Contents

Test Plan - Change Requests

Nicola Tortora, Gaspare Galasso

August 14, 2025

1. Test Objectives

Questo documento descrive il piano di testing per le **Change Requests** applicate sul sistema.

- Verificare la correttezza delle modifiche introdotte.
- Aggiornare e ampliare le test suite per riflettere i cambiamenti di comportamento.
- Eseguire test di regressione per garantire la non introduzione di malfunzionamenti su funzionalità esistenti.

2. 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.

3. Tools

- `pytest`, `unittest` per unit/integration test
- `GitHub` Actions per automazione dei test di sistema prima del rilascio

4. Esecuzione

- **Ambiente containerizzato (Docker Compose)**: per test di sistema.
- **Ambiente locale**: per unit e integration test.

5. 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.

6. Regression Testing

Il **regression testing** ha lo scopo di verificaretr che le modifiche inodotte nel sistema non abbiano compromesso il corretto funzionamento delle funzionalità già esistenti.

I test di regressione verranno eseguiti **dopo la validazione e l'accettazione di una Change Request**, in particolare, dopo l'esecuzione e il superamento dei test unitari e di integrazione sui nuovi componenti introdotti;

la fase successiva vedrà l'esecuzione dei test di sistema e l'eventuale rilascio.

La suite di regression testing sarà composta dai **test preesistenti alla modifica**, eventualmente aggiornati in base all'impatto delle modifiche stesse. In linea generale, per la selezione dei test di regressione si potrebbero adottare diversi criteri, ma dato il numero contenuto di test, risulta praticabile rieseguire l'intera suite ad ogni ciclo di modifica.

Tutti i test preesistenti devono continuare a produrre gli stessi esiti positivi, altrimenti la regressione dovrà essere analizzata e risolta prima di proseguire con le fasi successive del testing.

7. System Test

I test di sistem, oltre ad essere eseguiti in locale, verranno eseguiti anche in ambiente containerizzato attravero file docker-compose (docker-compose.system-test.yml)

8. Change Request 1

8.1. Descrizione

La modifica introduce una nuova modalità di invocazione del tool CLI attraverso una shell interattiva (REPL), mantenendo invariata la logica di business.

8.2. Impact Analysis

Dall'analisi dell'impatto risulta che l'unico elemento colpito nei test esistenti è il **System Test**. Le unità e componenti integrati aggiunti sono nuovi e quindi necessitano di specifici test dedicati.

8.3. Approccio

Viene adottato un approccio **Black Box**, con derivazione dei test tramite **Category Partition Method**, partizionando gli input in classi funzionali rilevanti e valutando combinazioni valide e non valide di parametri d'ingresso.

8.4. Livelli di Test

- **Unit Test**: coprono la nuova logica di parsing CLI e gestione REPL.
- **Integration Test**: verificano la corretta inizializzazione del contesto applicativo (AppContext) e l'interazione tra CLIInvoker, Command e pipeline logiche.
- **System Test**: modificato per coprire il nuovo comportamento interattivo e mantenere la compatibilità con l'esecuzione batch.

8.5. Unit Test

8.5.1 Test di Unità - `CLIInvoker.set_command()`

Questa sezione descrive i test di unità per il metodo `set_command(args)` della classe `CLIInvoker`, responsabile del parsing degli argomenti CLI e dell'inizializzazione del comando corretto.

Funzionalità

Il metodo interpreta gli argomenti forniti da riga di comando e assegna dinamicamente a `self.command` l'istanza corrispondente:

- `RunCommand` richiede: `--filepath`, `--model`, `--vuln-limit`, `--contract-limit`
- `SetModelCommand` richiede: `--model_name`, `--source`, (opzionali: `--api_key`, `--base_url`)
- `ModelListCommand` non accetta alcun parametro

Categorie dei parametri

Parametro	**Categorie**
comando	valido: `run`, `set-model`, `model-list`; non esistente
`--filepath`	presente, non presente
`--vuln-limit`	presente, non presente
`--contract-limit`	presente, non presente
`--model`	presente, non presente
`--model_name`	presente, non presente
`--source`	presente, non presente
`--api_key`	presente, non presente
`--base_url`	presente, non presente

