
UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

REPORT DI PROGETTO

# Ingegneria del Software: Tecniche Avanzate - Report del Progetto

Luigi Auriemma

Matricola: NF22500161

Ivan Chiello

Matricola: NF22500167

Repository: `https://github.com/LuigiAuriemma/smell_ai.git`

Anno Accademico 2025-2026


sesa
lab
SOFTWARE ENGINEERING
SALERNO

# Indice

# Elenco delle figure

CAPITOLO 1

# Descrizione del Sistema

## 1.1 Contesto Applicativo

Lo sviluppo del software ha subito una trasformazione sostanziale con l'integrazione sempre più frequente di componenti di IA all'interno dei sistemi tradizionali, dando vita ai ML-enabled systems.
Queste applicazioni si distinguono per la loro complessità, dovuta alla interdipendenza tra diverse componenti.
In questo contesto le necessità di ridurre i tempi di rilascio spingono spesso gli sviluppatori ad adottare implementazioni rapide e non sempre perfette. Sebbene tali scelte accelerano lo sviluppo del sistema nel breve periodo, contribuiscono alla creazione di Technical Debt (nel contesto dell'IA prende il nome di AI Technical Debt).
Una delle manifestazioni più concrete di questo debito è rappresentata dai Code Smells, ovvero sintomi di scelte progettuali inadeguate che possono deteriorare la qualità del sistema nel tempo. Per quanto riguarda le pipeline di Machine Learning, sono emersi nuovi pattern di cattiva programmazione soprannominati come Machine Learning Specific Code Smells (ML-CSs).
Un ML-CS è definito come una soluzione implementativa sub-ottimale all'interno di una pipeline di ML che può degradare la qualità del sistema.

CodeSmile nasce in questo contesto come uno strumento mirato a rilevare queste problematiche per prevenire il degrado dei sistemi basati su ML.

## 1.2 Lo Strumento CodeSmile

CodeSmile è uno strumento software di analisi statica appositamente progettato per individuare ML-CSs all'interno di ML-enabled Systems.
Nello specifico, CodeSmile è stato sviluppato per operare su progetti scritti in linguaggio Python, analizzando l'utilizzo delle librerie più diffuse nell'ambito del ML. Attraverso l'analisi statica del codice con l'utilizzo di Abstract Syntax Trees (AST) e all'adozione di un approccio rule-based (basato su regole), lo strumento è in grado di identificare automaticamente un set di 16 ML-CSs che impattano negativamente su manutenibilità, efficienza e robustezza del modello.
Questi smell sono suddivisi in due macro-categorie principali:

- **API-Specific Smells**: legati all'uso impreciso delle librerie ML.
- **Generic Smells**: pattern inefficienti di natura più generale.

### 1.2.1 Analisi del Funzionamento

L'analisi statica effettuata da CodeSmile si basa sull'esplorazione della struttura del contenuto del progetto in analisi. Il funzionamento può essere descritto attraverso i seguenti passaggi:

- Costruzione dell'AST: Il tool effettua il parsing dei file Python trasformando il codice sorgente in un AST. Questa rappresentazione permette di navigare tra i nodi del codice (chiamate a metodi, assegnazioni, ecc.) in modo strutturato.
- Inferenza dei Tipi e Associazione Librerie: identifica le librerie importate e traccia le variabili istanziate per determinare se appartengono a contesti ML.
- Applicazione delle Regole: Il sistema verifica la presenza dei code smells confrontando l'AST con le regole predefinite.

# CAPITOLO 2

# Struttura del Sistema

L'architettura di CodeSmile è modulare e organizzata in package distinti, ognuno preposto a una specifica responsabilità nel ciclo di vita dell'analisi, dall'interazione utente all'estrazione delle metriche. Di seguito viene presentata l'analisi dettagliata dei moduli principali.

## 2.1 Package Principali del Sistema

### 2.1.1 cli

L'obiettivo è fornire un punto di accesso da terminale per eseguire il processo di analisi dei progetti.
Moduli principali:

- cli_runner.py: Ospita la classe CodeSmileCLI e l'entry point main (per istanziare la CLI). Questo modulo è responsabile della validazione degli argomenti forniti dall'utente e del controllo del flusso di esecuzione. In base alla configurazione scelta, invoca i metodi del ProjectAnalyzer per avviare l'analisi (in modalità sequenziale o parallela) e gestisce l'aggregazione finale dei risultati.

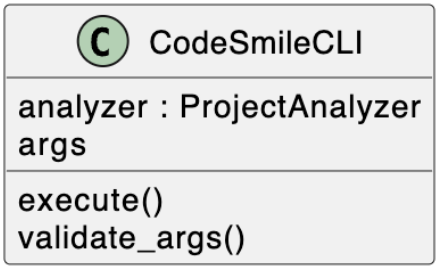


**Figura 2.1:** Analisi modulo cli.

### 2.1.2 code_extractor

Il suo obiettivo è identificare, tracciare e catalogare gli elementi sintattici rilevanti (come librerie, variabili e strutture dati) necessari per la fase di detection.
Moduli principali:

- library_extractor.py: responsabile della mappatura degli import. Analizza i nodi ast.Import e ast.ImportFrom per costruire un dizionario che associa ogni libreria al suo alias nel codice (es. mappa pd a pandas). Offre inoltre metodi per risalire alla libreria di appartenenza di una specifica chiamata di funzione all'interno dell'AST.

- variable_extractor.py: si occupa del tracciamento delle variabili. Identifica i punti di definizione (assegnamenti) e traccia le occorrenze di utilizzo.

- dataframe_extractor.py: Specializzato nell'analisi della libreria Pandas, utilizza liste di metodi noti (caricati da file di configurazione) per riconoscere le variabili che rappresentano DataFrame e tracciare le operazioni effettuate su di esse, come l'accesso alle colonne o l'invocazione di metodi specifici.

- model_extractor.py: Gestisce la knowledge base relativa ai modelli di Machine Learning e alle operazioni tensoriali. La classe ModelExtractor carica definizioni da file CSV esterni per identificare metodi e proprietà specifici dei modelli ML. Questo permette al sistema di distinguere tra operazioni generiche e quelle legate al training o all'inferenza dei modelli.

