Contents

Manutenzione ed Evoluzione del sistema LLM-SmartContractScanner

Nicola Tortora, Gaspare Galasso

July 24, 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).

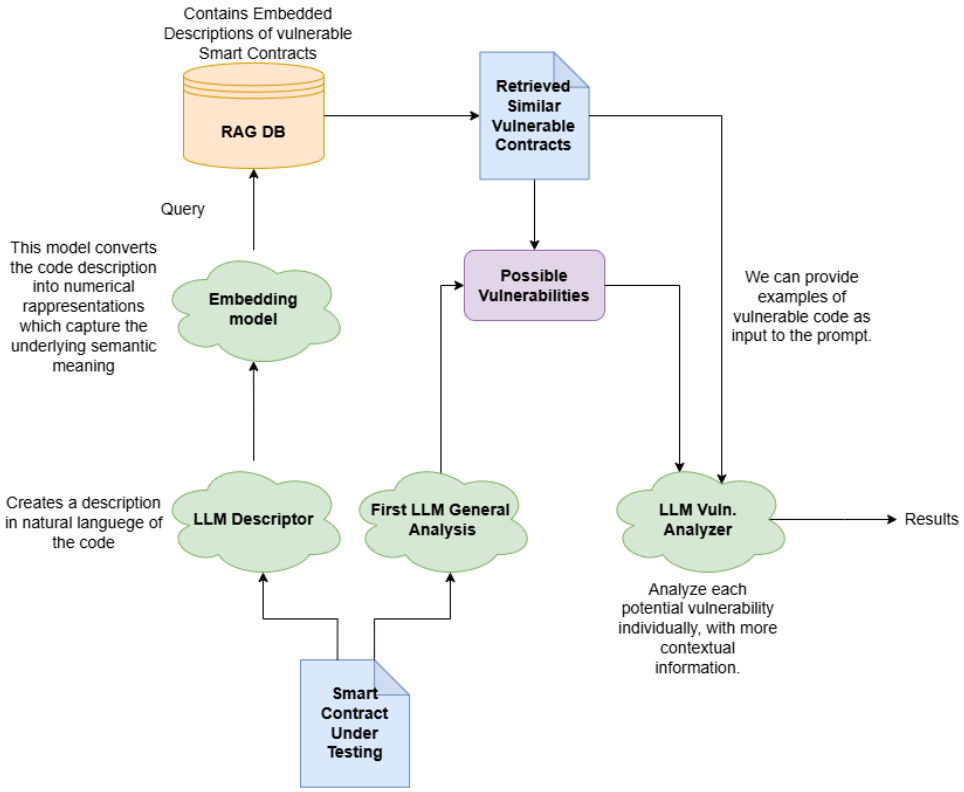

Figure 1: Architettura generale del sistema

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.

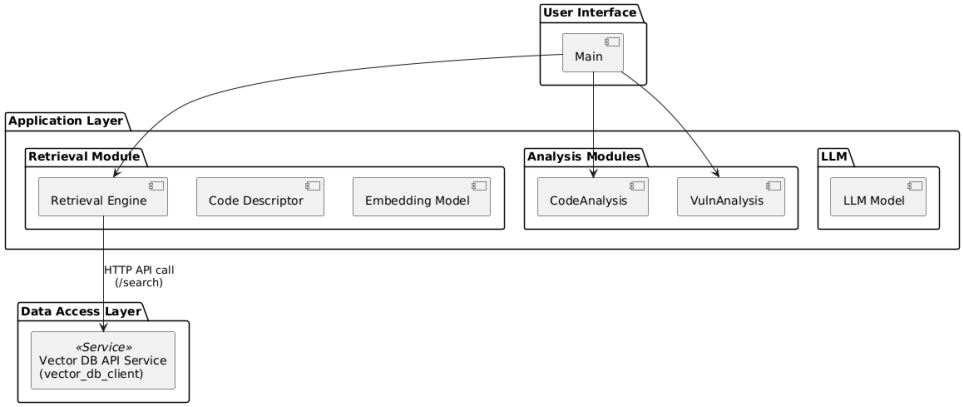

Figure 2: Architettura del sistema

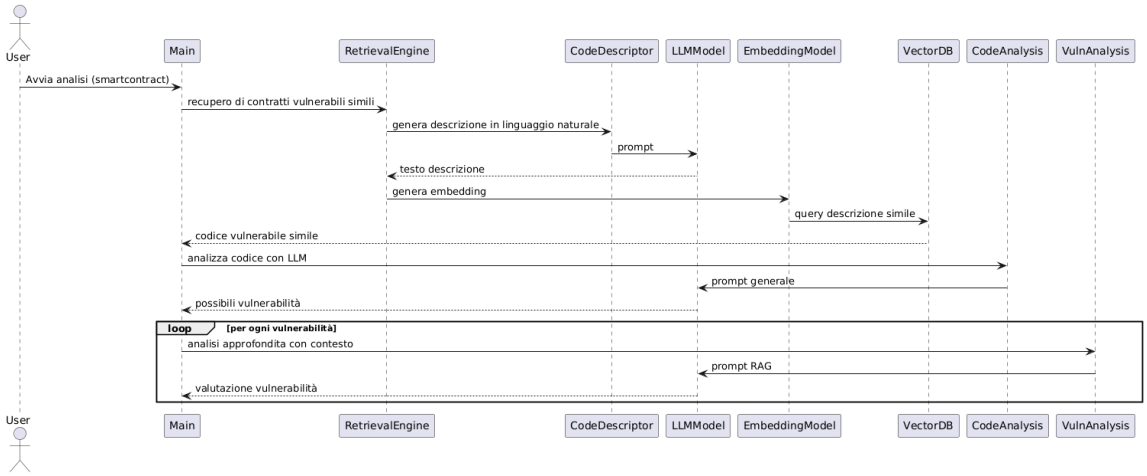

Figure 3: Diagramma di sequenza delle interazioni tra i moduli

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.