Table 1: Categorie per il metodo `CLIInvoker.set_command()`

Test Cases

ID	Comando	Combinazione parametri	Esito atteso
C1	run	tutti presenti e validi	Comando inizializzato come `RunCommand`
C2	run	manca `--filepath`	Errore: argomento obbligatorio mancante
C3	run	manca `--vuln-limit`	inizializzazione valida (parametri opzionali)
C4	run	manca `--contract-limit`	inizializzazione valida (parametri opzionali)
C5	run	manca `--model`	Errore: argomento obbligatorio mancante
C6	set-model	tutti presenti e validi	Comando inizializzato come `SetModelCommand`
C7	set-model	manca `--model_name`	Errore: argomento obbligatorio mancante
C8	set-model	manca `--source`	Errore: argomento obbligatorio mancante
C9	set-model	senza `--api_key`, `--base_url`	Inizializzazione valida (parametri opzionali)
C10	model-list	nessun argomento	Comando inizializzato come `ModelListCommand`
C11	model-list	con argomenti non previsti	Errore di parsing
C12	comando non esistente	qualsiasi combinazione	Errore di parsing

Table 2: Casi di test per `CLIInvoker.set_command()`

8.5.2 Test di Unità - Comandi CLI

Questa sezione descrive i test di unità per le classi derivate da `Command`.

Funzionalità: Inizializzazione e `RunCommand.execute()`

Parametri e categorie

Parametro	Categorie
`model`	vuota, piena
`filepath`	vuota, piena
`vuln-limit`	negativo, zero, positivo
`contract-limit`	negativo, zero, positivo

Table 3: Categorie per `RunCommand.execute()`

Casi di test derivati

ID	`model`	`filepath`	`vuln-limit`	`contract-limit`	Esito atteso
RC1	piena	piena	positivo	positivo	Pipeline eseguita correttamente
RC2	vuota	piena	positivo	positivo	Errore: modello mancante
RC3	piena	vuota	positivo	positivo	Errore: percorso del file mancante
RC4	piena	piena	negativo	positivo	Errore: flag non può essere negativo
RC5	piena	piena	positivo	negativo	Errore: flag non può essere negativo
RC6	piena	piena	zero	zero	Errore: la pipeline non può essere eseguita con 0 vulnerabilità considerate

Table 4: Casi di test per `RunCommand.execute()`

Funzionalità: Inizializzazione e `SetModelCommand.execute()`

Parametri e categorie

Parametro	Categorie
`model_name`	vuota, piena
`source`	valida: openai/huggingface, non valida
`api_key`	vuota, piena
`base_url`	vuota, piena

Table 5: Categorie per `SetModelCommand.execute()`

Casi di test derivati

ID	model_name	source	api_key	base_url	Esito atteso
SC1	piena	openai	piena	piena	Modello aggiunto con successo
SC2	vuota	openai	piena	piena	Errore: nome modello mancante
SC3	piena	non valida	-	-	Errore: sorgente non supportata
SC4	piena	openai	vuota	piena	Errore: API key mancante per OpenAI
SC5	piena	huggingface	vuota	vuota	Modello huggingface salvato
SC6	piena	huggingface	vuota	piena	Errore: parametro base_url non necessario
SC7	piena	huggingface	piena	vuota	Errore: parametro api_key non necessario

Table 6: Casi di test per `SetModelCommand.execute()`

Funzionalità: Inizializzazione e `ModelListCommand.execute()`

Parametri: Nessuno.

Casi di test derivati

ID	Contenuto Configurazione	Esito atteso
MC1	configurazione con modelli presenti	Elenco modelli stampato correttamente

Table 7: Casi di test per `ModelListCommand.execute()`

8.5.3 Test di Unità - Inizializzazione della classe `AppContext`

Questa sezione descrive i test di unità per il costruttore della classe `AppContext`, che incapsula la configurazione e l'inizializzazione dei moduli principali del sistema.

Funzionalità

: Inizializzazione dei componenti: `CodeAnalysis`, `VulnAnalysis`, `RetrievalEngine`.