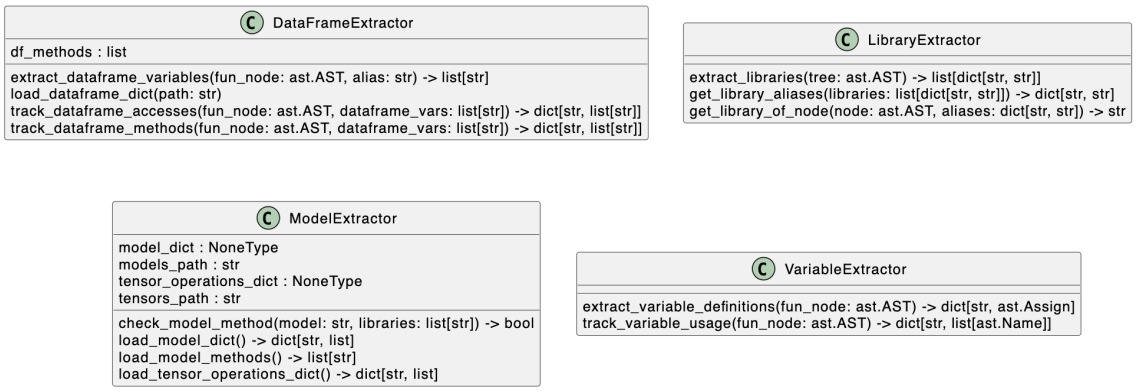


**Figura 2.2:** Analisi modulo code_extractor.

### 2.1.3 components

Questo package rappresenta il nucleo operativo (core logic) di CodeSmile. Qui risiedono i componenti che orchestrano il processo di analisi, dalla gestione del download dei progetti fino all'applicazione delle regole di detection sul singolo file. Moduli principali:

- project_repository_cloner.py: Modulo dedicato all'acquisizione dei dati. La classe ProjectRepositoryCloner gestisce il download automatico dei repository da GitHub. Partendo da un dataset CSV di input, filtra i progetti in base a criteri predefiniti e li clona localmente per prepararli all'analisi, gestendo anche la creazione e la pulizia delle directory di lavoro.
- project_analyzer.py: Agisce come motore di orchestrazione ad alto livello tramite la classe ProjectAnalyzer. Le sue responsabilità includono:
  - La gestione del parallelismo: utilizza ThreadPoolExecutor per distribuire l'analisi dei progetti su più thread (walkers), ottimizzando i tempi di esecuzione.
  - La gestione dell'I/O: coordina la pulizia delle cartelle di output e il salvataggio dei risultati, sia parziali (per singolo progetto) che aggregati (merge_all_results).
- inspector.py: La classe Inspector è responsabile dell'analisi puntuale del singolo file sorgente. Funge da "ponte" tra il codice grezzo e la logica di detection:

converte il file in AST e inizializza tutti gli estrattori necessari (Library, Model, DataFrame, Variable) per costruire il contesto semantico prima di invocare il controllo delle regole.

- rule_checker.py: Contiene la logica decisionale del sistema. La classe RuleChecker mantiene il registro di tutti gli smell supportati (sia API-Specific che Generic). Riceve dall'Inspector il nodo AST e i metadati estratti, iterando su ogni regola configurata per verificare la presenza di violazioni e restituendo un report dettagliato degli smell individuati.

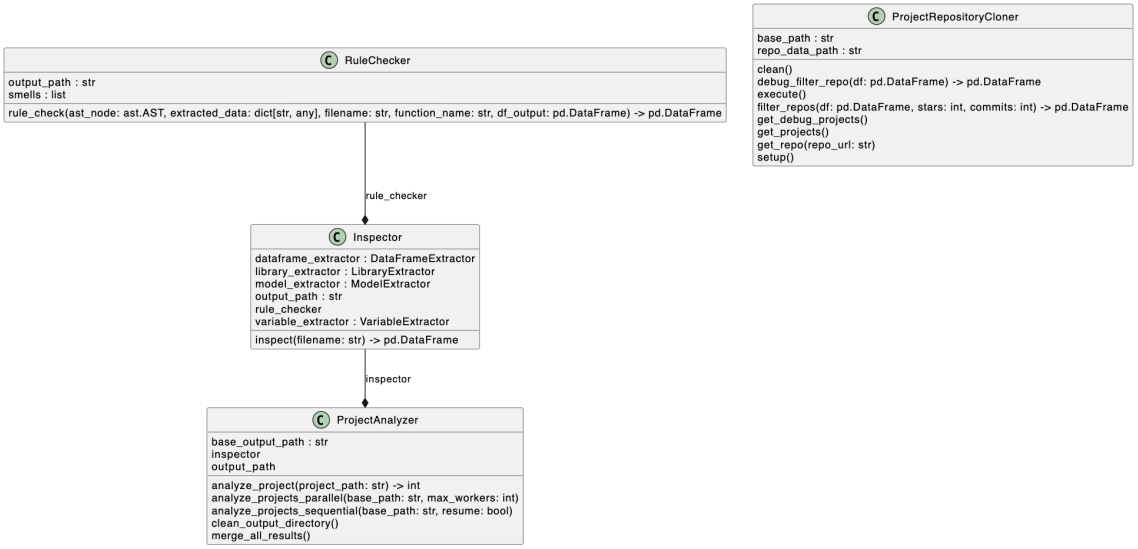


**Figura 2.3:** Analisi modulo components.

### 2.1.4 detection_rules

Questo package costituisce il catalogo delle regole di qualità. Contiene l'implementazione concreta degli algoritmi necessari per individuare ciascuno dei Machine Learning Specific Code Smells (ML-CSs). L'architettura del package è progettata per essere estensibile, permettendo l'aggiunta di nuove regole senza modificare smell.py.