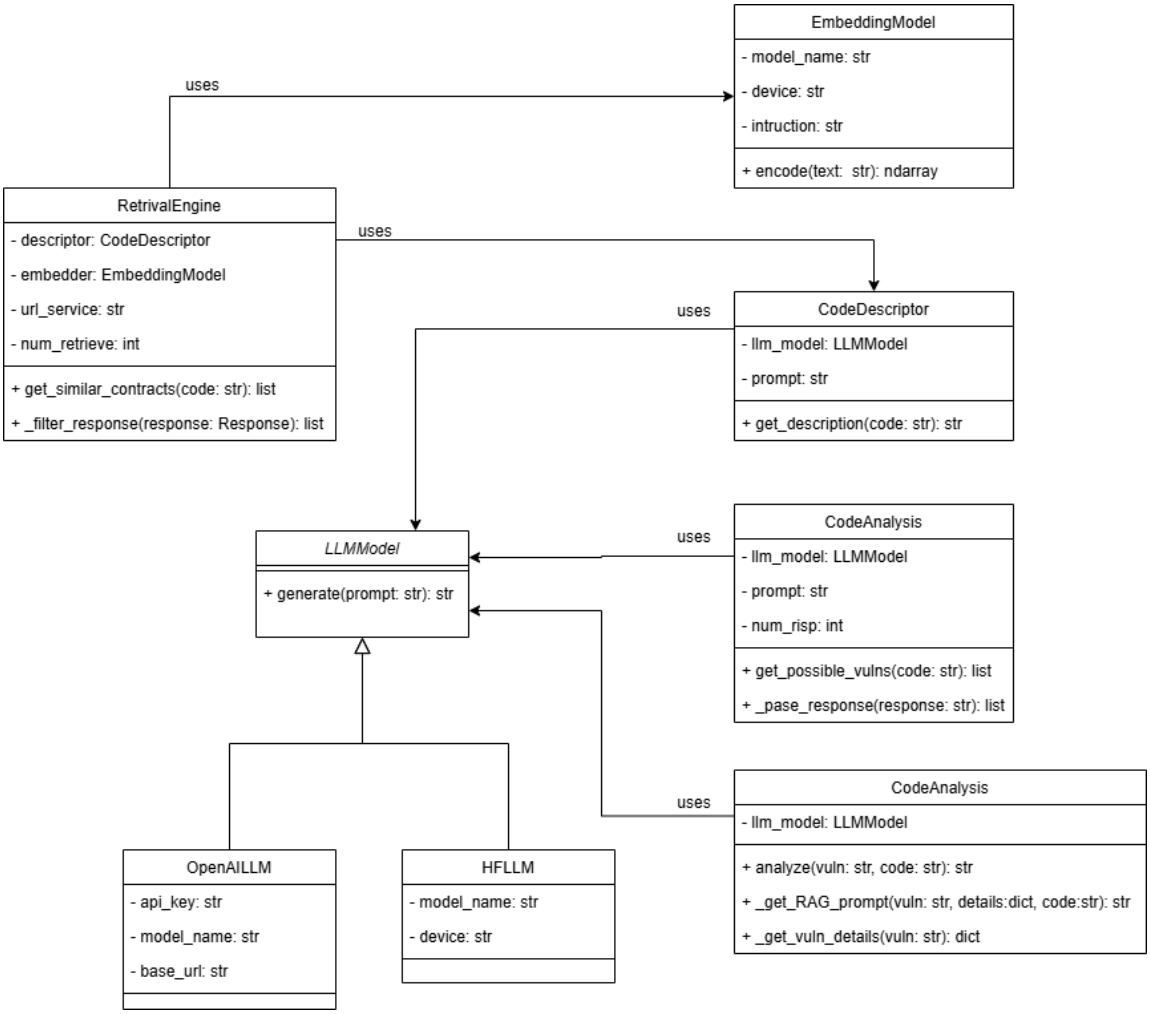

Figure 4: Diagramma delle classi

- **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.

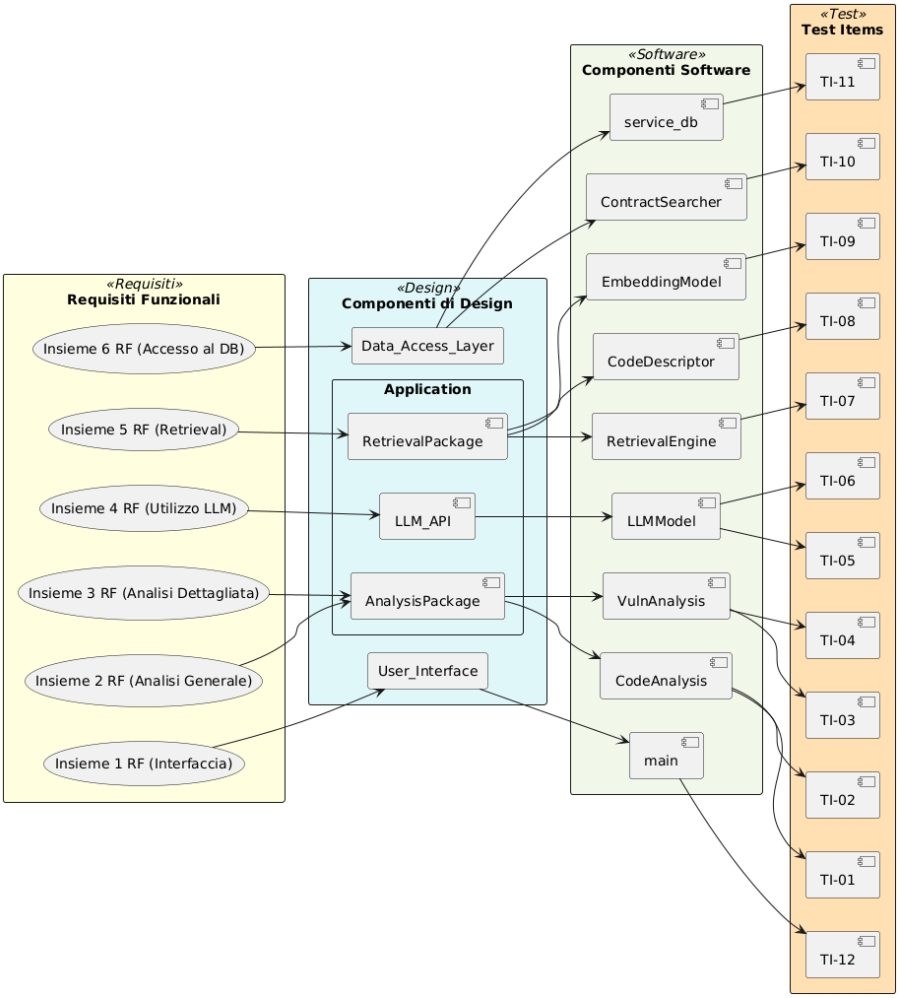

Figure 5: Traceability Graph

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.