Parametri e categorie

Parametro	Categorie
`model`	presente nel file di config, assente nel file
`vuln_limit`	positivo, zero, negativo
`contract_limit`	positivo, zero, negativo

Table 8: Categorie per il costruttore di `AppContext`

Casi di test derivati

ID	`model`	`vuln_limit`	`contract_limit`	Esito atteso
AC1	presente	positivo	positivo	Oggetto inizializzato: tutti i componenti non nulli
AC2	assente	positivo	positivo	Errore: modello non trovato nel file di configurazione
AC3	presente	negativo	positivo	Errore vuln-limit non può essere negativo
AC4	presente	positivo	negativo	Errore contract-limit non può essere negativo
AC5	presente	zero	zero	Errore:Il contesto non può essere inizializzato con zero vulnerabilità da considerare

Table 9: Casi di test per `AppContext.__init__`

Verifiche sullo stato

Dopo l'inizializzazione si deve verifica che lo stato dell'oggetto sia correttamente impostato:

- `context.get_code_analyzer()` restituisce un oggetto `CodeAnalysis`.
- `context.get_vuln_analyzer()` restituisce `VulnAnalysis`.
- `context.get_retrieval_engine()` è un `RetrievalEngine`.

8.5.4 Test di Unità - Classe `LLMFactory`

Questa sezione descrive i test di unità per la classe `LLMFactory`, responsabile della costruzione del modello LLM corretto sulla base della configurazione fornita.

Funzionalità testate

Il metodo `build(config)` prende in ingresso un dizionario di configurazione e restituisce un'istanza del modello LLM appropriato, a seconda del valore del campo `source`.

- Se `source = "openai"` ⇒ ritorna istanza di `OpenAILLM`.
- Se `source = "huggingface"` ⇒ ritorna istanza di `HFLLM`.
- Se `source` non è supportato ⇒ solleva `ValueError`.

Categorie del parametro `config`

Parametro	**Categorie**
`source`	openai, huggingface, non supportato
`model_name`	valido, mancante
`api_key` (per openai)	presente, mancante
`base_url` (per openai)	presente, assente
`device` (per huggingface)	presente, assente

Table 10: Categorie del parametro di ingresso per `LLMFactory.build()`

Casi di test derivati

ID	`source`	Altri campi	**Esito atteso**
LF1	openai	tutti presenti	Istanza di `OpenAILLM`
LF2	openai	manca `base_url`	Istanza di `OpenAILLM` con URL di default
LF3	openai	manca `api_key`	Errore durante inizializzazione modello
LF4	huggingface	`model_name` presente, `device` presente	Istanza di `HFLLM`
LF5	huggingface	manca `device`	Istanza con device = "auto"
LF6	non supportato	qualsiasi	`ValueError` sollevato

Table 11: Casi di test per `LLMFactory.build()`

8.5.5 Test di Unità - Classe `ConfigManager`

Questa sezione documenta i test di unità per la classe `ConfigManager`, responsabile della gestione del file di configurazione del sistema (`config.json`). I test sono progettati secondo l'approccio **Black-Box**, utilizzando la tecnica del **Category Partitioning**, e si focalizzano sulle funzionalità esposte e sul comportamento osservabile a fronte di input validi e non validi.

Funzionalità testate

- `load_config(model_name)`: carica la configurazione associata al modello richiesto.
- `add_model_config(model_config)`: aggiunge un nuovo modello alla configurazione.
- `save_config(config)`: salva un dizionario come file `config.json`.

Categorie dei parametri

Metodo	Categorie di input
`load_config(model_name)`	nome presente; nome assente nella configurazione
`add_model_config(model_config)`	config. complete; config. mancante di campi chiave
`save_config(config)`	dizionario ben formato, dizionario malformato

Table 12: Categorie di input per i metodi della classe `ConfigManager`

Casi di test derivati

ID	Metodo	Esito atteso
CM1	load_config(model presente)	Ritorna la configurazione corretta, con campo `llm` filtrato
CM2	load_config(model assente)	Solleva `ValueError`: modello non trovato
CM3	add_model_config($model_{config}$)	Configurazione aggiornata con nuovo modello
CM4	add_model_config($model_{configmancantecampi}$)	Modello comunque aggiunto, validazione demandata altrove
CM5	save_config(config valido)	File aggiornato correttamente
CM6	save_config(config malformato)	Potenziale eccezione se oggetto non serializzabile