- smell.py: Definisce l'interfaccia comune per tutte le regole tramite la classe Smell. Questa classe impone un contratto rigoroso: ogni regola deve implementare il metodo detect(ast_node, extracted_data). Il metodo detect riceve in input l'albero sintattico (AST) e i dati contestuali estratti (librerie, variabili,

definizioni). Restituisce una lista contenenti i dettagli delle violazioni (riga, messaggio, descrizione).

**api_specific**

Questo modulo raggruppa le regole che identificano violazioni legate all'uso scorretto dalle librerie di ML (Pandas, TensorFlow, PyTorch, NumPy).

- chain_indexing_smell.py (Pandas): Rileva il pattern di indicizzazione concatenata.
- gradients_not_cleared_before_backward_propagation.py (PyTorch): Analizza i cicli per verificare se la chiamata a backward() è preceduta da zero_grad(). La regola traccia le chiamate ai metodi all'interno dello scope del loop per prevenire l'accumulo errato dei gradienti.
- pytorch_call_method_misused.py (PyTorch): Identifica le invocazioni esplicite del metodo .forward() sui modelli. La regola suggerisce di utilizzare l'invocazione diretta dell'istanza per garantire l'esecuzione corretta.
- matrix_multiplication_api_misused.py (NumPy): Cerca l'utilizzo della funzione dot() o del metodo .dot per le moltiplicazioni matriciali, deprecata in certi contesti a favore di matmul per una gestione più chiara delle dimensioni.
- tensor_array_not_used.py (TensorFlow): Verifica se costanti vengono modificate o accresciuti dinamicamente all'interno di cicli, suggerendo la conversione verso strutture dati ottimizzate come TensorArray.
- dataframe_conversion_api_misused.py (Pandas): Segnala l'uso dell'attributo values, considerato ambiguo.

**generic**

Queste regole mirano a identificare pattern di codice che, pur essendo sintatticamente corretti, violano le best practices della Data Science, introducendo inefficienze computazionali o rischi di instabilità.

- broadcasting_feature_not_used.py: rileva l'uso della funzione tf.tile per replicare tensori prima di operazioni aritmetiche e suggerisce di sfruttare il broadcasting implicito di TensorFlow.
- columns_and_datatype_not_explicitly_set.py: Controlla le chiamate a pd.read_csv o pd.DataFrame. Se non vengono specificati i parametri dtype o i nomi delle colonne c'è il rischio di inferenza automatica dei tipi errata.
- deterministic_algorithm_option_not_used.py: identifica l'attivazione di torch.use_deterministic_algorithms(True). Questa opzione degrada notevolmente le prestazioni in produzione.
- empty_column_misinitialization.py: rileva l'inizializzazione di intere colonne di DataFrame con 0 o stringhe vuote "". La regola suggerisce l'uso di np.nan.
- hyperparameters_not_explicitly_set.py: analizza le istanziazioni dei modelli. Se il costruttore viene chiamato senza argomenti, viene segnalata la mancanza di iperparametri espliciti.
- in_place_apis_misused.py: Gestisce l'ambiguità delle operazioni in-place. Segnala come errore i casi in cui metodi vengono chiamati senza assegnare il risultato a una variabile e senza impostare esplicitamente inplace=True, il che renderebbe l'istruzione priva di effetto.
- memory_not_freed.py: ispeziona i cicli in cui vengono istanziati nuovi modelli. Se non rileva una chiamata esplicita a tf.keras.backend.clear_session(), segnala il rischio di memory leak.
- merge_api_parameter_not_explicitly_set.py: verifica che le operazioni di merge specifichino chiaramente i parametri on, how e validate, per evitare comportamenti errati.
- nan_equivalence_comparison_misused.py: verifica la presenza di confronti diretti di uguaglianza o disuguaglianza con NaN e suggerisce l'uso delle funzioni dedicate isnan().

- unnecessary_iteration.py: analizza i cicli (For, While) per individuare iterazioni inefficienti su DataFrame (es. iterrows, itertuples).


**Figura 2.4:** Analisi modulo detection_rules.

### 2.1.5 gui

Questo package implementa l'interfaccia grafica utente (GUI). Il suo obiettivo è fornire un'alternativa visuale alla riga di comando.

- gui_runner.py: funge da launcher per l'applicazione grafica e quindi si occupa dell'inizializzazione della finestra principale e permette di avviare il ciclo degli eventi principale (mainloop).
- code_smell_detector_gui.py: Contiene la logica di presentazione e interazione tramite la classe CodeSmellDetectorGUI. Permette di costruire i widget per la selezione delle directory e per la configurazione dei parametri di esecuzione. Il metodo run_analysis avvia l'analizzatore (ProjectAnalyzer) su un thread separato rispetto a quello della GUI.
- textbox_redirect.py: intercetta lo stream di output standard e lo reindirizza verso un widget testuale all'interno della finestra dell'applicazione.

