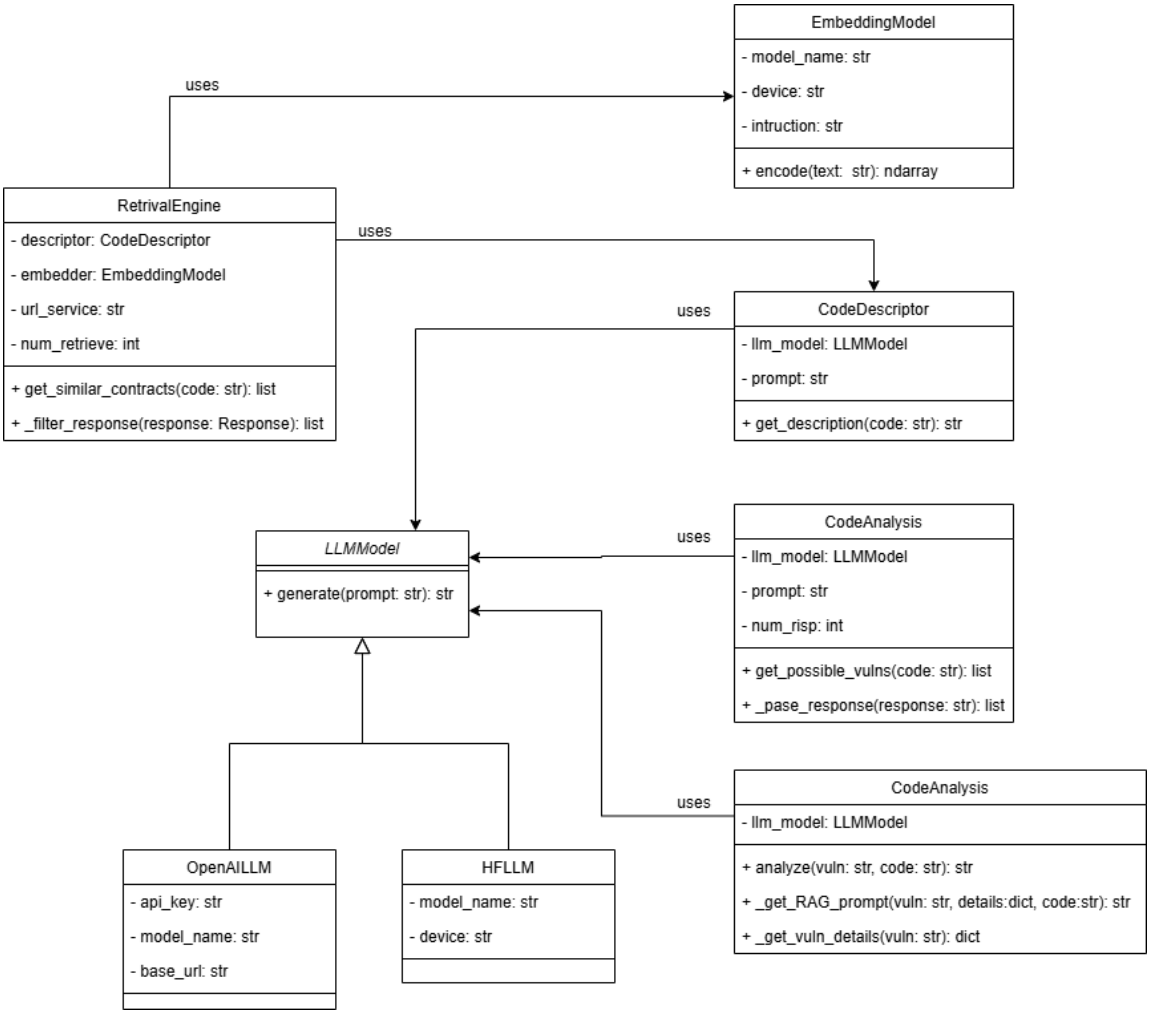

Figure 4: Diagramma delle classi

- **CodeDescriptor**: Trasforma il codice in una descrizione in linguaggio naturale tramite un modello LLM.
- **EmbeddingModel**: Converte un testo in un embedding numerico usando SentenceTransformer.
- **LLMModel** (Classe astratta): Definisce l'interfaccia comune per i modelli LLM. Sottoclassi:
  - **OpenAILLM**: usa API OpenAI (es. DeepSeek).
  - **HFLLM**: usa modelli HuggingFace (es. LLaMA).
- **CodeAnalysis**: Analizza lo smartcontract tramite un LLM per trovare le possibili vulnerabilità
- **VulnAnalysis**: Analizza lo smartcontract su vulnerabilità singole con prompt RAG (arricchiti con informazioni di contesto)