Table 13: Casi di test per i metodi della classe `ConfigManager`

8.6. Integration Test

In questa fase vengono verificate le interazioni tra le classi del sistema attraverso test di integrazione strutturati secondo un approccio **bottom-up**. Seguiamo la seguente convenzione per identificare le dipendenze tra classi: *una classe A dipende da una classe B se A la utilizza direttamente (composizione) o ne eredita.*

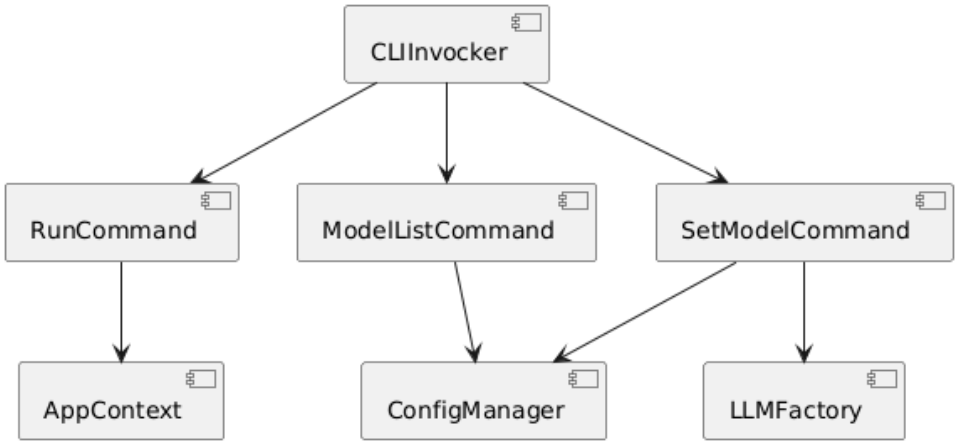

Figure 1: Relazioni di dipendenza tra le classi principali

Le classi foglia, cioè quelle che non dipendono da altre componenti, sono già coperte dai test di unità. L'integrazione procede quindi testando i livelli superiori della gerarchia, assicurando che i moduli che le utilizzano collaborino correttamente. Le principali interazioni testate sono:

- `CLIInvoker` → `RunCommand` → `AppContext` → `ConfigManager`, `LLMFactory`
- `CLIInvoker` → `ModelListCommand` → `ConfigManager`
- `CLIInvoker` → `SetModelCommand` → `LLMFactory`, `ConfigManager`

Per ogni catena di dipendenza, i test si concentrano sul corretto passaggio di dati tra i moduli, sull'invocazione dei metodi attesi e sulla gestione delle eccezioni. Le dipendenze indirette vengono simulate o mockate quando necessario per isolare il comportamento specifico sotto test.

8.6.1 Test di Integrazione - `CLIInvoker.run()` → `RunCommand` → `AppContext`

Questa sezione descrive i test di integrazione tra le componenti `CLIInvoker`, `RunCommand`, `AppContext`, `ConfigManager` e `LLMFactory`. L'obiettivo è verificare il corretto flusso di dati e la cooperazione tra i moduli responsabili dell'esecuzione del comando `run`, senza analizzare le implementazioni interne (approccio **Black-Box**).

Funzionalità

L'invocazione del comando `run` da riga di comando comporta:

1. Parsing e inizializzazione del comando tramite `CLIInvoker`.
2. Esecuzione della logica `RunCommand.execute()`, che interagisce con `AppContext`.
3. `AppContext` utilizza `ConfigManager` per leggere la configurazione e `LLMFactory` per ottenere un'istanza del modello richiesto.

Categorie dei parametri

Parametro	Categorie
`--filepath`	F1: File valido, F2: File inesistente
`--model`	M1: Modello installato, M2: Modello non installato

Table 14: Categorie per il comando `run()`

Casi di test derivati

ID	Filepath	Model	Comando	Esito atteso
T1	F1 (valido)	M1 (installato)	run	Analisi completata con successo, output generato
T2	F1 (valido)	M2 (non installato)	run	Il modello viene scaricato tramite `LLMFactory`, quindi l'analisi prosegue
T3	F2 (inesistente)	M1 (installato)	run	Errore gestito: file non trovato, l'analisi non parte
T4	F2 (inesistente)	M2 (non installato)	run	Errore gestito: file non trovato, modello comunque non scaricato