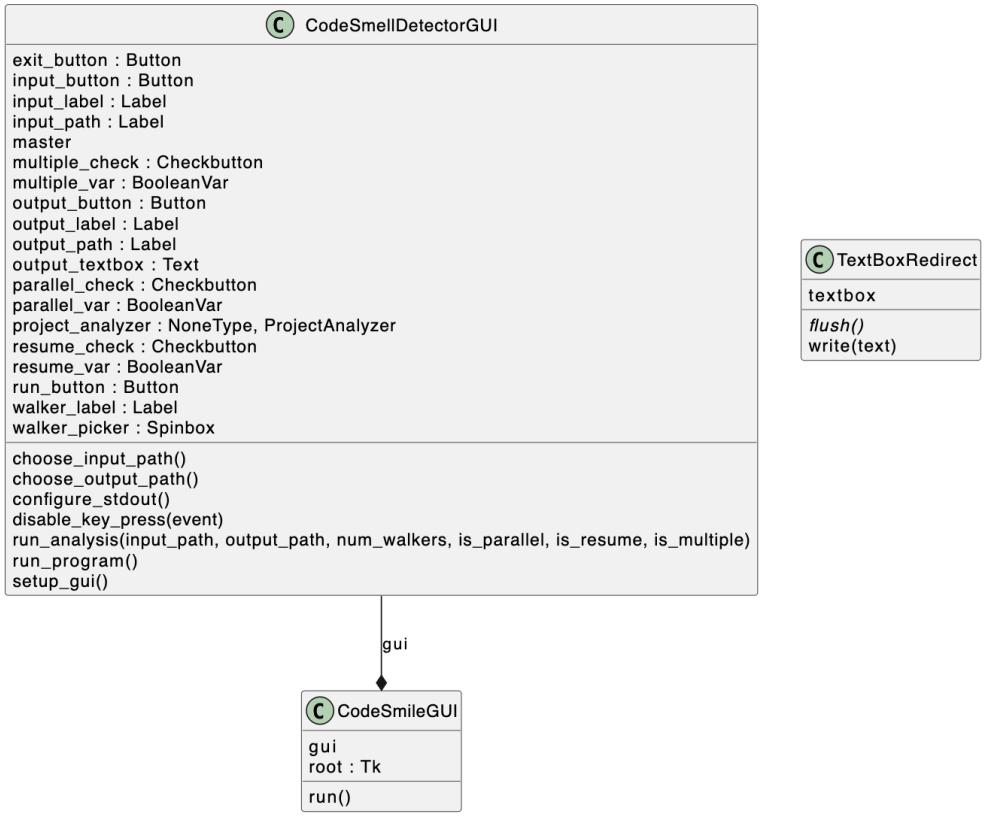


**Figura 2.5:** Analisi modulo gui.

### 2.1.6 report

Gestisce l'aggregazione e la visualizzazione dei risultati post-analisi.

- report_generator.py: responsabile della visualizzazione dei CSV ottenuti in output. Il modulo calcola e mostra statistiche offrendo sia riepiloghi testuali che grafici. Include inoltre un'interfaccia a menu per permettere all'utente di selezionare dinamicamente il tipo di report o visualizzazione desiderata.

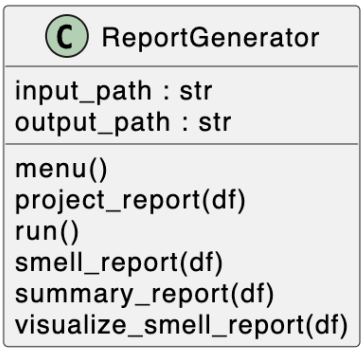


**Figura 2.6:** Analisi modulo report.

### 2.1.7 obj_dictionaries

Questa cartella fornisce una base di conoscenza per il sistema andando a conservare dei file CSV per mappare dinamicamente metodi e classi appartenenti alle librerie supportate, facilitando l'estensione del tool senza necessità di ricompilazione.

- dataframes.csv: contiene un elenco esaustivo dei metodi di Pandas e i relativi tipi di ritorno. Viene utilizzato dal DataFrameExtractor per riconoscere le operazioni che generano o trasformano DataFrame.
- models.csv: cataloga i costruttori dei modelli di Machine Learning supportati, permettendo al ModelExtractor di identificare l'istanziazione dei modelli di ML.
- tensors.csv: definisce le operazioni specifiche sui tensori e i loro metadati.

### 2.1.8 utils

Raggruppa le funzioni di utilità trasversali per la gestione dell'I/O.

- file_utils.py: Contiene la classe FileUtils che centralizza operazioni comuni come la scansione ricorsiva dei file .py, la gestione e pulizia delle directory di output e la scrittura sincronizzata dei log di esecuzione.

### 2.1.9 data_preparation

Gestisce l'intera pipeline di creazione del dataset, dal mining dei dati alla generazione sintetica.

- Gestione del processo: dataset_creation_runner.py gestisce l'intero processo di creazione, analisi e preparazione dei dataset per il progetto; base_llm.py permette al resto dell'applicazione di scambiare facilmente diversi modelli LLM senza dover modificare il codice che li utilizza; qwen_llm.py è l'implementazione concreta del LLM da utilizzare nel progetto.
- Scraping e Iniezione: repository_downloader.py e function_dataset_builder.py acquisiscono codice; injected_smells_dataset_builder.py utilizza code_smell_injector.py per generare varianti affette da smell.
- Validazione: balanced_dataset_builder.py unifica i dati puliti e iniettati, mentre code_smell_analyzer.py e dataset_evaluator.py verificano la qualità e la distribuzione del dataset finale.