2 Attività di Manutenzione

Nel seguente capitolo verranno descritte le attività di manutenzione svolte a partire dalle Change Request proposte per il tool di analisi delle vulnerabilità LLM-SmartContractScanner. L'analisi riguarderà i benefici, i costi, i rischi e l'impatto delle modifiche da apportare al sistema. In questo contesto, la fase di **Impact Analysis** è fondamentale per determinare quali componenti del sistema attuale potrebbero essere coinvolti dalle modifiche.

Uno strumento chiave per svolgere questa valutazione è il **Traceability Graph**, che consente di tracciare le dipendenze orizzontali tra elementi appartenenti a diversi livelli di astrazione — dai requisiti al design, al codice e ai test — facilitando una visione complessiva dell'impatto delle modifiche.

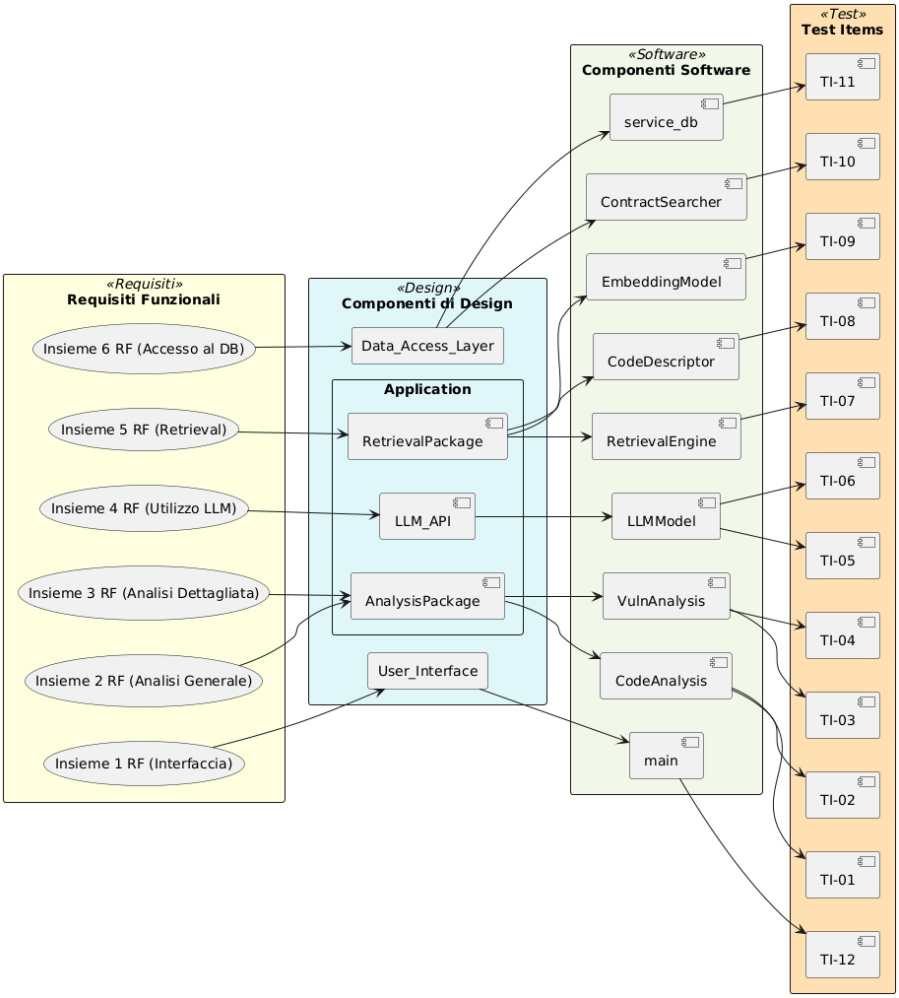

Figure 5: Traceability Graph

2.1 Change Request 1

La prima fase consiste nella comprensione dettagliata della richiesta di modifica, al fine di chiarire l'intento e gli impatti derivanti dall'aggiunta o modifica di requisiti nel sistema attuale.

