UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Ingegneria del Software: Tecniche Avanzate

Progetto ISTA – Evoluzione di CodeSmile

Master Test Plan

Professore:
Ch.mo Prof.
Andrea De Lucia

Tutor di riferimento:
Giammaria Giordano
Gilberto Recupito

Studenti:
Antonio D'Auria
Mat. NF22500063
Emanuele D'Auria
Mat. 0522501918
Luca Casillo
Mat. NF22500037

ANNO ACCADEMICO 2025/2026

Indice

1 Introduzione

Il presente documento definisce il *Master Test Plan* del progetto, fornendo una visione complessiva delle attività di testing previste e del loro inquadramento rispetto alle Change Request analizzate.

Il documento ha lo scopo di consolidare, in un unico riferimento, i criteri organizzativi e decisionali che guidano le attività di verifica, senza reintrodurre nel dettaglio i test già eseguiti o gli incidenti rilevati, ampiamente documentati nei precedenti elaborati.

In particolare, il Master Test Plan:

- definisce la strategia generale di testing adottata;
- delimita il perimetro di verifica rilevante rispetto alle Change Request;
- chiarisce la copertura dei test rispetto alle componenti del sistema;
- stabilisce i criteri di accettazione delle modifiche proposte.

Le sezioni successive dettagliano tali aspetti in modo coerente con le attività di testing già svolte e con gli obiettivi di validazione del progetto.

2 Strategia di Testing

La strategia di testing adottata nel progetto è stata definita con l'obiettivo di garantire una validazione sistematica e ripetibile del sistema, tenendo conto della natura eterogenea dell'architettura (strumento di analisi statica, servizi backend, applicazione web) e delle diverse modalità di interazione tra le componenti.

Questa sezione non introduce nuove strategie di verifica, ma consolida e formalizza le attività di testing già svolte e documentate nei precedenti elaborati, in particolare nel documento di Pre-Modification System Testing. e nel Pre-Modification Test Incident Report, fornendone una visione unificata e coerente.

2.1 Approccio Generale

La strategia di testing adottata nel progetto si basa su una strutturazione progressiva delle attività di verifica.

L'esecuzione dei test è stata organizzata in modo incrementale, partendo dalla verifica delle singole unità software fino alla validazione dei flussi utente end-to-end, secondo una progressione che consente di individuare precocemente difetti locali e di valutare successivamente il comportamento complessivo del sistema.

2.2 Livelli di Testing

La strategia prevede i seguenti livelli di test:

- **Unit Testing**: verifica del comportamento delle singole unità software in isolamento, con particolare attenzione ai casi limite e alla gestione delle condizioni di errore;
- **Integration Testing**: verifica delle interazioni tra moduli e servizi cooperanti, focalizzata sul rispetto dei contratti delle API e sulla coerenza delle strutture dati scambiate;
- **System Testing**: validazione del comportamento del sistema nel suo complesso, utilizzando casi di test strutturati e scenari realistici di utilizzo;
- **End-to-End Testing**: verifica dei principali flussi utente dell'applicazione web, simulando l'interazione completa tra frontend e backend.

Ogni livello di testing è stato applicato in modo mirato alle diverse componenti del sistema, come descritto nelle sezioni seguenti.

2.3 Strategia per lo Static Analysis Tool

Per lo *Static Analysis Tool*, la strategia di testing ha previsto:

- **Unit Test**: volti a verificare il corretto funzionamento dei moduli di analisi, parsing e rilevazione dei code smell;
- **Integration Test**: orientati alla collaborazione tra i moduli del core di analisi;
- **System Test**: eseguiti tramite CLI, basati su directory di input e progetti strutturati, al fine di validare il comportamento complessivo dello strumento in scenari realistici.

I test di sistema sono stati automatizzati mediante script dedicati, consentendo l'esecuzione ripetibile dei casi di test e la raccolta sistematica dei risultati in formato strutturato.

2.4 Strategia per la Web Application

Per la Web Application, la strategia di testing ha adottato un approccio complementare, distinguendo chiaramente tra frontend e backend:

- **Unit Test**: eseguiti con Jest, mirati a verificare il corretto rendering e comportamento dei componenti UI in isolamento;
- **Integration Test**: eseguiti con PyTest, focalizzati sull'interazione tra gateway e servizi backend, con particolare attenzione al rispetto dei contratti delle API;
- **End-to-End Test**: eseguiti con Cypress in modalità headless, finalizzati a validare i principali flussi utente dell'applicazione web.

I test end-to-end sono stati progettati per osservare il comportamento del sistema dal punto di vista dell'utente finale, includendo sia scenari di successo sia condizioni di errore, e sono stati utilizzati anche per individuare problematiche di flakiness e sincronizzazione asincrona.