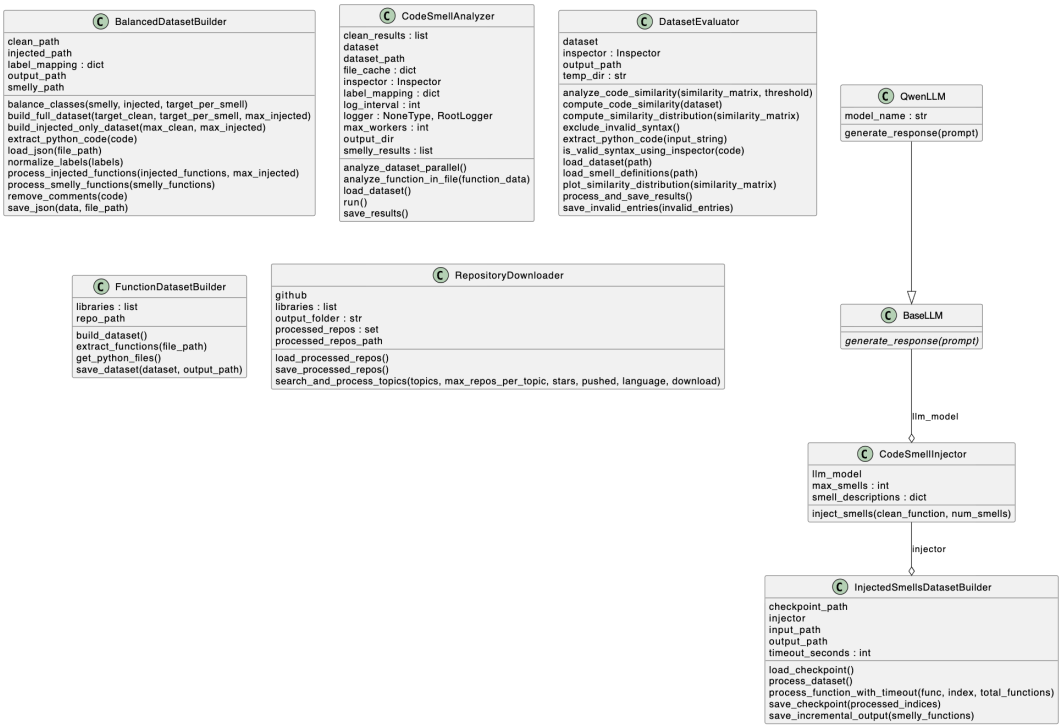


**Figura 2.7:** Analisi modulo data_preparation.

### 2.1.10 finetuning

#### train

Questo modulo gestisce l'intero ciclo di vita del processo di addestramento del Large Language Model, con l'obiettivo di specializzarlo nel riconoscimento dei code smells. L'architettura del package è progettata per orchestrare il caricamento e l'ottimizzazione del modello tramite LoRA. Parallelamente, si occupa della preparazione e formattazione dei dataset e della configurazione granulare degli iperparametri di training, garantendo l'esecuzione supervisionata e il salvataggio finale del modello ottimizzato pronto per l'inferenza.

#### validation

Questo modulo è dedicato alla valutazione delle performance del modello post-addestramento. L'obiettivo è misurare l'efficacia del Large Language Model nel riconoscere correttamente i code smells su dati non visti durante il training. Il sistema gestisce il caricamento del modello fine-tunato ottimizzandolo per l'inferenza.

Attraverso una pipeline di testing, sottopone al modello snippet di codice del dataset di validazione e ne raccoglie le risposte generative. Un componente essenziale di questo modulo è il parser semantico, che traduce l'output testuale naturale del LLM in etichette di classificazione strutturate. Infine, il modulo confronta queste predizioni, calcolando e riportando metriche statistiche come accuracy, precision, recall e F1-score.


**Figura 2.8:** Analisi modulo finetuning.

### 2.1.11 test

#### unit_testing

Questo modulo si concentra sulla verifica granulare e isolata dei singoli blocchi funzionali del codice. La struttura delle directory ricalca fedelmente quella del codice sorgente (components, detection_rules, code_extractor, etc.), permettendo test mirati per ogni componente. L'obiettivo qui è validare la logica interna pura e la correttezza, astraendo completamente le dipendenze esterne tramite l'uso di Mock. Questo approccio garantisce che ogni componente del sistema sia corretto prima di considerare il flusso di esecuzione completo.

#### integration_testing

Questo modulo è dedicato alla verifica della corretta interoperabilità tra i diversi componenti architetturali del sistema. A differenza dei test unitari isolati, l'obiettivo qui è validare il flusso di esecuzione e le interfacce di comunicazione tra i moduli principali. Le suite di test coprono l'intera catena di elaborazione, assicurando che i comandi impartiti tramite le interfacce utente (sia CLI che GUI) vengano correttamente propagati al motore di analisi centrale. Successivamente, viene verificata la catena di deleghe che porta dall'orchestrazione del progetto fino all'ispezione del singolo file e all'attivazione delle regole di detection. Attraverso l'uso di mock per isolare le dipendenze esterne, il sistema valida che i dati (configurazioni, AST e metadati) siano trasmessi coerentemente tra i vari livelli logici, garantendo che l'integrazione tra front-end, controller e logica di business avvenga senza errori.

#### system_testing

La cartella è organizzata in Test Cases (TC) numerati che contengono specifici "Code Smells" noti, mentre altri rappresentano codice senza smells. Il sistema viene testato, seguendo i principi del Black-Box Testing, contro questi input per verificare che i meccanismi di detection identifichino correttamente gli errori presenti e che l'intero processo, dal caricamento del progetto alla generazione del report, produca l'output atteso per quel determinato scenario d'uso.

### 2.1.12 webapp

#### app

Questa cartella rappresenta il "core" dell'applicazione frontend basata su Next.js. Qui sono definiti il layout e gli stili global ed anche le pagine pubbliche accessibili dall'utente tramite browser.

#### components

Contiene i componenti React riutilizzabili che compongono l'interfaccia utente (UI). Questo approccio modulare facilita la manutenzione e la coerenza visiva.

**context**

Gestisce lo stato globale dell'applicazione frontend utilizzando le React Context API. Permette di condividere,aggiornare e memorizzare dati e stati complessi .

**tests**

Contiene i test unitari e di integrazione per il frontend. Verifica il corretto rendering e comportamento dei componenti e delle pagine, garantendo che l'interfaccia risponda correttamente alle azioni dell'utente.

**cypress**

Dedicata al testing End-to-End (E2E) automatizzato tramite Cypress. Contiene gli scenari di test che simulano l'interazione di un utente reale con l'applicazione completa (navigazione, upload di file, visualizzazione report).

**gateway**

Rappresenta il punto di ingresso (API Gateway) per l'analisi tramite LLM, statica e report. Agisce come intermediario tra il frontend e i vari microservizi interni e gestisce l'instradamento delle richieste e l'aggregazione delle risposte.

**services**

Contiene la logica di business del backend suddivisa in microservizi distinti, ognuno isolato e containerizzabile.

- aiservice: Servizio dedicato alle funzionalità di Intelligenza Artificiale. Utilizza un modello LLM per analizzare snippet di codice e rilevare "code smells". Espone l'endpoint 'detect_smell_ai'.
- staticanalysis: Servizio che si occupa dell'analisi statica del progetto. Salva temporaneamente gli snippet di codice ricevuti, esegue le regole di detection e restituisce i risultati. Espone l'endpoint 'detect_smell_static'.

- report: Servizio responsabile dell'aggregazione e della post-elaborazione dei dati. Riceve i risultati delle analisi da più progetti e, utilizzando librerie come Pandas, compila statistiche e strutture dati ottimizzate per la visualizzazione grafica nel frontend. Espone l'endpoint 'generate_report'.