Campo	Valore
ID	CR_01_CLI
Descrizione	Implementazione di un'interfaccia a riga di comando (CLI) che consenta di utilizzare il sistema in modalità interattiva e che possa essere integrata in pipeline CI/CD. La CLI dovrà offrire opzioni per personalizzare i parametri dell'analisi: • Numero di vulnerabilità da considerare nella fase di *Code Analysis*; • Numero di contratti simili da recuperare nella fase di *Retrieval*; • Possibilità di specificare un modello LLM arbitrario; Per garantire portabilità e semplicità d'uso, si richiede inoltre la creazione di una Docker image e l'orchestrazione dei servizi tramite Docker Compose (CLI, API server, DB vettoriale).
Tipo di manutenzione	Adaptive
Componente	Interfaccia utente (CLI)
Prodotto	LLM-SmartContractScanner
Priorità	Alta

2.1.1 Impact Analysis

L'analisi dell'impatto evidenzia la necessità di introdurre una nuova modalità di utilizzo del sistema, tramite interfaccia a riga di comando con opzioni configurabili da terminale. Inoltre, si rende necessaria la predisposizione all'esecuzione containerizzata dell'intero sistema.

Poiché i moduli di analisi e retrieval sono già progettati per accettare input parametrizzati, le modifiche principali impattano esclusivamente i requisiti dell'*interfaccia* nel seguente modo:

Interfaccia

- **RF1.1** – Il sistema deve essere avviabile tramite CLI da terminale;
- **RF1.2** – La CLI deve accettare parametri configurabili per la fase di analisi:
  - Numero di vulnerabilità da considerare nella fase di Code Analysis;
  - Numero di contratti simili da recuperare nella fase di Retrieval;

  - Specifica di configurazione di un nuovo LLM;

- **RF1.3** - I parametri devono essere salvati in file di configurazione

Inoltre vengono aggiunti nuovi requisiti relativi alla containerizzazione:

Containerizzazione

- **RF7.1** – Il sistema deve essere eseguibile in ambiente Docker;
- **RF7.2** – Il sistema deve essere orchestrabile tramite Docker Compose.

Osservando il grafo di tracciabilità (5) è possibile dedurre gli ulteriori elementi coinvolti. A livello di design è impattato il componente *User Interface*, nel codice il modulo `main`, e nei test, il *System Test*.

Start Impact Set (SIS)

Definiamo come SIS il sottoinsieme di componenti software direttamente toccati dalla modifica. In questo caso, il SIS comprende:

$$\{\ \texttt{main}\ \}$$

Candidate Impact Set (CIS)

Per individuare ulteriori componenti potenzialmente affetti, analizziamo le dipendenze del modulo `main` tramite il suo call graph, come mostrato in Figura 6.

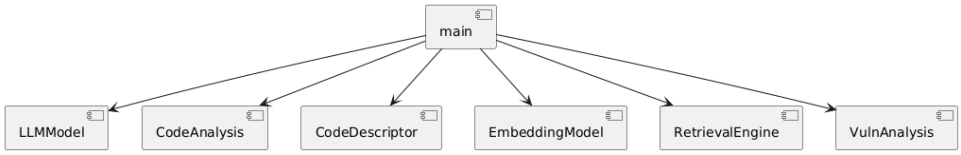

Figure 6: Dipendenze del modulo `main`

Il modulo `main`, responsabile dell'orchestrazione della pipeline, coordina le chiamate ai moduli di analisi e retrieval. Tuttavia, la modifica richiesta non altera le interfacce dei moduli invocati, ma esclusivamente la modalità di avvio della pipeline. Di conseguenza, il **CIS** resta invariato e comprende solo:

