Contents

Test Plan - LLM-SmartContractScanner

Nicola Tortora, Gaspare Galasso

1 Test Objectives

L'obiettivo principale della campagna di test è validare il comportamento del sistema esistente in modo approfondito prima dell'introduzione di modifiche evolutive. Questo processo consente di garantire l'affidabilità del software attuale ed è utile per eventuali regressioni o malfunzionamenti introdotti da cambiamenti futuri.

2 Methodology

È stata adottata una metodologia white-box, poiché la struttura interna del codice è nota e accessibile. Ciò ha permesso di progettare test che esercitano ogni logica condizionale e flusso di controllo. Il criterio di copertura utilizzato è il **branch coverage**, il quale assicura che ogni ramo (vero/falso) di ogni condizione logica venga valutato almeno una volta, aumentando così la probabilità di individuare difetti.

3 Test Approach

La strategia di testing utilizzata è la seguente:

- **Unit Tests**: Testano in isolamento le singole funzioni o classi, con l'ausilio di `mock` per simulare eventuali dipendenze esterne.
- **Integration Tests**: Verificano l'interazione tra componenti adiacenti.
- **System Tests**: Simulano scenari realistici e testano l'intero sistema, partendo dall'esecuzione del `main` e considerando input provenienti da file reali.

4 Pass/Fail Criteria

- **Pass**: Il test trova un errore, quando rileva un output errato, genera eccezioni non previste, oppure si blocca per crash.
- **Fail**: Il test non trava errori, rilevando un output conforme alle aspettative, senza generare eccezioni non gestite o comportamenti anomali.
- **Sospensione**: Il test viene sospeso e non valutato, se si rileva un errore dovuto a dipendenze esterne non raggiungibili.

5 Tools

I seguenti strumenti sono stati utilizzati per lo sviluppo, l'esecuzione e l'analisi dei test:

- `unittest` - Framework integrato di Python per la scrittura e l'esecuzione strutturata dei test.
- `unittest.mock` - Utilizzato per la simulazione di componenti esterni, attraverso stub e driver.
- `coverage.py` - Misura la copertura del codice, indicando le righe ed i rami eseguiti dai test.

6 Deliverables

I risultati della campagna di testing sono distribuiti come segue:

- `test plan` - Documento che descrive l'obiettivo, l'approccio e la copertura dei test.
- `test suites` - Codice dei test organizzato nella cartella `tests/` del repository.
- `test report` - Riassunto dei risultati ottenuti durante l'esecuzione.
- `coverage reports` - Statistiche dettagliate sulla copertura del codice.

7 Unit Testing

I test di unità coprono le singole funzioni e classi del sistema. Le dipendenze sono simulate tramite `mock`, permettendo così un'esecuzione rapida e isolata.

ID	Classe	Caratteristiche da Testare
TI-01	`CodeDescriptor`	Caricamento del prompt da file, generazione della descrizione tramite LLM, gestione di errori del modello.
TI-02	`CodeAnalysis`	Generazione del prompt, invocazione del modello LLM, parsing della risposta, gestione delle eccezioni.
TI-03	`VulnAnalysis`	Caricamento dettagli vulnerabilità da file JSON, costruzione del prompt, chiamata al LLM, gestione di file mancanti o chiavi assenti.
TI-04	`EmbeddingModel`	Inizializzazione del modello con parametri personalizzati, encoding del testo in vettori.
TI-05	`RetrievalEngine`	Descrizione e embedding del codice, richiesta POST al database, parsing della risposta JSON, gestione di errori HTTP.
TI-06	`OpenAILLM`	Comunicazione con le API di OpenAI, generazione di output a partire da prompt, gestione di timeout ed errori di rete.
TI-07	`HFLLM`	Integrazione con modelli HuggingFace, esecuzione del modello e parsing dell'output.
TI-08	`ContractSearcher`	Gestione delle query al database, filtraggio per tipo di vulnerabilità, restituzione dei risultati strutturati.