Table 15: Casi di test per l'integrazione `CLIInvoker` → `RunCommand` → `AppContext`

8.6.2 Test di Integrazione - `CLIInvoker.set_model()` → `SetModelCommand` → `LLMFactory`, `ConfigManager`

Questa sezione descrive i test di integrazione per il comando `set-model`, che coinvolge l'interazione tra `CLIInvoker`, `SetModelCommand`, `LLMFactory` e `ConfigManager`. I test sono condotti secondo l'approccio **Black-Box** con la tecnica di **Category Partitioning**, concentrandosi su combinazioni di parametri rilevanti per la corretta configurazione di un modello linguistico.

Funzionalità

Il comando `set-model` consente di configurare un modello specificando:

- Il nome del modello da impostare (`--model_name`).
- Il provider (`--source`: `openai` oppure `huggingface`).
- Le credenziali necessarie in caso di provider `openai` (`--api_key`, `--base_url`).

`SetModelCommand` delega la creazione del modello a `LLMFactory`, mentre `ConfigManager` si occupa di salvare la configurazione aggiornata.

Categorie dei parametri

Parametro	Categorie
`--source`	S1: `openai`, S2: `huggingface`
`--model_name`	M1: stringa valida
`--api_key`	A1: presente, A2: assente (richiesto solo se `source=openai`)
`--base_url`	B1: stringa valida (richiesta solo se `source=openai`)

Table 16: Categorie per il comando `set-model`

Casi di test derivati

ID	Source	Model Name	API Key	Base URL	Esito atteso
T1	openai	valido	presente	valida	Configurazione salvata correttamente, istanza creata da LLMFactory
T2	openai	valido	assente	valida	Errore: `--api_key` obbligatorio per OpenAI
T3	openai	valido	presente	assente	Errore: `--base_url` obbligatorio per OpenAI
T4	huggingface	valido	assente	assente	Configurazione salvata, LLM creato tramite HuggingFace (nessuna credenziale richiesta)
T5	huggingface	valido	presente	valida	Configurazione salvata, ma parametri `api_key` e `base_url` ignorati

Table 17: Casi di test per l'integrazione `CLIInvoker` → `SetModelCommand` → `LLMFactory`, `ConfigManager`

8.6.3 Test di Integrazione - `CLIInvoker.model_list()` → `ModelListCommand` → `ConfigManager`

Questa sezione descrive i test di integrazione per il comando `model-list`, che coinvolge l'interazione tra `CLIInvoker`, `ModelListCommand` e `ConfigManager`. Il comando è progettato per elencare i modelli configurati dall'utente, senza richiedere parametri.

Funzionalità

Alla chiamata del comando `model-list`, `CLIInvoker` inizializza una nuova istanza di `ModelListCommand`. Quest'ultima interroga `ConfigManager` per ottenere e stampare la lista dei modelli configurati localmente, compreso il modello attualmente selezionato.

Categorie dei parametri

Poiché il comando non richiede input da riga di comando, le uniche varianti rilevanti riguardano lo stato interno di `ConfigManager`.

Elemento	**Categorie**
Modelli configurati	Nessun modello, Un solo modello, Più modelli

Table 18: Categorie per il comando `model-list` (stato di `ConfigManager`)

Casi di test derivati

ID	**Modelli configurati**	**Esito atteso**
M1	Nessun modello	Messaggio: "Nessun modello configurato"
M2	Un solo modello	Output: elenco con un modello e indicazione di "selezionato"
M3	Più modelli	Elenco completo, evidenziato il modello attivo

Table 19: Casi di test per l'integrazione `CLIInvoker` → `ModelListCommand` → `ConfigManager`

8.7. System Test

In questa sezione vengono presentati i test di sistema aggiornati per la nuova versione del sistema. L'approccio utilizzato è il **black-box testing**. I test coprono tutti i comandi CLI disponibili, includendo sia i casi preesistenti, riadattati alla nuova implementazione, sia quelli introdotti per testare le nuove funzionalità.

Tutti i test vengono eseguiti in ambiente containerizzato, replicando il contesto operativo di produzione.