$$\{\ \texttt{main}\ \}$$

2.1.2 Valutazione Costi, Benefici e Rischi

Una volta completata l'analisi dell'impatto, è fondamentale valutare i costi, benefici e rischi associati all'implementazione della **Change Request 1**. Questa valutazione consente di comprendere la fattibilità della modifica, il valore aggiunto per il sistema e le eventuali criticità da tenere sotto controllo. Le seguenti tabelle sintetizzano tale analisi, includendo una stima qualitativa dell'importanza o dell'intensità di ciascun elemento.

Voce di Costo	Descrizione	Valutazione
Sviluppo CLI	Progettazione e implementazione dell'interfaccia a riga di comando.	Media
Integrazione parametri CLI	Adeguamento del modulo `main` per propagare i parametri agli altri moduli.	Bassa
Scrittura Dockerfile	Creazione di un'immagine container portabile per i servizi del sistema.	Bassa
Setup Docker Compose	Configurazione dell'ambiente multi-container per orchestrazione.	Media
Testing	Validazione della CLI e verifica dell'integrazione in ambienti containerizzati.	Alta

Table 1: Analisi dei costi della Change Request

Beneficio	Descrizione	Valutazione
Flessibilità d'uso	Permette l'esecuzione parametrica da terminale.	Alta
Portabilità	Il sistema può essere eseguito su qualsiasi host tramite Docker.	Alta
Manutenibilità	La separazione dei servizi semplifica aggiornamenti.	Media
Distribuzione semplificata	Le immagini possono essere pubblicate e scaricate facilmente.	Alta

Table 2: Analisi dei benefici della Change Request

Rischio	Descrizione	Valutazione
Errori nella CLI	Possibilità di input errati non gestiti correttamente.	Media
Compatibilità ambienti	Differenze tra ambienti locali e container possono causare malfunzionamenti.	Alta
Overhead iniziale	Tempo necessario per apprendere e configurare Docker.	Bassa
Immagini Docker troppo pesanti	L'uso di librerie complesse (Torch, SentenceTransformers) può aumentare significativamente le dimensioni delle immagini, rallentando il deploy e la distribuzione.	Alta

Table 3: Analisi dei rischi della Change Request

2.1.3 Design

Le modifiche principali hanno riguardato il componente *User Interface*, il quale è stato esteso per gestire diversi tipi di comandi attraverso l'interfaccia a riga di comando. I comandi attualmente supportati includono:

- **run**: avvia l'analisi utilizzando le opzioni specificate come argomenti;
- **set-model**: consente di impostare un nuovo modello LLM da utilizzare per l'analisi;
- **model-list**: visualizza i modelli LLM attualmente salvati nel sistema.

Per supportare queste funzionalità, sono stati adottati diversi pattern architetturali, descritti di seguito:

Command Pattern

Il *Command Pattern* consente di gestire i comandi dell'utente in modo modulare, mappando ciascun comando CLI a una classe dedicata che implementa l'interfaccia comune `Command`. Ogni classe implementa il metodo `execute()`, in cui viene definita la logica specifica del comando, come ad esempio l'invocazione dei componenti dell'*Application Layer*. In questo modo, l'interfaccia utente rimane separata dalla logica di business. Il componente `CLIInvoker` è responsabile del parsing dell'input dell'utente e della creazione del comando corrispondente. Questo approccio favorisce l'estendibilità del sistema semplificando l'introduzione di nuovi comandi.

Factory Pattern

Il *Factory Pattern* è stato adottato per la costruzione dinamica dei modelli LLM, incapsulando la logica di selezione, inizializzazione e validazione all'interno della classe `LLMFactory`. Questa soluzione permette di supportare agevolmente diversi backend, tra cui:

- modelli OpenAI accessibili tramite API key;
- modelli Hugging Face, sia locali che remoti.