Table 1: Elenco dei Test Item

7.1 Unit Test TI-01

L'obiettivo di questi test è verificare il comportamento della classe `CodeDescriptor`, assicurandosi che vengano coperti tutti i rami (branch) del codice e che le eccezioni vengano gestite correttamente.

Metodi da testare Poiché lo stato interno dell'oggetto non viene modificato dopo l'esecuzione dei metodi, non è necessario testare lo stato, ma è sufficiente validare l'output dei metodi esposti.

- `__init__(LLMModel)`
- `get_description(str)`

Stub Per garantire un ambiente di test isolato e controllato, vengono utilizzati degli stub per simulare le dipendenze esterne:

- Stub del modello LLM utilizzato internamente
- Stub della funzione di lettura del file dei prompt (`load_string`)

7.1.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo atteso
TC_01_01	Inizializzazione corretta del prompt e del modello LLM	`test_init_successful`	L'oggetto viene creato correttamente con gli attributi valorizzati
TC_01_02	Gestione errore nel caricamento del prompt	`test_init_failure`	Viene sollevata un'eccezione di tipo `RuntimeError`
TC_01_03	Esecuzione corretta del metodo `get_description`	`test_get_description_successful`	Viene restituito l'output generato dal mock del modello LLM
TC_01_04	Gestione errore nel metodo `get_description`	`test_get_description_failure`	Viene restituito un messaggio di errore predefinito

7.2 Unit Test TI-02

L'obiettivo di questi test è verificare il comportamento della classe `CodeAnalysis`, assicurandosi che vengano coperti tutti i rami (branch) del codice e che le eccezioni vengano gestite correttamente.

Metodi da testare I metodi della classe non modificano lo stato interno dopo la loro esecuzione, quindi è sufficiente verificarne il comportamento attraverso il valore restituito.

- `__init__(LLMModel, int)`
- `get_possible_vulns(str)`
- `_parse_response(str)`

Stub Per creare un ambiente di test isolato vengono utilizzati stub per le dipendenze esterne:

- Stub del modello LLM (`generate`)
- Stub della funzione di caricamento del prompt (`load_string`)

7.2.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo atteso
TC_02_01	Inizializzazione corretta del prompt e del modello	`test_init_successful`	Oggetto istanziato correttamente
TC_02_02	Gestione errore durante il caricamento del prompt	`test_init_failure`	Viene sollevata un'eccezione `RuntimeError`
TC_02_03	Generazione corretta delle vulnerabilità dal metodo `get_possible_vulns`	`test_get_possible_vulns_success`	Viene restituita una lista di vulnerabilità
TC_02_04	Gestione eccezione nella generazione della risposta LLM	`test_get_possible_vulns_generation_error`	Viene restituita una lista vuota e un log di errore
TC_02_05	Gestione eccezione nel parsing della risposta LLM	`test_get_possible_vulns_parse_error`	Viene restituita una lista vuota e un log di errore
TC_02_06	Parsing corretto della risposta nel formato atteso	`test_parse_response_success`	Lista JSON estratta correttamente
TC_02_07	Parsing fallito per assenza della sezione ```list	`test_parse_response_missing_section`	Viene restituita una lista vuota e un log di errore

TC_02_08	Parsing fallito per errore JSON	`test_parse_response_json_error`	Viene restituita una lista vuota e un log di errore

7.3 Unit Test TI-03

L'obiettivo è quello di testare tutti i branch del codice della classe `VulnAnalysis` e sollevare tutte le possibili eccezioni, in particolare quelle relative alla lettura del file JSON e alla costruzione del prompt.

Metodi da testare Il comportamento della classe dipende esclusivamente dai metodi invocati e non da modifiche dello stato interno, pertanto l'unit test si concentra sull'invocazione e sulle eccezioni gestite.

