

Contents

1	Test Objectives	2
2	Pass/Fail Criteria	2
3	Tools	2
4	Esecuzione	2
5	Deliverables	3
6	Regression Testing	3
7	System Test	3
8	Change Request 1	3
8.1	Descrizione	3
8.2	Impact Analysis	3
8.3	Approccio	4
8.4	Livelli di Test	4
8.5	Unit Test	4
8.6	Integration Test	12
8.7	System Test	17

Test Plan - Change Requests

Nicola Tortora, Gaspare Galasso

July 10, 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 trova 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 verificare che le modifiche introdotte 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 sistema, oltre ad essere eseguiti in locale, verranno eseguiti anche in ambiente containerizzato attraverso 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: <code>run</code> , <code>set-model</code> , <code>model-list</code> ; non esistente
<code>--filepath</code>	presente, non presente
<code>--vuln-limit</code>	presente, non presente
<code>--contract-limit</code>	presente, non presente
<code>--model</code>	presente, non presente
<code>--model_name</code>	presente, non presente
<code>--source</code>	presente, non presente
<code>--api_key</code>	presente, non presente
<code>--base_url</code>	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 <code>RunCommand</code>
C2	run	manca <code>--filepath</code>	Errore: argomento obbligatorio mancante
C3	run	manca <code>--vuln-limit</code>	inizializzazione valida (parametri opzionali)
C4	run	manca <code>--contract-limit</code>	inizializzazione valida (parametri opzionali)
C5	run	manca <code>--model</code>	Errore: argomento obbligatorio mancante
C6	set-model	tutti presenti e validi	Comando inizializzato come <code>SetModelCommand</code>
C7	set-model	manca <code>--model_name</code>	Errore: argomento obbligatorio mancante
C8	set-model	manca <code>--source</code>	Errore: argomento obbligatorio mancante
C9	set-model	senza <code>--api_key</code> , <code>--base_url</code>	Inizializzazione valida (parametri opzionali)
C10	model-list	nessun argomento	Comando inizializzato come <code>ModelListCommand</code>
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
<code>model</code>	vuota, piena
<code>filepath</code>	vuota, piena
<code>vuln-limit</code>	negativo, zero, positivo
<code>contract-limit</code>	negativo, zero, positivo

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

Casi di test derivati

ID	<code>model</code>	<code>filepath</code>	<code>vuln-limit</code>	<code>contract-limit</code>	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
<code>model_name</code>	vuota, piena
<code>source</code>	valida: openai/huggingface, non valida
<code>api_key</code>	vuota, piena
<code>base_url</code>	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
<code>model</code>	presente nel file di config, assente nel file
<code>vuln_limit</code>	positivo, zero, negativo
<code>contract_limit</code>	positivo, zero, negativo

Table 8: Categorie per il costruttore di `AppContext`

Casi di test derivati

ID	<code>model</code>	<code>vuln_limit</code>	<code>contract_limit</code>	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 <code>vuln-limit</code> non può essere negativo
AC4	presente	positivo	negativo	Errore <code>contract-limit</code> 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 verificare 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"` \Rightarrow ritorna istanza di `OpenAILLM`.
- Se `source = "huggingface"` \Rightarrow ritorna istanza di `HFLLM`.
- Se `source` non è supportato \Rightarrow solleva `ValueError`.

Categorie del parametro config

Parametro	Categorie
<code>source</code>	openai, huggingface, non supportato
<code>model_name</code>	valido, mancante
<code>api_key</code> (per openai)	presente, mancante
<code>base_url</code> (per openai)	presente, assente
<code>device</code> (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 <code>OpenAILLM</code>
LF2	openai	manca <code>base_url</code>	Istanza di <code>OpenAILLM</code> con URL di default
LF3	openai	manca <code>api_key</code>	Errore durante inizializzazione modello
LF4	huggingface	<code>model_name</code> presente, <code>device</code> presente	Istanza di <code>HFLLM</code>
LF5	huggingface	manca <code>device</code>	Istanza con <code>device = "auto"</code>
LF6	non supportato	qualsiasi	<code>ValueError</code> 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
<code>load_config(model_name)</code> <code>add_model_config(model_config)</code>	nome presente; nome assente nella configurazione config. complete; config. mancante di campi chiave
<code>save_config(config)</code>	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	<code>load_config(model presente)</code>	Ritorna la configurazione corretta, con campo <code>llm</code> filtrato
CM2	<code>load_config(model assente)</code>	Solleva <code>ValueError</code> : modello non trovato
CM3	<code>add_model_config(model_{config}completo)</code>	Configurazione aggiornata con nuovo modello
CM4	<code>add_model_config(model_{config}mancantecampi)</code>	Modello comunque aggiunto, validazione demandata altrove
CM5	<code>save_config(config valido)</code>	File aggiornato correttamente
CM6	<code>save_config(config malformato)</code>	Potenziata 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.*

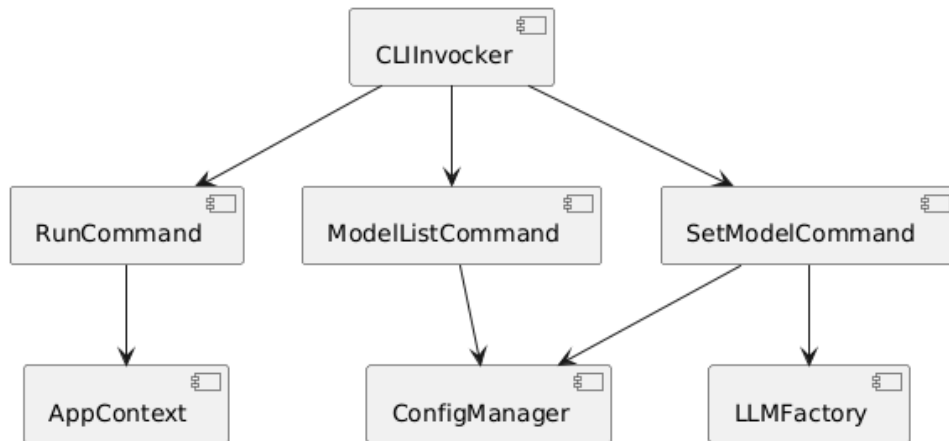


Figure 1: Relazioni di dipendenza tra le classi principali

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
<code>--source</code>	S1: openai, S2: huggingface
<code>--model_name</code>	M1: stringa valida
<code>--api_key</code>	A1: presente, A2: assente (richiesto solo se <code>source=openai</code>)
<code>--base_url</code>	B1: stringa valida (richiesta solo se <code>source=openai</code>)

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: <code>--api_key</code> obbligatorio per OpenAI
T3	openai	valido	presente	assente	Errore: <code>--base_url</code> 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 <code>api_key</code> e <code>base_url</code> ignorati

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

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

Questa sezione descrive i test di integrazione per il comando `model-list`, che coinvolge l'interazione tra `CLIInvoker`, `ModellListCommand` 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 `ModellListCommand`. 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` → `ModellListCommand` → `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 <code>model-list</code>	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 <code>run</code> 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