Configuration Layer

È stato introdotto un *Configuration Layer* che si interpone tra l'interfaccia utente e l'*Application Layer*, con l'obiettivo di gestire in maniera centralizzata la configurazione dei componenti dell'analisi. Il componente `AppContext` ha il compito di leggere le configurazioni, sia dai comandi CLI sia tramite il `ConfigManager`, che gestisce in modo persistente i modelli e le opzioni disponibili nel sistema.

La Figura 7 mostra il nuovo design architetturale del sistema:

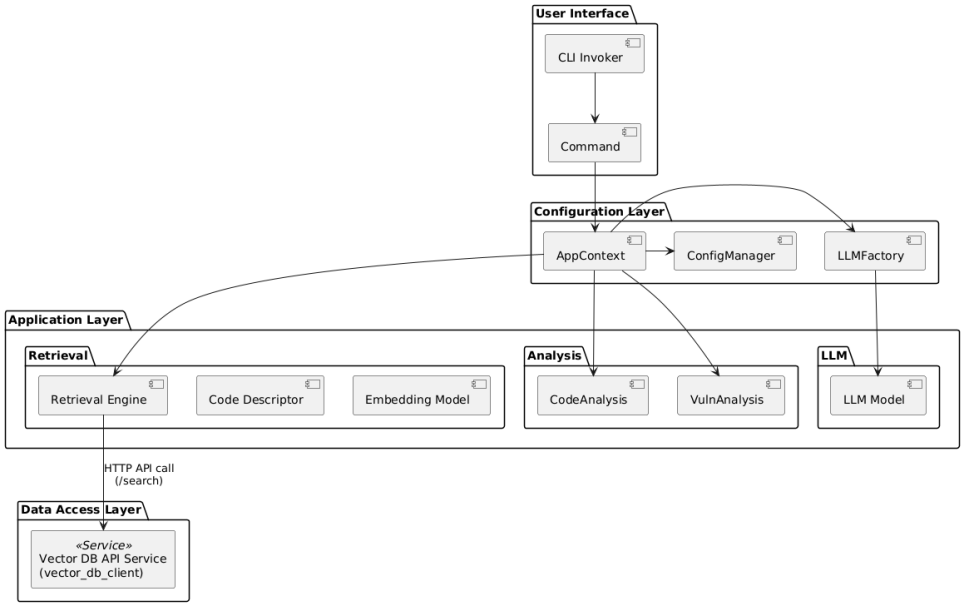

Figure 7: Nuovo design del sistema

Le Figure 8 e 9 rappresentano i diagrammi di sequenza relativi ai comandi `run` e `set-model`, descrivendone il flusso di interazioni tra i componenti coinvolti.