- __init__(LLMModel)
- get_vuln_analysis(str, str)
- _get_prompt(str, dict, str)
- _get_vuln_details(str)

Stub Per garantire un ambiente di test isolato, sono stati impiegati stub e mock per simulare elementi esterni:

- Stub del modello LLM invocato nel metodo `generate()`.
- Stub del file system tramite `mock_open` per simulare l'apertura del file JSON contenente le vulnerabilità.
- Stub della funzione `load_string` per il caricamento del prompt statico.

7.3.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_03_01	Analisi corretta con input valido	test_get_vuln_analysis_success	Output generato dal modello LLM
TC_03_02	Analisi su codice vuoto	test_get_vuln_analysis _empty_code	Viene restituito un messaggio di errore "Analisi impossibile su codice vuoto"
TC_03_03	Errore nel parsing del JSON (formato non valido)	test_get_vuln_analysis _json_decode_error	Viene restituito un messaggio di errore "Impossibile caricare i dettagli della vulnerabilità"
TC_03_04	File dei dettagli della vulnerabilità non trovato	test_get_vuln_analysis _file_not_found	Viene restituito un messaggio di errore "Impossibile caricare i dettagli della vulnerabilità"

TC_03_05	Vulnerabilità non presente nel file JSON	test_get_vuln_analysis _key_error	Viene restituito un messaggio di errore per chiave mancante
TC_03_06	Dettagli vulnerabilità incompleti (campo mancante)	test_get_vuln_analysis _missing_detail_field	Viene sollevata una `KeyError` gestita nel metodo
TC_03_07	Controllo della struttura del prompt	test_get_prompt_structure	Prompt costruito correttamente con tutti i campi presenti

7.4 Unit Test TI-04

L'obiettivo è quello di testare il comportamento della classe `EmbeddingModel`, assicurandosi che gestisca correttamente l'inizializzazione del modello, la generazione dell'embedding e le condizioni eccezionali.

Motivazione In questo caso non vengono utilizzati stub per simulare il comportamento del modello, in quanto l'obiettivo principale del test è verificare l'integrazione e l'effettiva capacità del modello reale di generare embedding. La classe incapsula completamente la logica di interazione con `SentenceTransformer`, perciò testare il comportamento reale fornisce informazioni concrete sul funzionamento dell'oggetto; è importante quindi eseguire i test solo se si ha la certezza della raggiungibilità del servizio esterno, per evitare **flaky tests**.

Metodi da testare

- __init__(model_name: str, device: str, instruction: str)
- encode(text: str)

Dipendenze reali

- Modello reale: `hkunlp/instructor-xl`
- Framework: `sentence-transformers`

7.4.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_04_01	Fallimento nell'inizializzazione del modello	test_init_failure	Eccezione `RuntimeError` sollevata
TC_04_02	Fallimento del metodo `encode` (simulato con mock)	test_encode_failure	Ritorno di lista vuota
TC_04_03	Generazione dell'embedding con input valido	test_encode_valid_input	Embedding restituito, lista di float
TC_04_04	Gestione input estremamente lungo (oltre il limite token)	test_encode_exceeds_max_tokens	Viene sollevata una eccezione: "Input troppo lungo"
TC_04_05	Generazione embedding con stringa vuota	test_encode_empty_input	Embedding comunque restituito

7.5 Unit Test TI-05

L'obiettivo è quello di testare il comportamento della classe `RetrievalEngine`, verificando il corretto funzionamento del metodo `get_similar_contracts` in condizioni normali ed eccezionali.

Metodi da testare

- __init__(url: str, descriptor: CodeDescriptor, embedder: EmbeddingModel)
- get_similar_contracts(str)

Stub Per isolare la logica della classe, sono stati utilizzati stub per simulare i componenti esterni:

- Stub di `CodeDescriptor`: fornisce una descrizione sintetica del contratto.
- Stub di `EmbeddingModel`: restituisce un vettore embedding artificiale.
- La funzione `requests.post` è stata mockata per simulare il comportamento del servizio HTTP di retrieval.
- La funzione `map_vulnerability` è stata patchata per rimuovere dipendenze esterne alla logica da testare.