Vecchi test riadattati

ID	Descrizione	Esito Atteso
TC_ST_01	Contratto valido analizzato correttamente	Output contiene "Analisi completata"
TC_ST_02	Contratto vuoto non analizzabile	Output contiene "Codice vuoto"
TC_ST_03	Contratto non valido (es. sintatticamente errato)	Output contiene "Errore di sintassi nel codice inserito"
TC_ST_04	File non trovato	Sollevata eccezione FileNotFoundError

Table 20: Test di sistema - casi preesistenti riadattati

Nuovi test case introdotti

ID	Descrizione	Esito Atteso
TC_ST_05	Invocazione comando `model-list`	Output contiene elenco dei modelli installati
TC_ST_06	Impostazione modello da HuggingFace	Output contiene conferma di impostazione
TC_ST_07	Impostazione modello da OpenAI con API key e URL	Output contiene conferma di salvataggio configurazione
TC_ST_08	Impostazione modello OpenAI senza API key	Errore di input segnalato (argparse)
TC_ST_09	Esecuzione `run` senza modello selezionato	Output di errore: modello non configurato
TC_ST_10	Comando non riconosciuto	Errore di parsing

Table 21: Test di sistema - nuovi casi introdotti

9. Change Request 2

9.1. Descrizione

La modifica introduce un nuovo componente, `ReportGenerator`, incaricato della generazione automatica di report a partire dai risultati dell'analisi delle vulnerabilità. L'implementazione iniziale prevede la generazione di report in formato HTML, con due varianti: un template per report contenenti vulnerabilità e un template alternativo per report senza vulnerabilità rilevate.

9.2. Impact Analysis

Dall'analisi dell'impatto risulta che i componenti impattati sono i seguenti: CLIInvoker, RunCommand, run_pipeline, AppContext, ConfigManager, docker-compose.yml. è doveroso integrare le test suite relative a questi componenti, altre che crearne altre per testare ReportGenerator.

9.3. Approccio

Viene adottato un approccio **Black Box**, con derivazione dei test tramite **Category Partition Method**, partizionando gli input in classi funzionali rilevanti e valutando combinazioni valide e non valide di parametri d'ingresso.

9.4. Unit test

9.4.1 Test di Unità - HTMLReportGenerator

Questa sezione descrive i test di unità per la classe HTMLReportGenerator.

Funzionalità: Generazione del report

Parametri e categorie

Parametro	Categoria	Valori rappresentativi
`results`	Lista vuota	`[]`
	Lista ben formata	`[{ "vulnerability": ..., "analysis": ... }]`
	Lista con dati parziali	`[{ "vulnerability": ..., "analysis": None }]`
	Lista malformata	`[{ "invalid_key": ... }]`
`file_path`	Percorso valido	`"output_report/contract.txt"`
	Percorso vuoto	`""`
	Percorso non valido	`"notvalid/contract.txt"`
`report_name`	Non specificato	`None`
	Valido	`"report1"`
	Già esistente	`"existing_report"`
	Non Valido	`"<report>"`

Table 22: Categorie e valori rappresentativi per i parametri di `generate()`

Casi di test derivati

ID	`results`	`file_path`	`report_name`	**Esito atteso**
RG1	Lista ben formata	Percorso valido	Non specificato	Report HTML generato correttamente con nome basato su `file_path`
RG2	Lista vuota	Percorso valido	Non specificato	Report di tipo "nessuna vulnerabilità rilevata" generato correttamente
RG3	Lista mal formata	Percorso valido	Non specificato	Sollevamento eccezione per formato dati errato, log di errore generato
RG4	Lista con dati parziali	Percorso valido	Valido	Report generato solo con le informazioni disponibili, ignorando input incompleti
RG5	Lista valida	Percorso vuoto	Valido	Report generato correttamente, nome file usato come base per output
RG6	Lista valida	Percorso non valido	Valido	Sollevamento eccezione o fallimento scrittura; log di errore generato
RG7	Lista valida	Percorso valido	Già esistente	Report generato con suffisso numerico
RG8	Lista valida	Percorso valido	Non Valido	Errore: caratteri non validi nel nome del report

Table 23: Test case derivati

9.4.2 Test di unità - Modifiche alle Test Suite Esistenti

L'introduzione del nuovo componente `ReportGenerator` ha comportato l'aggiornamento delle test suite relative ai seguenti componenti:

CLIInvoker

È stato aggiunto un nuovo test case per verificare il corretto comportamento del sistema in assenza dell'opzione `--out`, ora obbligatoria per la generazione del report.