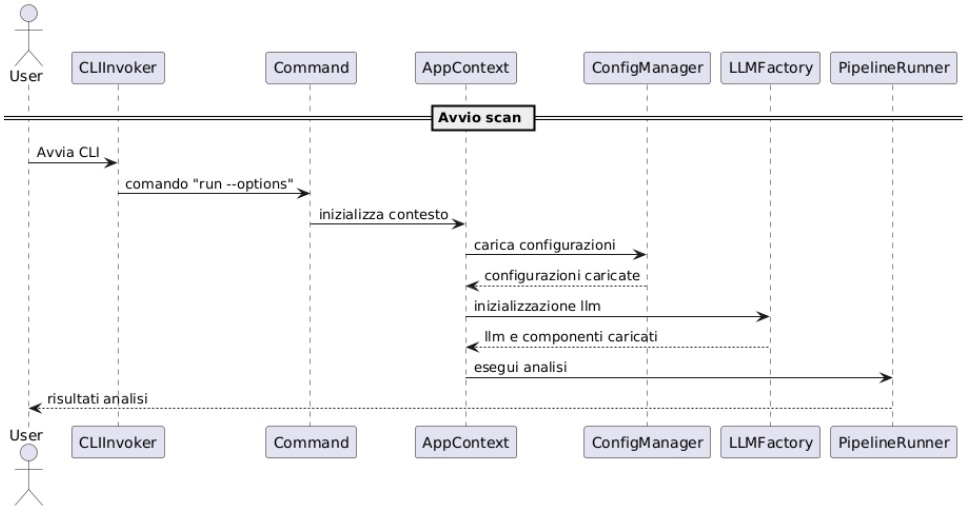

Figure 8: Sequence Diagram per il comando `run`

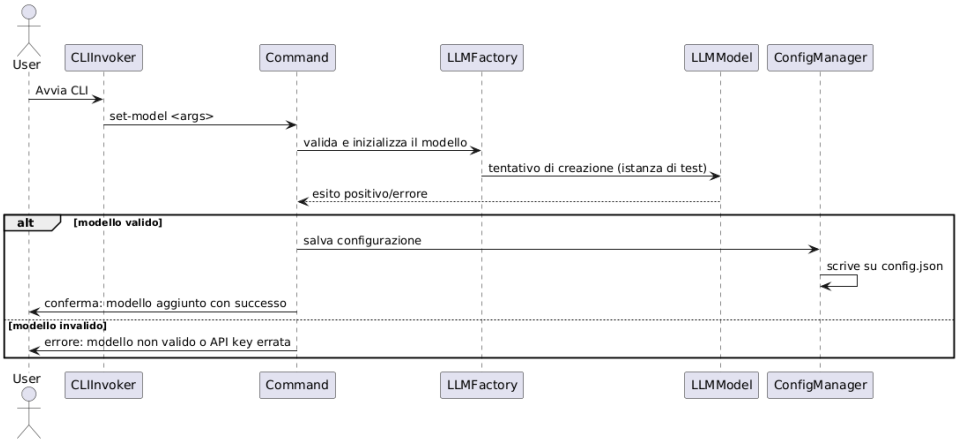

Figure 9: Sequence Diagram per il comando `set-model`

Infine, le Figure 10 e 11 mostrano i diagrammi delle classi dei nuovi componenti relativi alla *User Interface* e al *Configuration Layer*.