7.5.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_05_01	Contratti simili trovati con input valido	test_get_similar_contracts_success	Restituzione di una lista di vulnerabilità ordinate
TC_05_02	Errore HTTP durante la richiesta	test_get_similar_contracts _http_error	Solleva l'eccezione RequestException
TC_05_03	Risposta malformata dal server (non JSON)	test_get_similar_contracts _invalid_json	Solleva l'eccezione JSONDecodeError
TC_05_04	Errore nel componente `CodeDescriptor`	test_get_similar_contracts _descriptor_error	Solleva un'eccezione e restituisce un messaggio di errore

7.6 Unit Test TI-06

L'obiettivo è quello di testare il comportamento della classe `OpenAILLM` in fase di inizializzazione e durante la generazione di risposte tramite API OpenAI. In particolare, si testano le condizioni di errore comuni, come API key invalida, URL errato, modello non trovato e prompt troppo lungo.

Metodi da testare

- __init__(api_key: str, model_name: str, base_url: str)
- generate(prompt: str)

Nota sullo stub Non è stato utilizzato alcuno stub per il metodo `generate`, al fine di testare direttamente il comportamento reale del client OpenAI e le relative eccezioni. Questo approccio è giustificato dalla necessità di verificare la robustezza della gestione degli errori che si verificano nel contesto di utilizzo reale; è importante quindi eseguire i test solo se si ha la certezza della raggiungibilità del servizio esterno, per evitare **flaky tests**.

7.6.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_06_01	Inizializzazione fallita con URL non valido	test_init_invalid_url	Viene sollevata eccezione con messaggio: "Errore nella configurazione del client OpenAI."
TC_06_02	Inizializzazione fallita con API key invalida	test_init_invalid_apikey	Viene sollevata eccezione: "Errore di autenticazione"
TC_06_03	Inizializzazione fallita con modello inesistente	test_init_invalid_modelname	Viene sollevata eccezione: "Errore: modello LLM non trovato"
TC_06_04	Inizializzazione corretta con parametri validi	test_init_success	L'oggetto viene creato correttamente
TC_06_05	Errore di generazione con API key invalida	test_generate_invalid_apikey	Eccezione sollevata: "Errore di autenticazione"
TC_06_06	Errore di generazione con modello inesistente	test_generate_invalid_modelname	Eccezione sollevata: "modello LLM non trovato"
TC_06_07	Errore di generazione con URL non raggiungibile	test_generate_invalid_url	Eccezione sollevata: "Errore di connessione"

TC_06_08	Errore con prompt troppo lungo	test_generate_long_prompt	Eccezione sollevata: “prompt non valido o troppo lungo”

7.7 Unit Test TI-07

L'obiettivo è verificare il comportamento della classe `HFLLM` durante l'inizializzazione e la generazione del testo. I test si concentrano su casi di inizializzazione fallita (modello inesistente, pipeline non creabile) e sull'effettiva generazione di testo, inclusa la gestione di input molto lunghi.

Metodi da testare

- __init__(model_name: str, device: str)
- generate(prompt: str)

Nota sull'uso del modello reale Anche in questo caso si è scelto di non utilizzare stub, al fine di testare la robustezza della classe rispetto a errori reali derivanti da modelli inesistenti, configurazioni errate o limiti nella generazione. La classe incapsula direttamente il comportamento del modello HuggingFace, quindi è fondamentale verificarne l'integrazione end-to-end; è importante quindi eseguire i test solo se si ha la certezza della raggiungibilità del servizio esterno, per evitare **flaky tests**.