ID	Input	Esito atteso
C13	`run` senza opzione `--out`	Inizializzazione valida (parametro opzionale)
C14	`run` con valore di `--out` non valido	Errore: il valore di –out non deve contenere caratteri speciali

Table 24: Aggiornamento test suite CLIInvoker

RunCommand

Sono stati aggiunti due test case per valutare il comportamento del comando `run` con diversi valori del parametro `--out`.

ID	`model`	`filepath`	`vuln-limit`	`contract-limit`	`out`	Esito atteso
RC1 (modificato)	piena	piena	positivo	positivo	piena	Pipeline eseguita correttamente con generazione del report nella directory specificata
RC7	piena	piena	positivo	positivo	vuota	Pipeline eseguita correttamente con generazione del report nel percorso predefinito

Table 25: Aggiornamento test suite RunCommand

AppContext

È stato aggiornato il test di inizializzazione del contesto applicativo per verificare la corretta creazione e configurazione del componente `ReportGenerator`, mediante l'inserimento di un'asserzione esplicita.

ConfigManager

La classe di test associata al `ConfigManager` è stata aggiornata per riflettere i nuovi campi di configurazione, in particolare quello relativo alla directory di output per il report.

9.5. Integration Test

Il questa fase andiamo a modificare i test di integrazione tra i componenti per riflettere le modifiche. In particolare modifichiamo i test relativi alle interazioni tra i seguenti componenti: `CLIInvoker` → `RunCommand` → `AppContext` → `ConfigManager`, `LLMFactory`

9.5.1 Test di Integrazione - Modifiche alle Test Suite Esistenti

`CLIInvoker` → `RunCommand` → `AppContext` → `ConfigManager`, `LLMFactory`

ID	Filepath	Model	Out	Esito atteso
T5	F1 (valido)	M1 (installato)	O1 (presente)	Analisi completata con successo nel file HTML specificato
T6	F1 (valido)	M1 (installato)	O1 (non presente)	Analisi completata con successo, nome del file HTML generato automaticamente

Table 26: Modifiche test di integrazione

9.6. System Test

Modifichiamo anche la test suite di sistema per riflettere la modifica.

TC_ST_01 (modificato)	Esecuzione del comando `run` con l'aggiunta del parametro `--out` valido	File HTML generato nella directory specificata
TC_ST_12	Esecuzione del comando `run` con parametro `--out` non valido	Errore di input segnalato
TC_ST_13	Esecuzione del comando `run` senza specificare `--out`	Report salvato con nome generato automaticamente

Table 27: Test di sistema - nuovi casi introdotti

10. Change Request 3

10.1. Descrizione

La modifica prevede un redesign architetturale del sistema, tramite la separazione delle responsabilità tra frontend (strumento CLI) e backend (server API). La logica di business è stata spostata nel backend, mentre la CLI ora funge da interfaccia remota, interagendo con il server attraverso chiamate HTTP REST. Sono stati introdotti pattern architetturali come MVC e Service Layer, che hanno portato alla creazione di componenti come Controller e Service, i quali dovranno essere oggetto di test specifici.

10.2. Impact Analysis

L'analisi dell'impatto ha individuato i seguenti componenti coinvolti: SetModelCommand, RunCommand, ModelListCommand, RetrievalEngine, Dockerfile (CLI tool), Dockerfile (API server), e docker-compose.yml. Sarà quindi necessario aggiornare i test relativi a questi elementi per riflettere le modifiche introdotte.

Sebbene le classi dei comandi e la RetrievalEngine abbiano subito modifiche nella logica interna, i loro input e output sono rimasti invariati. Di conseguenza, sarà sufficiente aggiornare i test di unità e integrazione modificando esclusivamente le dipendenze esterne simulate.