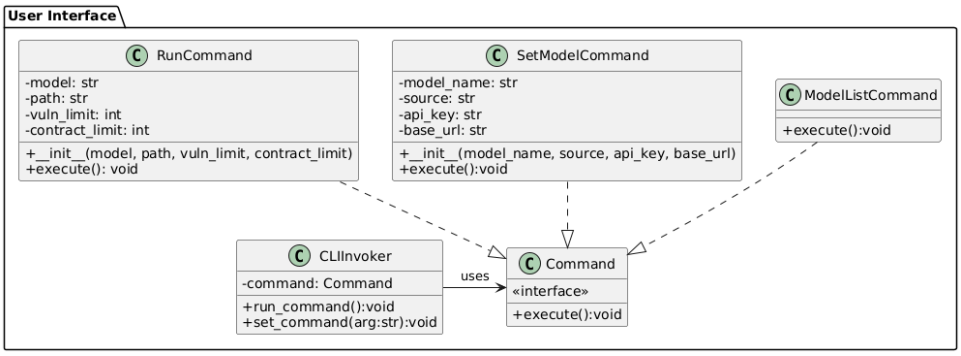

Figure 10: Class Diagram della *User Interface*

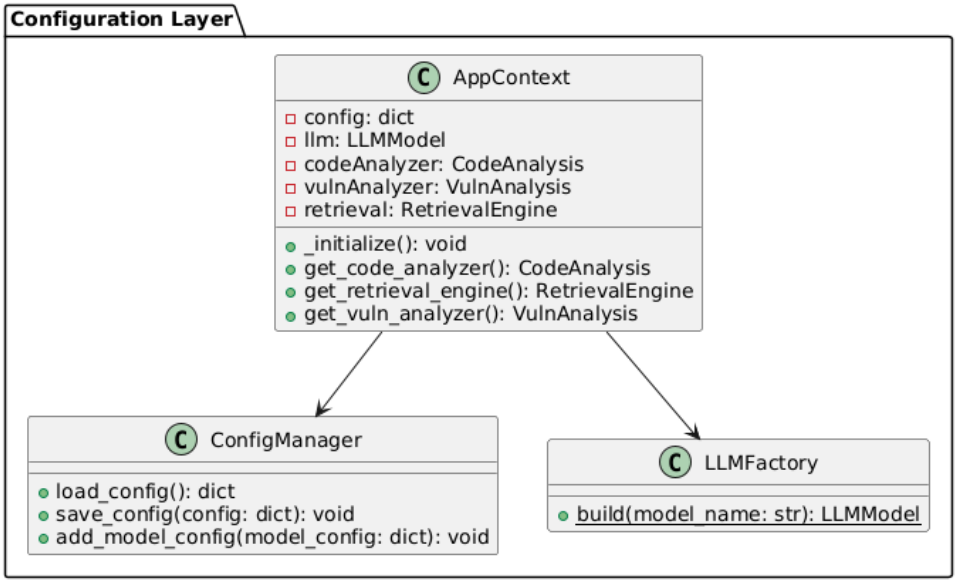

Figure 11: Class Diagram del *Configuration Layer*

Gestione delle Immagini Docker

Ogni componente del sistema è costruito a partire da un'immagine Docker preconfigurata. Ciascuna immagine è definita tramite un apposito `Dockerfile`, che specifica in modo esplicito tutte le dipendenze software necessarie per il corretto funzionamento del servizio. Questo approccio garantisce che ogni immagine sia autosufficiente, riproducibile e facilmente distribuibile.

In particolare:

- Le dipendenze (come librerie Python, modelli di elaborazione semantica, ecc...) sono installate durante la fase di `build` dell'immagine.
- L'uso di immagini minimali (es. python:3.11-slim) consente di ridurre la dimensione complessiva.

Ad ogni modifica significativa al codice sorgente o alle dipendenze di un componente, viene avviata una nuova build dell'immagine Docker corrispondente. Una volta verificata e testata, la nuova immagine viene pubblicata su Docker Hub, taggata come `latest` per definire l'ultima release stabile.

Orchestrazione

L'architettura del sistema è stata definita attraverso un file `docker-compose.yml`, che orchestra tre principali componenti containerizzati: la base dati (Qdrant), il server API e il tool CLI per l'interazione dell'utente.

- **qdrant**: è il database vettoriale utilizzato per il recupero degli smartcontract vulnerabili. È avviato tramite un'immagine preconfigurata su Docker Hub (`niktor99/sc-vector-db`) e mantiene la persistenza dei dati attraverso il volume `qdrant_data`.
- **api_server**: è il responsabile della comunicazione tra il database Qdrant e i moduli di analisi. L'immagine viene ottenuta da Docker Hub (`niktor99/api-server:latest`). Il servizio dipende da `qdrant`, assicurando l'ordine corretto di avvio, ed espone la porta `8000`.
- **cli_tool**: contiene l'interfaccia utente a riga di comando e la logica del tool, eseguita tramite l'immagine prebuildata `niktor99/cli-tool:latest`. La configurazione prevede l'uso delle opzioni `stdin_open: true` e `tty: true` per supportare l'interazione da terminale. L'`entrypoint` avvia lo script `main.py` che funge da entrypoint.

Questa architettura presenta i seguenti vantaggi:

- **Isolamento**: ogni componente può essere sviluppato, testato e scalato indipendentemente.
- **Portabilità**: l'intero sistema può essere eseguito su qualsiasi host con Docker e Docker Compose.

2.1.4 Implementazione

L'implementazione si è focalizzata sull'aggiunta di un'interfaccia a riga di comando (CLI) modulare, mantenendo invariata la logica di business esistente del sistema.

Il componente `CLIInvoker` funge da invocatore centrale, che riceve i comandi da riga di comando, li interpreta tramite il modulo `argparse` di Python e li associa alla rispettiva classe `Command`. L'invocazione di comandi come `run`, `set-model` o `model-list` avviene dunque in maniera dinamica. Il comando `RunCommand`, ad esempio, all'interno del suo metodo `execute()` inizializza il contesto applicativo (`AppContext`) che a sua volta prepara tutti i componenti del sistema necessari per eseguire l'analisi. La logica di business sottostante non è stata modificata: i moduli di analisi (es. `CodeAnalysis`, `VulnAnalysis`, ecc...) sono stati semplicemente collegati attraverso il contesto.