7.7.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_07_01	Inizializzazione fallita con modello inesistente	test_init_invalid_model	Eccezione sollevata con messaggio: "Errore nel caricamento del modello: ¡nome¿"
TC_07_02	Inizializzazione fallita in fase di pipeline	test_init_pipeline_failure	Eccezione sollevata con messaggio: "Errore nel caricamento della pipeline"
TC_07_03	Inizializzazione corretta	test_init_success	Oggetto correttamente creato
TC_07_04	Generazione con prompt valido	test_generate_valid_prompt	Output testuale non vuoto
TC_07_05	Generazione con prompt molto lungo	test_generate_long_prompt	Viene restituito un testo oppure viene sollevata un'eccezione per superamento limiti
TC_07_06	Generazione con prompt vuoto	test_generate_empty_prompt	Viene comunque restituito un testo (modellato come comportamento previsto)

7.8 Unit Test TI-8

La classe `ContractSearcher` è responsabile dell'interrogazione del database vettoriale Qdrant al fine di reperire vulnerabilità simili a partire da un embedding numerico. I test condotti mirano a verificare il corretto comportamento della funzione `search_vulns`, inclusi i casi di successo e di errore.

Metodi da testare

- __init__(collection_name: str, url_db: str)
- search_vulns(vector: list)

Stub È stato impiegato utilizzato per simulare il comportamento del client Qdrant. L'uso di mock consente di testare la logica della classe in isolamento, evitando dipendenze da un database esterno realmente attivo.

7.8.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_08_01	Inizializzazione corretta del client Qdrant	test_init_success	Oggetto `ContractSearcher` creato correttamente
TC_08_02	Inizializzazione con URL errato	test_init_invalid_url	Viene sollevata un'eccezione di connessione
TC_08_03	Ricerca con risultati validi	test_search_vulns_with_results	Restituisce lista di dizionari contenenti `contract_id`, `vulnerability`, `score`
TC_08_04	Ricerca con nessun risultato	test_search_vulns_empty_results	Restituisce lista vuota
TC_08_05	Ricerca con payload incompleto	test_search_vulns _missing_payload	Solleva eccezione `KeyError` per campi mancanti
TC_08_06	Ricerca su collezione non esistente	test_search _vulns_collection_not_found	Solleva eccezione `UnexpectedResponse` da parte di Qdrant

8 Integration Testing

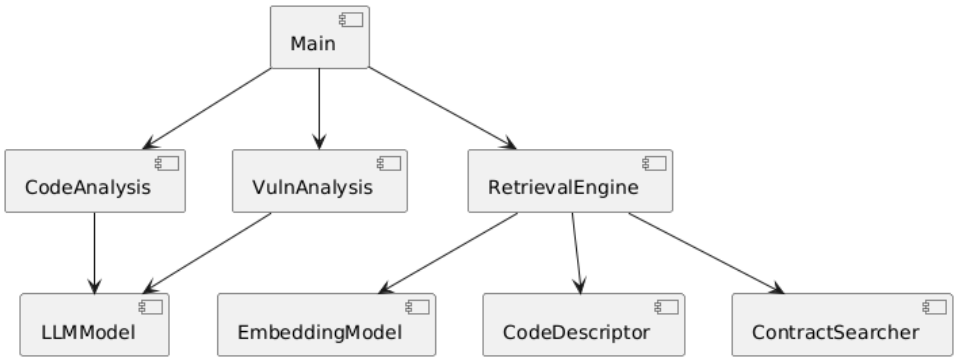

Figure 1: Grafo delle dipendenze

Per il test di integrazione utiliziamo una strategie bottom-up, ovvero testiamo prima le interazioni tra i componenti più in basso, e successivamente gli elementi più in alto

ID	Classe	Caratteristiche da Testare
TI-09	`RetrievalEngine -> EmbeddingModel, ContractSearcher, CodeDescriptor`	Caricamento del prompt da file, generazione della descrizione tramite LLM, gestione di errori del modello.
TI-10	`CodeAnalysis -> LLMModel`	Caricamento del prompt da file, generazione della descrizione tramite LLM, gestione di errori del modello.
TI-11	`VulnAnalysis -> LLMModel`	Caricamento del prompt da file, generazione della descrizione tramite LLM, gestione di errori del modello.