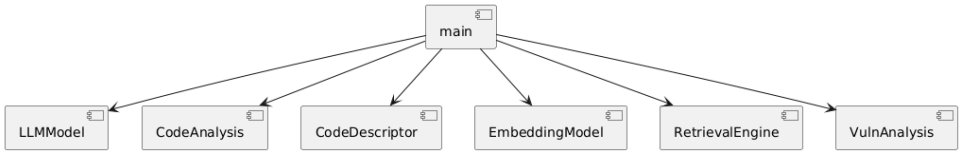

Figure 6: Dipendenze del modulo `main`

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.

Le Figure 7 e 8 rappresentano i diagrammi di sequenza relativi ai comandi `run` e `set-model`, descrivendone il flusso di interazioni tra i componenti coinvolti.

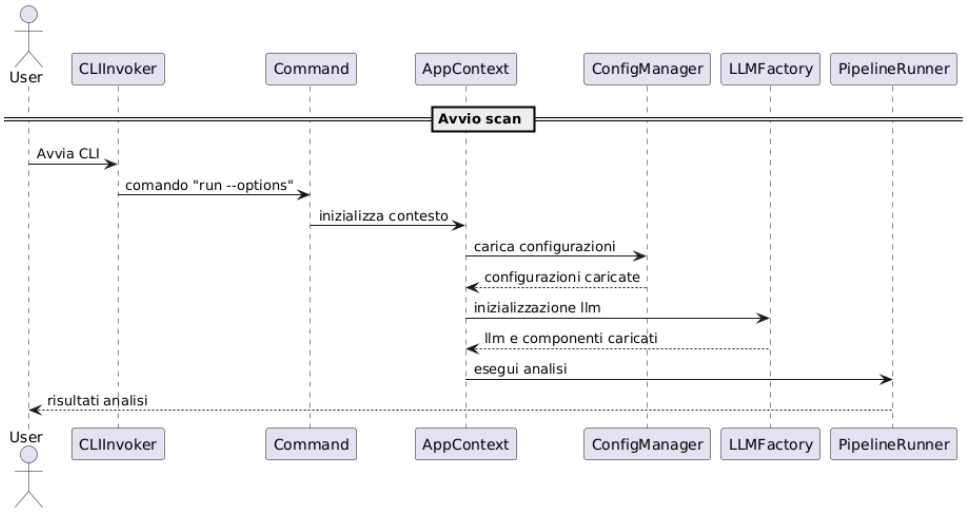

Figure 7: Sequence Diagram per il comando `run`

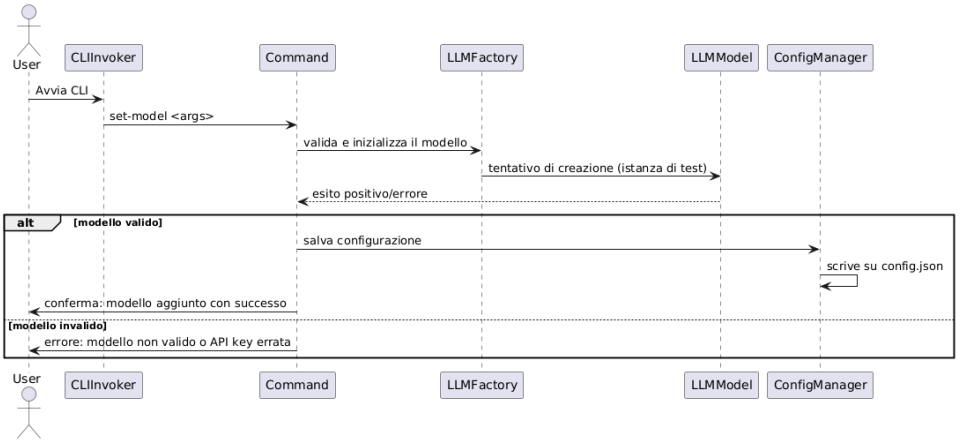

Figure 8: Sequence Diagram per il comando `set-model`

Infine, le Figure 9 e 10 mostrano i diagrammi delle classi dei nuovi componenti relativi alla *User Interface* e al *Configuration Layer*.