**types**

Definisce le interfacce e i tipi di dati utilizzati in tutta l'applicazione frontend.

**utils**

Raccoglie funzioni di utilità, helper e configurazioni condivise e contiene wrapper e funzioni per gestire le chiamate HTTP.

**integration_tests**

Contiene suite di test di integrazione specifiche per il backend. Verifica la corretta comunicazione e il flusso dati tra il Gateway e i vari microservizi assicurando che l'architettura a servizi funzioni come un sistema coeso.

# CAPITOLO 3

# Flusso di Funzionamento del Sistema

## 3.1 Tipologie di Utilizzo

CodeSmile è stato progettato per garantire flessibilità d'uso e integrazione in diversi contesti operativi. Il sistema offre tre modalità principali di interazione: tramite riga di comando, interfaccia grafica desktop e via applicazione web. Di seguito vengono analizzati i flussi di esecuzione per ciascuna modalità.

- Utilizzo tramite CLI (Command Line Interface): la modalità a riga di comando è gestita dal package cli, con il modulo cli_runner.py che funge da punto di ingresso. Questa interfaccia è ideale per l'integrazione in pipeline di CI/CD.
- Utilizzo tramite GUI (Graphical User Interface): l'interfaccia grafica desktop, implementata con la libreria Tkinter nel package gui, offre un approccio user-friendly per gli utenti che preferiscono un'interazione visiva.
- Utilizzo tramite Web App: L'applicazione web rappresenta l'evoluzione del sistema verso un'architettura a microservizi, rendendo CodeSmile accessibile direttamente da browser.

- Nel caso dell'Analisi Statica, il servizio backend crea un ambiente temporaneo, esegue l'Inspector sul codice caricato e restituisce la lista degli smell rilevati.
- Nel caso dell'Analisi AI, il codice viene inviato al modello LLM che genera una risposta in linguaggio naturale, successivamente parsata per estrarre le etichette degli errori.
- Visualizzazione Risultati: Il frontend riceve la risposta strutturata e aggiorna la pagina, mostrando all'utente una dashboard con i Code Smells individuati, la loro posizione nel codice e suggerimenti per il refactoring.

## 3.2 Flusso di Utilizzo

In questa sezione vengono dettagliati i flussi operativi che caratterizzano le tre modalità di interazione con il sistema CodeSmile. Per ogni interfaccia, si descrive la sequenza di eventi che porta dall'input dell'utente alla generazione del risultato finale.

### 3.2.1 Flusso di Utilizzo tramite CLI (Command Line Interface)

1. L'utente lancia lo script `cli_runner.py` specificando i parametri obbligatori `-input` e `-output`. Opzionalmente, può attivare flag come `-parallel` o `-resume`.

2. Il componente `CodeSmileCLI` riceve gli argomenti, verifica l'esistenza delle cartelle e la coerenza dei parametri (es. il numero di thread walkers deve essere positivo).

3. Viene istanziato il `ProjectAnalyzer`. Se è attiva la modalità `-multiple`, il sistema itera su tutte le sottocartelle trovate; altrimenti, si focalizza sulla singola directory target.

4. **Elaborazione (Sequenziale o Parallela):**

- In modalità sequenziale, i file vengono analizzati linearmente all'interno del ciclo principale (come dettagliato nel diagramma di sequenza allegato).
- In modalità parallela, viene configurato un `ThreadPoolExecutor` che distribuisce i file tra i worker disponibili. Ogni worker esegue la medesima logica di ispezione mostrata nel diagramma per i file assegnati.

5. Al termine dell'elaborazione, i risultati vengono consolidati e scritti in file CSV nella cartella di destinazione (`_save_results`), e il processo termina restituendo il controllo al terminale.