Table 10: Elenco dei Test Item

8.1 Integration Test TI-9

L'biettivo è verificare il corretto comportamento del componente *RetrievalEngine* nelle sue interazioni con i moduli sottostanti:

- EmbeddingModel
- CodeDescriptor
- endpoint HTTP esterno (Qdrant DB)

Stab Il metodo EmbeddingModel.encode è sostituito da uno stub che restituisce un vettore statico, in modo da evitare la dipendenza dal modello esterno. Per lo stesso motivo il metodo LLMModel.generate è simulato per evitare computazione reale del LLM esterno. Il metodo requests.post è mockato per restituire risposte HTTP simulate, evitando interazioni reali con il database Qdrant.

8.1.1 Test Cases

ID	**Obiettivo**	**Funzione di test**	**Oracolo**
TC_09_01	Verifica della retrieval su codice semplice	test_retrieval_simple_contract	Lista di vulnerabilità con score, mappate correttamente
TC_09_02	Gestione codice vuoto	test_retrieval_empty_code	Nessun risultato deve essere restituito
TC_09_03	Fallimento della fase di embedding	test_embedder_failure	Solleva un eccezione
TC_09_04	Fallimento della descrizione del codice	test_descriptor_failure	Solleva un eccezione
TC_09_05	Fallimento del componente searcher (errore HTTP)	test_searcher_failure	Solleva un eccezione RequestException

8.2 Integration Test TI-10

Verificare il corretto funzionamento del modulo CodeAnalysis in relazione al comportamento del modello LLM (LLMModel). Lo scopo è assicurarsi che il prompt venga correttamente generato, passato al modello e che l'output sia gestito coerentemente, anche in presenza di eccezioni.

Stab Viene creato un stub per il metodo LLMModel.generate, per evitare richieste reali a servizi esterni e restituire risposte controllate.

8.2.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_10_01	Generazione corretta del prompt e invocazione del modello	test_code_analysis_success	Risultato testuale coerente con la simulazione
TC_10_03	Fallimento del modello LLM	test_code_analysis_llm_failure	L'eccezione viene gestita, ritorna lista vuota

8.3 Integration Test TI-11

Verificare l'interazione tra il modulo VulnAnalysis e il modello LLMModel, accertando che il modello risponda con analisi coerenti e le eccezioni del modello vengano gestite.

Stab Viene creato un stub per il metodo LLMModel.generate, per evitare richieste reali a servizi esterni e restituire risposte controllate.

8.3.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_11_01	Generazione corretta dell'analisi per una vulnerabilità nota	test_code_analysis_success	Viene restituita una stringa coerente
TC_11_02	Fallimento del modello LLM durante la generazione dell'analisi	test_code_analysis_llm_failure	Viene sollevata un'eccezione controllata

9 System Test

Verificare il corretto funzionamento del sistema nella sua interezza a partire dall'esecuzione del modulo `main.py`, utilizzando file di input rappresentativi (validi, vuoti, non validi o inesistenti).

I test simulano l'esecuzione del programma principale con vari tipi di input, verificando che il comportamento del sistema sia corretto. Per testare l'intero flusso, si passa un percorso assoluto al file sorgente del contratto, osservando l'output stampato a `stdout`.

Dipendenze Esterne Poiché il programma interagisce con:

- LLM locale o remoto (tramite un'interfaccia come `generate()`),
- Motore di embedding,
- Database vettoriale (es. Qdrant via API HTTP),

è fondamentale che tali dipendenze siano attive e accessibili per evitare **test flaky**.

9.1 Test Cases

ID	Obiettivo	Funzione di test	Oracolo
TC_ST_01	Contratto valido analizzato correttamente	test_valid_contract	Output contiene "Analisi completata"
TC_ST_02	Contratto vuoto non analizzabile	test_empty_contract	Output contiene "Codice vuoto"
TC_ST_03	Contratto non valido (es. sintatticamente errato)	test_invalid_contract	Output contiene "Errore di sintassi nel codice inserito"
TC_ST_04	File non trovato	test_file_not_found	Eccezione FileNotFoundError sollevata
TC_ST_05	Percorso a cartella invece che file	test_directory_instead_of_file	Eccezione FileNotFoundError sollevata