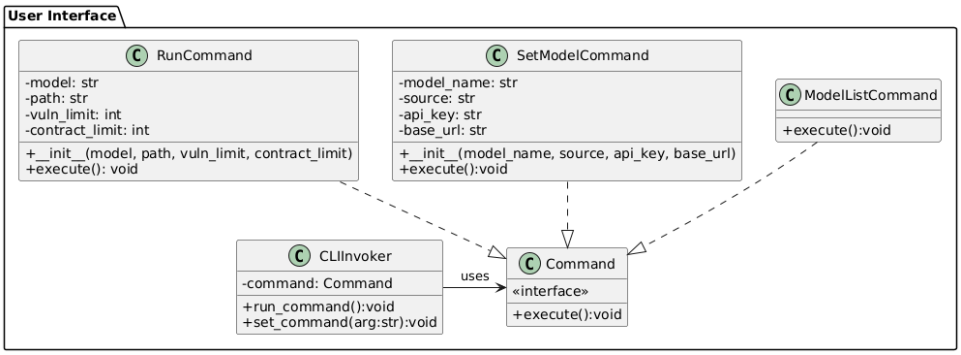

Figure 9: Class Diagram della *User Interface*

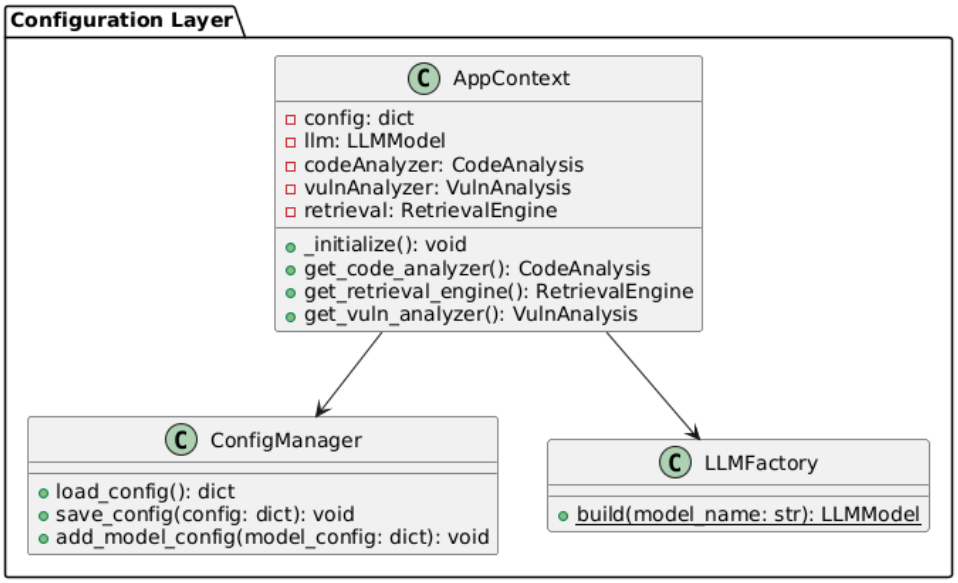

Figure 10: Class Diagram del *Configuration Layer*

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.

La Figura 11 mostra il nuovo design architetturale del sistema:

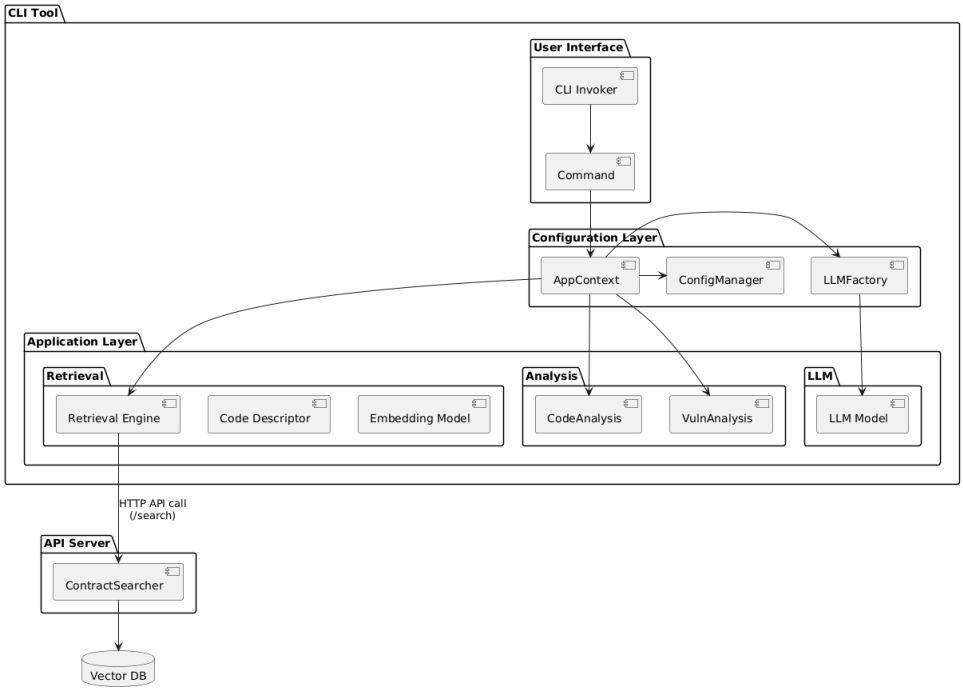

Figure 11: Nuovo design del sistema

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 ed eseguiti attraverso lo script `pipeline.py`, che esegue i componenti in successione. Quest'ultimo è stato introdotto per sollevare il main dal compito di orchestratore della pipeline.

Il nuovo design prevede l'introduzione di due entrypoint distinti:

- **Shell interattiva**: avvia un REPL dove l'utente può inserire comandi sequenziali (run, set-model, model-list).
- **Batch Mode**: consente l'esecuzione diretta di un comando specifico via CLI (e.g. `python main.py run ...`).

Dopo l'implementazione possiamo definire Actual Impact Set (AIS), che comprende i componenti effettivamente impattati dalla modifica e valutare il recall e la precision dell'impact analysis. L'unico componente imapttato è stato il modulo `main` come pianificato in precedenza.

$$\texttt{CIS: \{ main \}}$$

$$\texttt{AIS: \{ main \}}$$

$$\text{Recall} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{AIS}|} = \frac{1}{1} = 1$$

$$\text{Precision} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{CIS}|} = \frac{1}{1} = 1$$

2.1.5 Test

Le strategie di test sono descritte nel documento Test Plan-Maintenance, mentre i risultati delle esecuzione dei test sono presenti nel documento Test Report-Maintenance

2.2 Change Request 2

Analizziamo la richiesta di modifica **CR_02_Report** al fine di estrapolarne i nuovi requisiti funzionali e valutarne l'impatto sull'architettura esistente.