2.5 Perimetro di Testing

La strategia di testing adottata considera esclusivamente le componenti e i comportamenti direttamente coinvolti dalle Change Request proposte.

In particolare, le funzionalità basate su servizi di analisi AI esterni non rientrano nel perimetro di verifica del presente piano di test. I relativi test, quando presenti, sono stati esclusi dalle attività di validazione in quanto dipendenti da servizi infrastrutturali esterni e non impattati dalle modifiche oggetto delle Change Request.

Di conseguenza, eventuali fallimenti associati a tali componenti non sono stati trattati come fault applicativi né oggetto di intervento correttivo.

3 Coverage

La strategia di copertura adottata ha l'obiettivo di monitorare e preservare il livello di qualità del codice esistente, garantendo che le modifiche introdotte tramite le Change Request non comportino una regressione misurabile rispetto allo stato iniziale del sistema.

La copertura è stata misurata utilizzando lo strumento `coverage.py`, applicato a tutte le componenti testate mediante test automatici in Python. Dal calcolo delle metriche sono escluse le componenti non soggette a misurazione per motivi strutturali o progettuali, come dettagliato nelle sezioni seguenti.

L'indicatore di riferimento adottato è la **Branch Coverage**, per valutare la robustezza del comportamento del codice in presenza di ramificazioni logiche.

3.1 Static Analysis Tool

Per lo Static Analysis Tool, la copertura è stata misurata sui test unitari e di integrazione, con particolare attenzione ai package architetturali critici.

- **Unit Test**: l'esecuzione dei test unitari produce una **branch coverage complessiva pari all'85%**, con valori elevati sui moduli fondamentali del sistema.
  In particolare, i package `components`, `detection_rules`, `code_extractor` e `utils` presentano coperture individuali comprese tra l'**82% e il 100%**, garantendo un'ampia esercitazione della logica di analisi, parsing e rilevazione dei code smell.
- **Integration Test**: i test di integrazione raggiungono una **branch coverage complessiva pari al 34%**. Tale valore, inferiore rispetto ai test unitari, è considerato coerente con la finalità del livello di test, orientato alla validazione delle interazioni tra moduli e non all'esercizio esaustivo dei rami di codice.

I **System Test** dello Static Analysis Tool non sono inclusi nel calcolo della coverage. A causa della loro natura black-box e dell'esecuzione tramite CLI su directory di input, tali test validano il comportamento osservabile del sistema ma non consentono una tracciabilità affidabile dell'esecuzione interna del codice ai fini della misurazione della copertura.

3.2 Web Application

Per la Web Application, la copertura è stata misurata esclusivamente sui test di integrazione implementati in Python, che verificano l'interazione tra il gateway e i servizi backend.

- **Unit Test**: i test unitari del frontend, eseguiti con Jest, non sono inclusi nel calcolo della coverage. La misurazione della copertura JavaScript/TypeScript non rientra negli obiettivi del progetto e non è stata prevista nella configurazione originale.
- **Integration Test**: i test presenti in `webapp/integration_tests` sono stati eseguiti sotto `coverage.py`. A causa della modalità di esecuzione tramite client di test HTTP e dell'isolamento applicativo del gateway, tali test non producono una copertura significativa del codice Python misurabile a livello di branch coverage. L'assenza di dati di coverage è quindi attribuibile a limitazioni tecniche dello strumento e conferma che tale aspetto non rientra negli obiettivi del progetto.

Analogamente, i **test End-to-End** eseguiti con Cypress non contribuiscono alle metriche di coverage. Tali test sono orientati alla validazione dei flussi utente e del comportamento

funzionale complessivo dell'applicazione, e non forniscono informazioni significative in termini di copertura del codice sorgente Python o frontend.

4 Criteri di Accettazione

Le Change Request sono considerate accettate se e solo se sono soddisfatti tutti i seguenti criteri:

(1) **Preservazione della Coverage**: l'applicazione delle Change Request non deve comportare una riduzione delle metriche di *branch coverage* rispetto ai valori di riferimento documentati nello stato pre-modifica.
(2) **Assenza di Bug Critici**: non devono essere presenti bug critici o bloccanti che compromettano l'esecuzione dei test unitari, di integrazione o di sistema sulle componenti coinvolte dalle Change Request.
(3) **Allineamento ai Contratti Esistenti**: le modifiche non devono introdurre regressioni nei contratti delle API o nel comportamento osservabile del sistema rispetto alle specifiche già validate.
(4) **Assenza di Regressioni Funzionali**: i flussi funzionali già validati tramite system test ed end-to-end test devono continuare a produrre esito positivo.