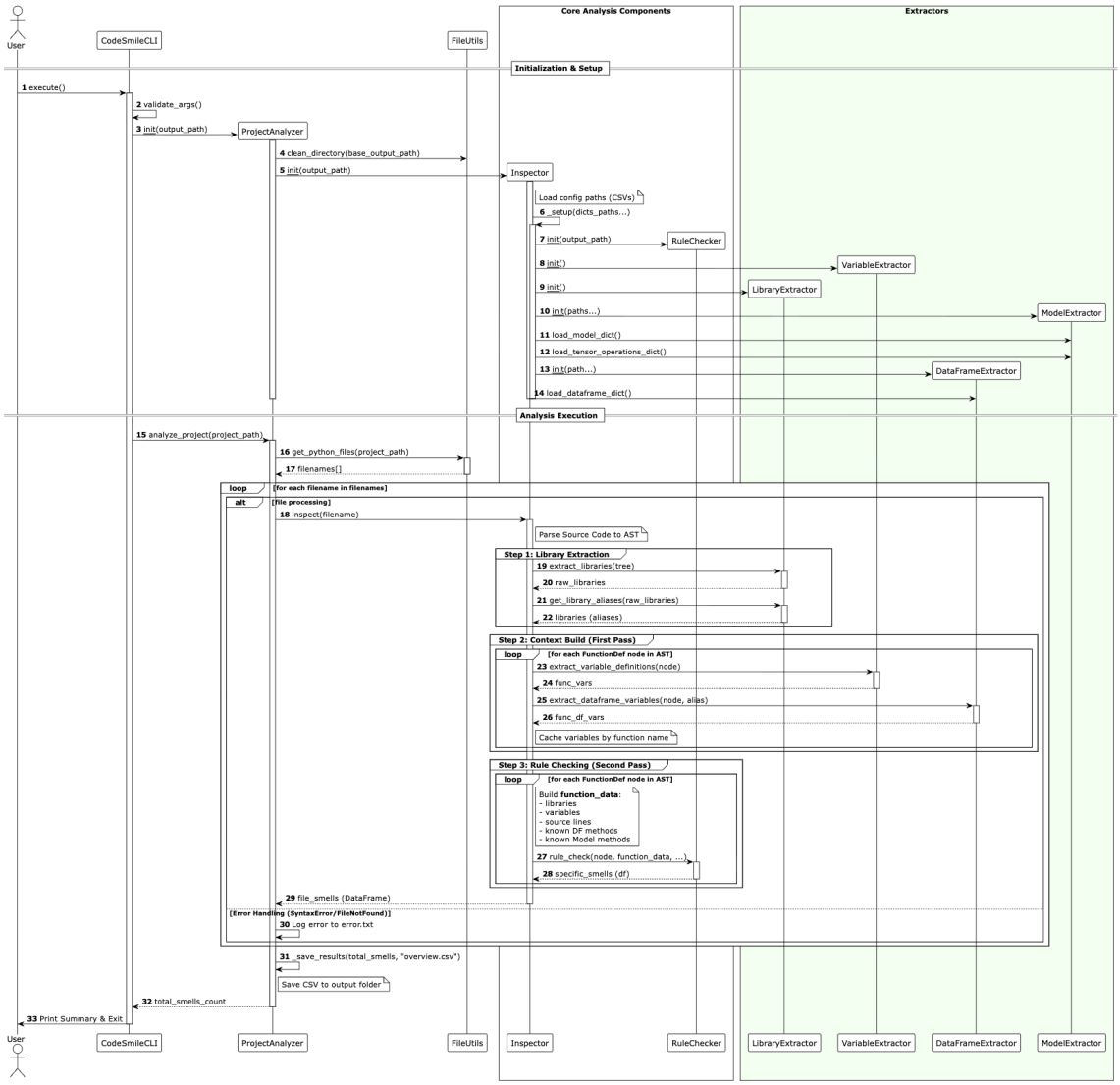


**Figura 3.1:** Sequence Diagram CLI.

**Nota sul Sequence Diagram:** Il diagramma focalizza l'attenzione sulla logica interna di *Analysis Execution* (passi 15-32). Mostra nel dettaglio come componenti quali `Inspector`, `Extractors` e `RuleChecker` collaborano per analizzare un singolo file (o una lista sequenziale), costruendo l'AST e verificando le regole definite.

### 3.2.2 Flusso di Utilizzo tramite GUI (Graphical User Interface)

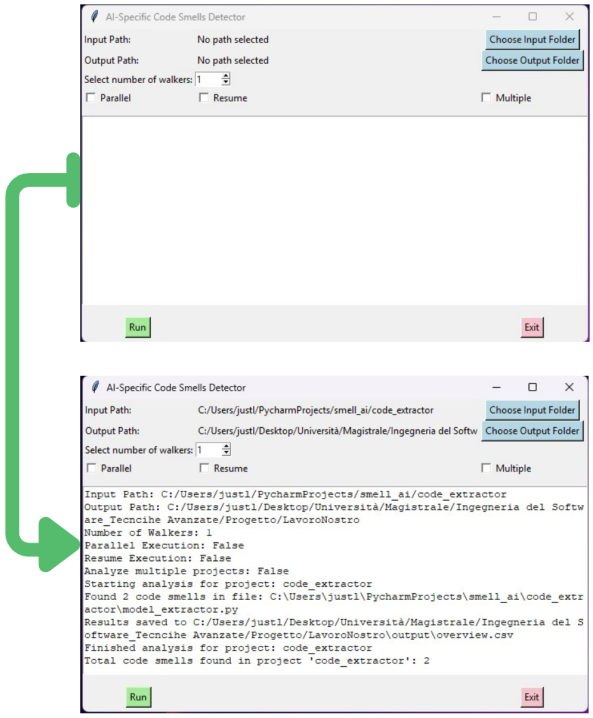


**Figura 3.2:** Sequence Diagram GUI.

1. L'utente esegue `gui_runner.py` aprendo la finestra principale. Tramite i pulsanti dedicati, naviga nel file system per selezionare visivamente le cartelle di progetto e di destinazione.

2. Attraverso widget di controllo (checkbox e slider), l'utente imposta le preferenze di analisi, come l'abilitazione del multi-threading o il numero di walker.

3. Alla pressione del pulsante "Run Analysis", la GUI avvia un thread secondario (`AnalysisThread`) dedicato esclusivamente all'esecuzione del `ProjectAnalyzer`, mantenendo l'interfaccia principale reattiva (non bloccante).

4. Un componente specifico (`TextBoxRedirect`) intercetta lo stream di output standard (`stdout`) e lo reindirizza in tempo reale verso l'area di testo della finestra, permettendo all'utente di monitorare il progresso (file analizzati, log) passo dopo passo.

5. Al termine del thread di analisi, l'applicazione notifica l'utente tramite GUI e conferma che i file CSV di report sono stati salvati nella directory di output selezionata.