Campo	Valore
ID	CR_02_Report
Descrizione	Introduzione di un componente per la generazione di report dettagliati, al fine di offrire una visione più chiara e leggibile dei risultati prodotti dall'analisi.
Tipo di manutenzione	Perfective
Componente coinvolto	Report Generator
Prodotto	LLM-SmartContractScanner
Priorità	Alta

Table 4: Change Request CR_02

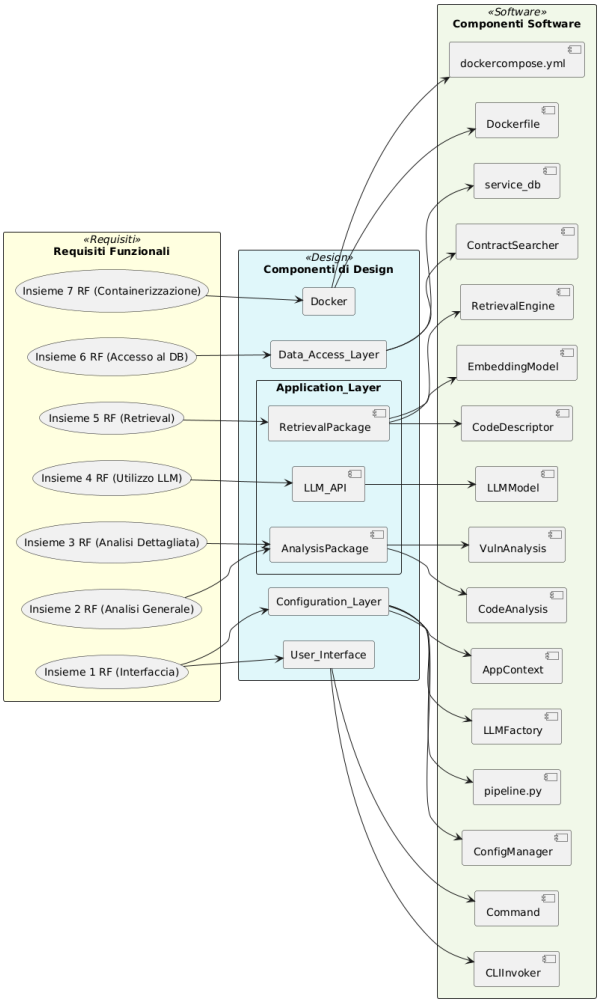

Figure 12: Traceability Graph aggiornato

2.2.1 Impact Analysis

L'analisi dell'impatto evidenzia la necessità di introdurre un nuovo componente software, il `ReportGenerator`, responsabile della produzione di un file `.html` contenente i risultati strutturati dell'analisi effettuata dal sistema. Questo componente dovrà essere attivato dal comando `run`, a cui verrà aggiunta una nuova opzione per la generazione del report. Il cambiamento comporta un'estensione dei requisiti funzionali relativi all'interfaccia del sistema:

Interfaccia

- **RF1.4** – Il sistema deve generare un file HTML contenente un report dettagliato delle vulnerabilità rilevate durante l'analisi, a seguito dell'esecuzione del comando `run`. Il nuovo componente `ReportGenerator` dovrà:
  - accettare in input i risultati dell'analisi;
  - eseguire operazioni di parsing e formattazione;
  - salvare il report in un file specificato tramite l'opzione `--out`.

Inoltre, è necessario garantire la persistenza e accessibilità dei report generati, specialmente in ambienti containerizzati. A tal fine si introduce il seguente requisito relativo alla containerizzazione:

Containerizzazione

- **RF7.3** – Il sistema deve permettere il salvataggio dei report in una directory interna al container Docker e consentire il montaggio di tale directory in una cartella condivisa esterna, al fine di consentire l'accesso ai report anche al di fuori del contenitore.

Start Impact Set (SIS)

Dal grafo di tracciabilità in Figure 12 si evince che la modifica ai requisiti impatta:

- il design dell'interfaccia utente (CLI);
- i componenti software `CLIInvoker`, che dovrà gestire la nuova opzione `--out`, e `RunCommand` che dovrà invocare il `ReportGenerator`;
- i file di configurazione Docker, in particolare `docker-compose.yml`, per la gestione delle directory di output condivise;
- i relativi test associati alle componenti modificate.

In base alla conoscenza architetturale del progetto, identifichiamo i seguenti elementi come direttamente impattati dalla modifica:

{ `CLIInvoker, RunCommand, docker-compose.yml` }

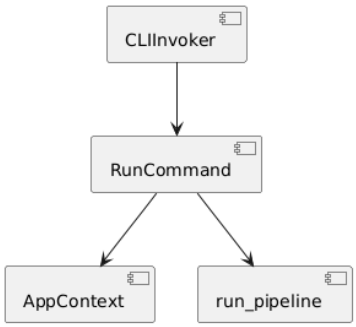

Figure 13: Call graph di CLIInvoker e RunCommand

Candidate Impact Set (CIS)

Per individuare ulteriori componenti impattatti, analizziamo il call graph dei componenti del SIS (Figura 13)

Possiamo notare che RunCommand chiama lo script run_pipeline per eseguire la logica principale del sistema. Quest'ultimo attualmente non restituisce alcun valore a RunCommand, ma stampa semplicemente i risultati su stdout.

Tuttavia, per permettere la generazione del report in formato HTML, è necessario che run_pipeline venga modificato affinché ritorni i risultati dell'analisi in una forma strutturata (es. dizionario o oggetto), così da poter essere passati al nuovo componente ReportGenerator. Di conseguenza, anche run_pipeline entra nel Candidate Impact Set.

Anche AppContext deve essere incluso in quanto potrebbe essere necessario configurare ReportGenerator come avviene con gli altri componenti.

{ `CLIInvoker, RunCommand, run_pipeline, AppContext, docker-compose.yml` }

2.2.2 Valutazione Costi, Benefici e Rischi

Voce di Costo	Descrizione	Valutazione
Sviluppo del componente	Progettazione e implementazione del nuovo modulo `ReportGenerator`.	Basso
Modifica pipeline	Adattamento dello script `run_pipeline` per restituire output strutturato.	Medio
Aggiornamento CLI	Estensione dei comandi CLI per supportare l'opzione `--out`.	Basso
Adattamento test	Creazione o aggiornamento di test di unità, integrazione e sistema.	Medio-alto
Manutenzione Docker	Modifica ai file `Dockerfile` e `docker-compose.yml` per gestire output HTML.	Basso

Table 5: Analisi dei costi della Change Request

Beneficio	Descrizione	Valutazione
Maggiore leggibilità	Output in formato HTML facilmente consultabile anche da utenti non tecnici.	Alto
Tracciabilità dei risultati	Possibilità di archiviare e confrontare più report nel tempo.	Medio-alto

Table 6: Analisi dei benefici della Change Request

Rischio	Descrizione	Valutazione
Complessità del parsing output	Errori nella generazione del report a causa di output non ben strutturato, dovuto alla variabilità degli output dei modelli LLM.	Medio-alto
Errori in ambienti Docker	Problemi con permessi di scrittura/montaggio volumi in ambienti containerizzati.	Medio

Table 7: Analisi dei rischi della Change Request

2.2.3 Design

Le modifiche introdotte richiedono l'integrazione di un nuovo componente, denominato `ReportGenerator`, responsabile della generazione di report a partire dai risultati dell'analisi.