10.3. Unit Test

10.3.1 Test di Unità - Run Controller

L'obiettovo del test dei controller è verificare che:

- e richieste HTTP correttamente formate ricevano risposte adeguate (es. 200 OK, 201 Created).
- le richieste errate siano gestite correttamente (400 Bad Request, 404 Not Found, ecc.).
- il controller invii correttamente i dati al livello di servizio e restituisca la risposta giusta

Parametri e categorie

Parametro	Categorie
`model`	vuota, piena
`source_code`	vuoto, corretto, non corretto
`vuln-limit`	negativo, zero, positivo
`contract-limit`	negativo, zero, positivo

Table 28: Categorie per `Run Controller`

Casi di test

ID	model	source_code	vuln_limit	contract_limit	Esito Atteso
RCo1	piena	corretto	positivo	positivo	200 OK
RCo2	vuota	corretto	positivo	positivo	422 Unprocessable Entity
RCo3	piena	vuoto	positivo	positivo	422 Unprocessable Entity
RCo4	piena	non corretto	positivo	positivo	200 OK
RCo5	piena	corretto	negativo	positivo	422 Unprocessable Entity
RCo6	piena	corretto	zero	positivo	200 OK
RCo7	piena	corretto	positivo	negativo	422 Unprocessable Entity
RCo8	piena	corretto	positivo	zero	200 OK
RCo9	vuota	vuoto	negativo	negativo	422 Unprocessable Entity

Table 29: Casi di test derivati

10.3.2 Test di Unità - SetModel Controller

Parametri e categorie

Parametro	Categorie
`model_name`	vuota, piena
`source`	valida: openai/huggingface, non valida
`api_key`	vuota, piena
`base_url`	vuota, piena

Table 30: Categorie per `SetModel Controller`

Casi di test derivati

ID	`model_name`	`source`	`api_key`	`base_url`	Esito atteso
SCo1	piena	openai	piena	piena	200 OK
SCo2	vuota	openai	piena	piena	422 Unprocessable Entity
SCo3	piena	non valida	-	-	422 Unprocessable Entity
SCo4	piena	openai	vuota	piena	422 Unprocessable Entity
SCo5	piena	huggingface	vuota	vuota	200 OK
SCo6	piena	huggingface	vuota	piena	422 Unprocessable Entity
SCo7	piena	huggingface	piena	vuota	422 Unprocessable Entity

Table 31: Casi di test

10.3.3 Test di Unità - Service

In questa fase testiamo i componenti Service adottando un approccio white-box basato sulla metodologia di branch coverage. L'obiettivo è verificare entrambi i rami logici tipicamente presenti in ogni service:

- Il ramo in cui l'elaborazione va a buon fine e viene restituito al controller il risultato corretto.
- Il ramo in cui, a causa di un errore interno alla logica di business del server, viene sollevata un'eccezione e gestita di conseguenza.

Run Service - Test Cases

ID	Descrizione	Input	Esito Atteso
Rse1	Esecuzione corretta del pipeline (ramo `success`)	`model` = "gpt-4", `source_code` = "codice valido", `vuln_limit` = 5, `contract_limit` = 2	`status` = "success", `results` = output del pipeline
Rse2	Esecuzione del pipeline che solleva un'eccezione (ramo `fail`)	`model` = "gpt-4", `source_code` = "codice non valido", `vuln_limit` = 5, `contract_limit` = 2	`status` = "fail", `results` = messaggio di errore dell'eccezione

Table 32: Casi di test per `RunService` - White Box Branch Coverage

SetModel Service - Test Cases

ID	Descrizione	Input	Esito Atteso
SSe1	Salvataggio corretto (ramo `success`)	`source` = "openai", `model_name` = "gpt-4", `base_url` = "", `api_key` = ""	`status` = "success", `results` = modello impostatto correttamente
SSe2	Esecuzione solleva un'eccezione (ramo `fail`)	`source` = "openai", `model_name` = "not-existent", `base_url` = "", `api_key` = ""	`status` = "fail", `results` = messaggio di errore dell'eccezione

Table 33: Casi di test per `SetModelService` - White Box Branch Coverage

ModelList Service - Test Cases

ID	Descrizione	Esito Atteso
MSe1	Recupero delle informazioni corretto (ramo `success`)	`status` = "success", `results` = lista dei modelli
MSe2	Esecuzione solleva un'eccezione (ramo `fail`)	`status` = "fail", `results` = messaggio di errore dell'eccezione

Table 34: Casi di test per `ModelListService` - White Box Branch Coverage