Per favorire la riusabilità e l'estendibilità del codice, è stato adottato il design pattern **Template Method**. Questo pattern prevede la definizione in una classe astratta che definisce la struttura generale di un algoritmo dividendolo in step e demandando alle sottoclassi l'implementazione dei singoli passaggi che possono variare.

Nel caso della generazione dei report, la classe base definisce i seguenti step principali dell'algoritmo:

1. **Preparazione dei dati (prepare_data)**: parsing dell'output testuale per estrarre informazioni strutturate.
2. **Formattazione del report (render)**: trasformazione dei dati strutturati in un formato specifico (nel caso attuale, HTML).
3. **Scrittura su file (write)**: salvataggio del report su file.

Questa struttura consente di evitare duplicazione di codice e di semplificare l'aggiunta di nuovi formati di output. Ad esempio, per supportare anche l'esportazione in `JSON` o `YAML`, sarà sufficiente estendere la classe base e sovrascrivere esclusivamente il passo di formattazione (render), mantenendo invariati i restanti.

Le Figure 14 15 mostrano rispettivamnete il design e il class diagram relativi alla modifica

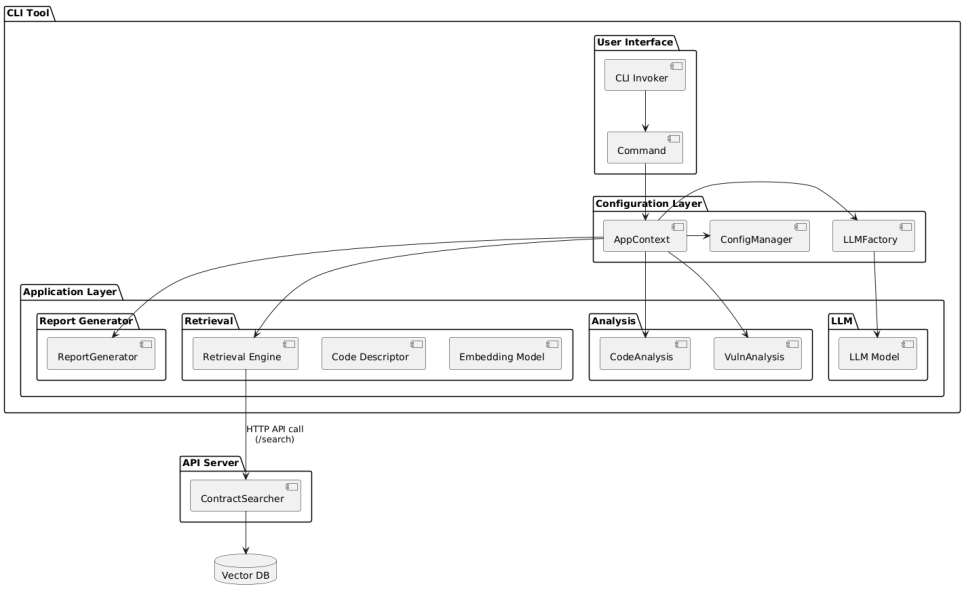

Figure 14: Nuovo Design

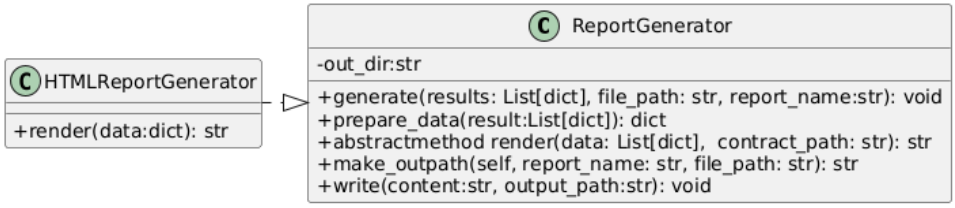

Figure 15: Class Diagram del Report Generator

2.2.4 Implementazione

L'implementazione si è concentrata principalmente sullo sviluppo dell'algoritmo di generazione dei report. Per permettere al componente `ReportGenerator` di elaborare i dati dell'analisi, lo script `run_pipeline` è stato modificato per restituire una lista strutturata di dizionari, ciascuno contenente il nome della vulnerabilità e il testo dell'analisi prodotta dal modello.

Il metodo principale, `generate()`, segue una sequenza di passi predefiniti:

- `prepare_data()`: riceve in input i risultati strutturati e applica operazioni di parsing tramite espressioni regolari, estraendo informazioni rilevanti dal testo dell'analisi, quali: descrizione della vulnerabilità, snippet di codice vulnerabile e codice sicuro suggerito.

- `render()`: è un metodo astratto, la cui implementazione concreta dipende dal tipo di report da generare. Nella sottoclasse `HTMLReportGenerator`, il metodo `render` utilizza un template HTML popolato con i dati elaborati da `prepare_data()`. Se non vengono rilevate vulnerabilità, viene utilizzato un template alternativo che riporta l'assenza di problemi.

- `make_outpath()`: si occupa di generare un nome univoco e coerente per il file di report, evitando sovrascritture o errori in caso di nome mancante o già esistente.

- `write()`: si limita a scrivere su file il contenuto generato, nel percorso indicato.

L'istanza di `ReportGenerator` viene configurata all'interno della classe `AppContext`, che imposta anche la directory di output a partire dalle impostazioni definite nel file di configurazione. Il metodo `generate()` viene infine invocato dal comando `RunCommand`.

Nel file `docker-compose.yml`, per rendere accessibile all'utente il report generato, viene montata una directory locale nel container:

```
– ./output_report:/app/output_report
```

A seguito dell'implementazione, è stato possibile definire l'**Actual Impact Set (AIS)**, cioè l'insieme dei componenti effettivamente modificati dalla change request. Oltre ai componenti già identificati nel **Candidate Impact Set (CIS)**, è risultato impattato anche il modulo `ConfigManager`, in seguito all'introduzione di un nuovo campo di configurazione.

```
CIS = {CLIInvoker, RunCommand, run_pipeline, AppContext, docker-compose.yml}

AIS = {CLIInvoker, RunCommand, run_pipeline, AppContext, ConfigManager, docker-compose
```

$$\text{Recall} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{AIS}|} = \frac{5}{6} \approx 0.83$$

$$\text{Precision} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{CIS}|} = \frac{5}{5} = 1$$

2.2.5 Test

Le strategie di test sono descritte nel documento Test Plan-Maintenance, mentre i risultati delle esecuzione dei test sono presenti nel documento Test Report-Maintenance

2.3 Change Request 3

Campo	Valore
ID	CR_03_Redesign
Descrizione	La modifica prevede il ripensamento dell'architettura del sistema tramite la separazione delle responsabilità tra frontend (CLI tool) e backend (server API). La logica di business, attualmente inglobata nel CLI, viene spostata sul server, rendendo il sistema più scalabile. Il CLI fungerà da semplice interfaccia remota, interagendo con il backend tramite chiamate HTTP REST.
Tipo di manutenzione	Perfective
Componente coinvolto	CLI Tool, API Server
Prodotto	LLM-SmartContractScanner
Priorità	Alta

Table 8: Change Request CR_03

2.3.1 Impact Analysis

Per *logica di business* si intende l'intera pipeline di analisi, attualmente integrata direttamente nel tool a riga di comando (CLI). Con questa modifica, tale logica verrà spostata completamente sul server API, che si occuperà di orchestrare l'analisi e restituire i risultati. La pipeline rimarrà invariata nelle sue funzionalità, ma sarà eseguita remotamente.

Il server API, che in precedenza gestiva unicamente l'interazione con il database vettoriale, assumerà ora un ruolo centrale. La CLI fungerà da semplice interfaccia client e, invece di invocare direttamente i componenti interni del sistema, invierà richieste HTTP REST al server. Di conseguenza, i comandi della CLI (`SetModelCommand`, `RunCommand`, `ModelListCommand`) non accederanno più ai moduli di business logic localmente, ma si limiteranno a comunicare con le corrispondenti API esposte.

Questa modifica implica anche un aggiornamento della configurazione dei contenitori Docker, relativi ai servizi cli_tool e api_server.

Start Impact Set (SIS)

{ `SetModelCommand, RunCommand, ModelListCommand,`

`Dockerfile(cli tool), Dockerfile(api server)` }

Candidate Impact Set (CIS)

Valutando le dipendenze tra i componenti, osserviamo che la classe RetrievalEngine effetua una chiama al server api per recuperare i contratti vulnerabili. Dato il cambiamento della logica del sistema riteniamo tale componente interessato alla modifica.

Un componente potenzialmente interessato alla modifica potrebbe essere il file docker-compose.yml, in quanto l'alterazione delle immagini docker dei servizi cli_tool e api_server, potrebbe modificare l'orchestrazione dei container.

{ `SetModelCommand, RunCommand, ModelListCommand, RetrievalEngine,`

`Dockerfile(cli tool), Dockerfile(api server), docker-compose.yml` }

2.3.2 Valutazione Costi, Benefici e Rischi

Voce di costo	Descrizione	Valutazione
Refactoring CLI	Necessario modificare l'implementazione dei comandi per interfacciarsi con il server via HTTP.	Bassa
Estensione Server API	La logica di analisi va integrata sul server, richiedendo nuove route.	Bassa
Testing	Saranno necessarie test suite di integrazione e sistema per garantire la correttezza nella comunicazione CLI - API.	Media
Dockerizzazione	Aggiornamento e gestione dei Dockerfile.	Bassa

Table 9: Costi CR_03_Redesign

Beneficio	Descrizione	Valutazione
Separation of Concerns	Separazione chiara tra interfaccia (CLI) e logica di business (server).	Alta
Estensibilità futura	Maggiore facilità nell'integrare nuove interfacce (es. frontend web) tramite API comuni.	Alta

Table 10: Benefici CR_03_Redesign

Non sono stati rilevati rischi rilevanti.

2.3.3 Design

Per realizzare il redesign architetturale volto a separare chiaramente le responsabilità all'interno del sistema, si adotta un approccio basato su una classica architettura **client-server**.

- Il **server API** assume il ruolo di componente centrale dell'applicazione, ospitando la **business logic**, ovvero l'intera pipeline di analisi.
- Il **CLI** viene trasformato in un'interfaccia client minimale, il cui unico compito è raccogliere gli input dell'utente, inviare le richieste al server tramite API REST e generare il report tramite il risultato ricevuto.

Per l'organizzazione interna del server si è scelto di adottare il pattern architetturale **Model-View-Controller (MVC)**, con l'obiettivo di separare le responsabilità tra le componenti del sistema:

- **Model**: contiene le classi relative alla configurazione dell'ambiente, alla gestione dei modelli e agli oggetti di dominio dell'analisi.

- **View**: è rappresentata dai formati di risposta delle API (principalmente in formato JSON), che vengono restituiti ai client.
- **Controller**: gestisce le richieste HTTP in ingresso, eseguendo la validazione dei dati e delegando la logica applicativa ai service.

Per migliorare ulteriormente la manutenibilità e la testabilità del sistema, è stato introdotto il pattern **Service Layer**, che funge da livello intermedio tra i controller e la logica di dominio. I *service* incapsulano le operazioni principali della logica di business. Questo approccio consente un riutilizzo più semplice dei test preesistenti (in particolare quelli sui vecchi comandi della CLI) e facilita l'evoluzione futura del sistema.

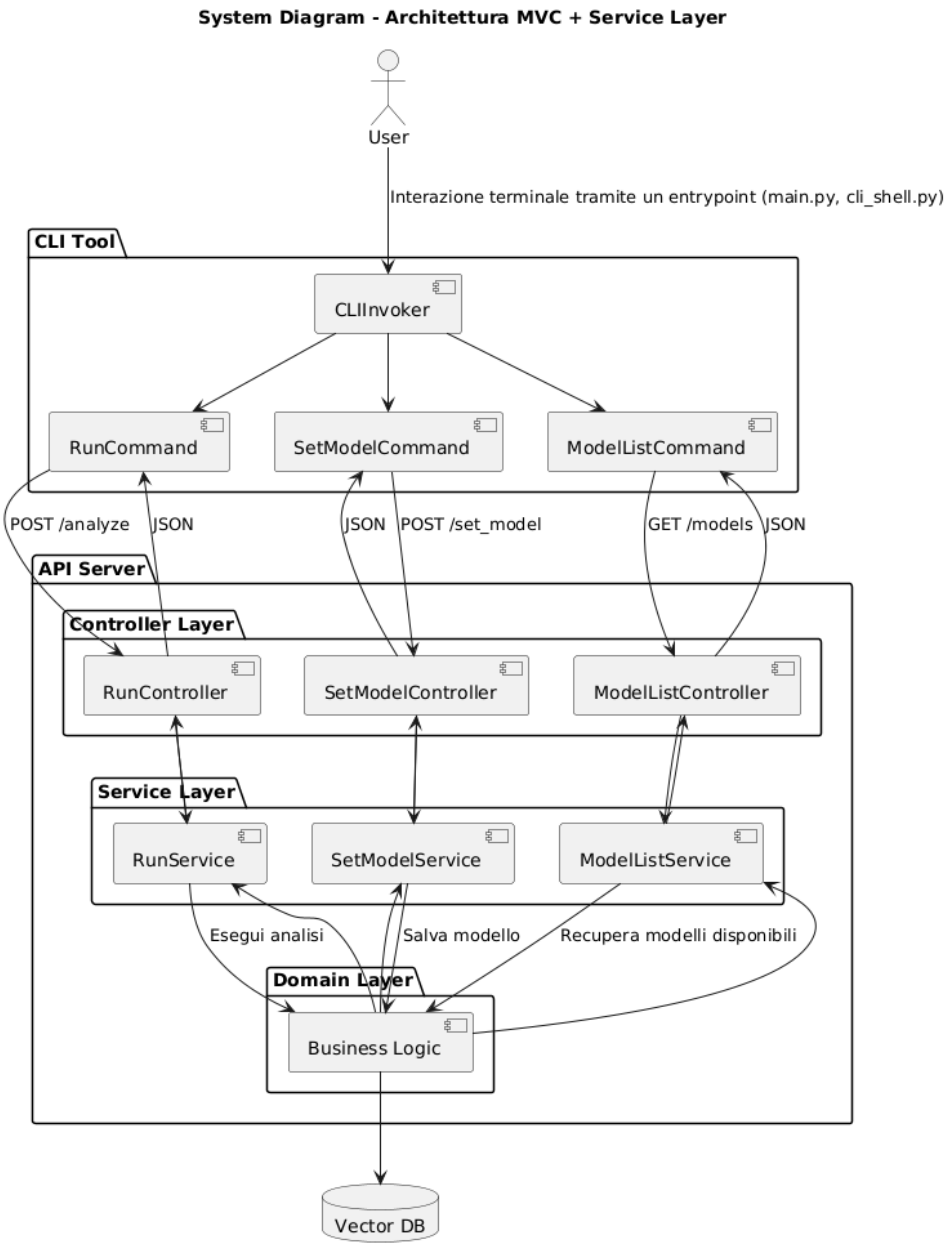

Figure 16: Nuova architettura client-server con pattern MVC e Service Layer

2.3.4 Implementazione

L'implementazione ha preso avvio con la rifattorizzazione dell'architettura del sistema, spostando tutti i package e i moduli contenenti la logica di business dal progetto CLI all'interno della work directory del server API.

Successivamente, è stata effettuata la definizione delle nuove route RESTful esposte dal server, ognuna associata a un controller specifico incaricato della gestione delle richieste provenienti dal client, delegando l'esecuzione della logica applicativa ai corrispondenti service.

La realizzazione dei service ha comportato un lavoro relativamente semplice, poiché la logica da implementare era in gran parte già definita nei vecchi `Command` della CLI. L'adattamento ha richiesto unicamente l'estrazione della logica in classi dedicate, ora riutilizzabili e facilmente testabili, che restituiscono i risultati sotto forma di oggetti JSON strutturati.

Dal lato client, i `Command` sono stati modificati per inviare richieste HTTP al server tramite il modulo `requests`, e successivamente stampare in console l'output ricevuto. In particolare, il `RunCommand`, oltre a richiedere l'analisi, riceve il risultato dell'analisi e genera il report come avveniva in precedenza.

Adesso lo script `run_pipeline` non deve più estrarre il file, ma riceve già la stringa del codice. Il compito di estrarre il codice dal file adesso è delegato a RunCommand che invia al server la richiesta contenente il codice estratto.

Sono stati inoltre aggiornati i file di configurazione per la containerizzazione, ovvero i `Dockerfile` dei due componenti (CLI e server) e il file `docker-compose.yaml`, per tenere conto delle nuove dipendenze introdotte e garantire il corretto avvio dei container nel nuovo assetto architetturale.

Infine, un cambiamento collaterale importante riguarda il componente `RetrievalEngine`, che in precedenza doveva interagire con il database vettoriale attraverso una richiesta esterna. Ora tale richiesta non è più necessaria, poiché il `ContractSearcher` è incorporato direttamente all'interno del backend server.

Possiamo definire l'AIS che contiene i seguenti componenti:

`AIS = { SetModelCommand, RunCommand, ModelListCommand, RetrievalEngine,`

`run_pipeline, Dockerfile(cli tool), Dockerfile(api server), docker-compose.yml }`

`CIS = { SetModelCommand, RunCommand, ModelListCommand, RetrievalEngine,`

`Dockerfile(cli tool), Dockerfile(api server), docker-compose.yml }`

$$\text{Recall} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{AIS}|} = \frac{7}{8} = 0.875$$

$$\text{Precision} = \frac{|\text{CIS} \cap \text{AIS}|}{|\text{CIS}|} = \frac{7}{7} = 1$$