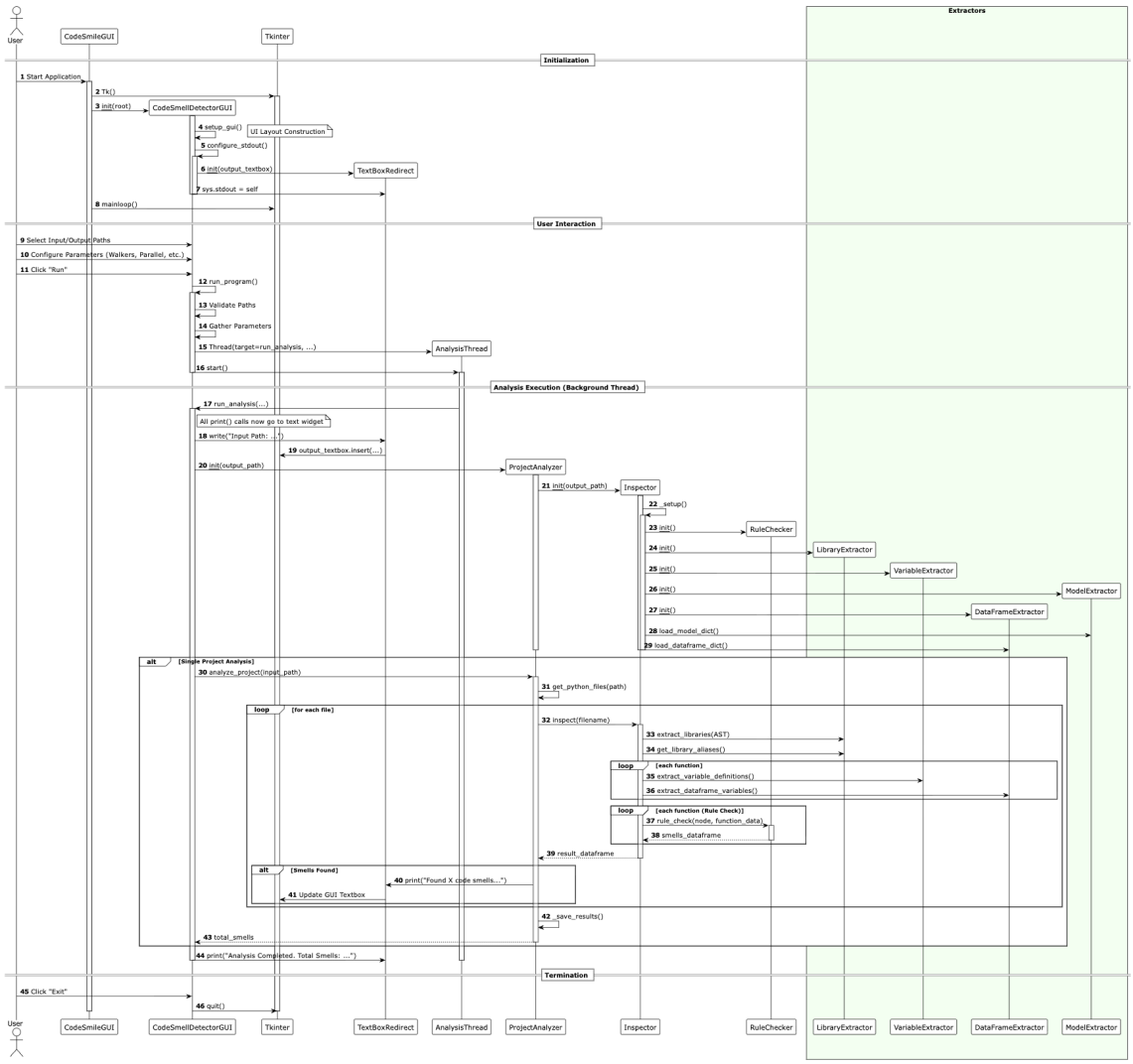


**Figura 3.3:** Sequence Diagram GUI.

### 3.2.3 Flusso di Utilizzo tramite Web App

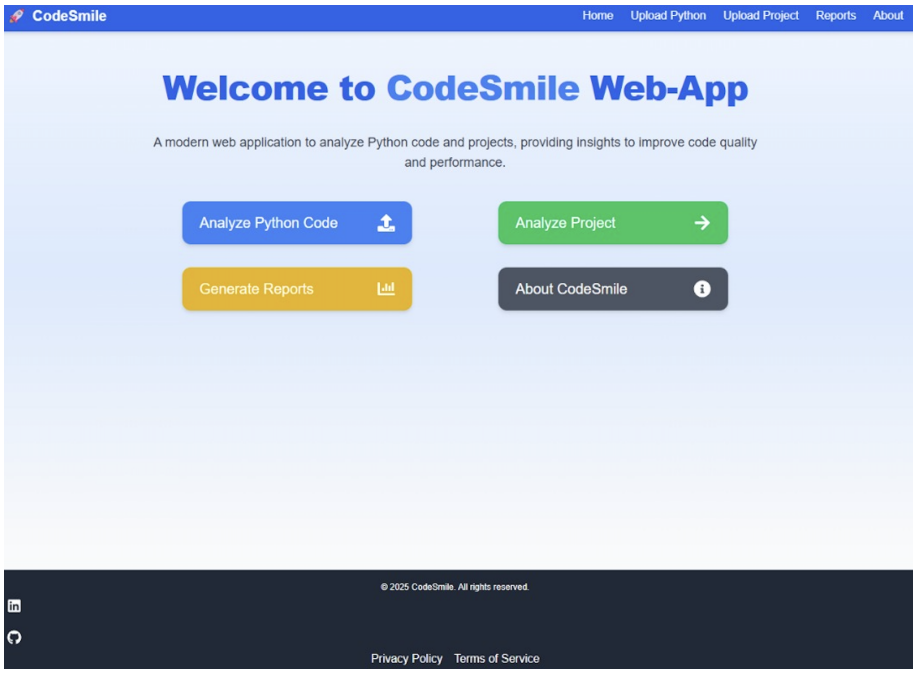


**Figura 3.4:** Sequence Diagram GUI.

1. L'utente accede all'applicazione web (Next.js) e utilizza il form di upload per caricare uno snippet di codice o un'intera cartella. Seleziona il motore di analisi (Statico o AI).

2. Il frontend itera sui file caricati e invia le richieste al Gateway API (FastAPI). Questo componente agisce da orchestratore, validando le richieste e instradandole al microservizio specifico.

3. **Elaborazione dei Microservizi:**

   - Nell'Analisi Statica (`Static Service`), il codice viene salvato in un file temporaneo; viene quindi invocato l'Inspector che riutilizza la logica core (AST parsing) per individuare le violazioni e restituire i risultati.
   - Nell'Analisi AI (`AI Service`), il codice viene inoltrato al modello LLM locale (gestito tramite Ollama), che effettua l'inferenza e restituisce una risposta strutturata sugli smell identificati.

4. I risultati vengono normalizzati in formato JSON e restituiti al frontend, che aggiorna lo stato locale (`ProjectContext`) e visualizza dinamicamente le card di errore nella dashboard.

5. **Generazione Report (Opzionale):** Su richiesta dell'utente ("Generate Report"), il frontend contatta il `Report Service`. Questo microservizio aggrega i dati analizzati, calcola le statistiche complessive e genera un documento scaricabile (PDF/JSON) contenente il riepilogo dell'analisi.

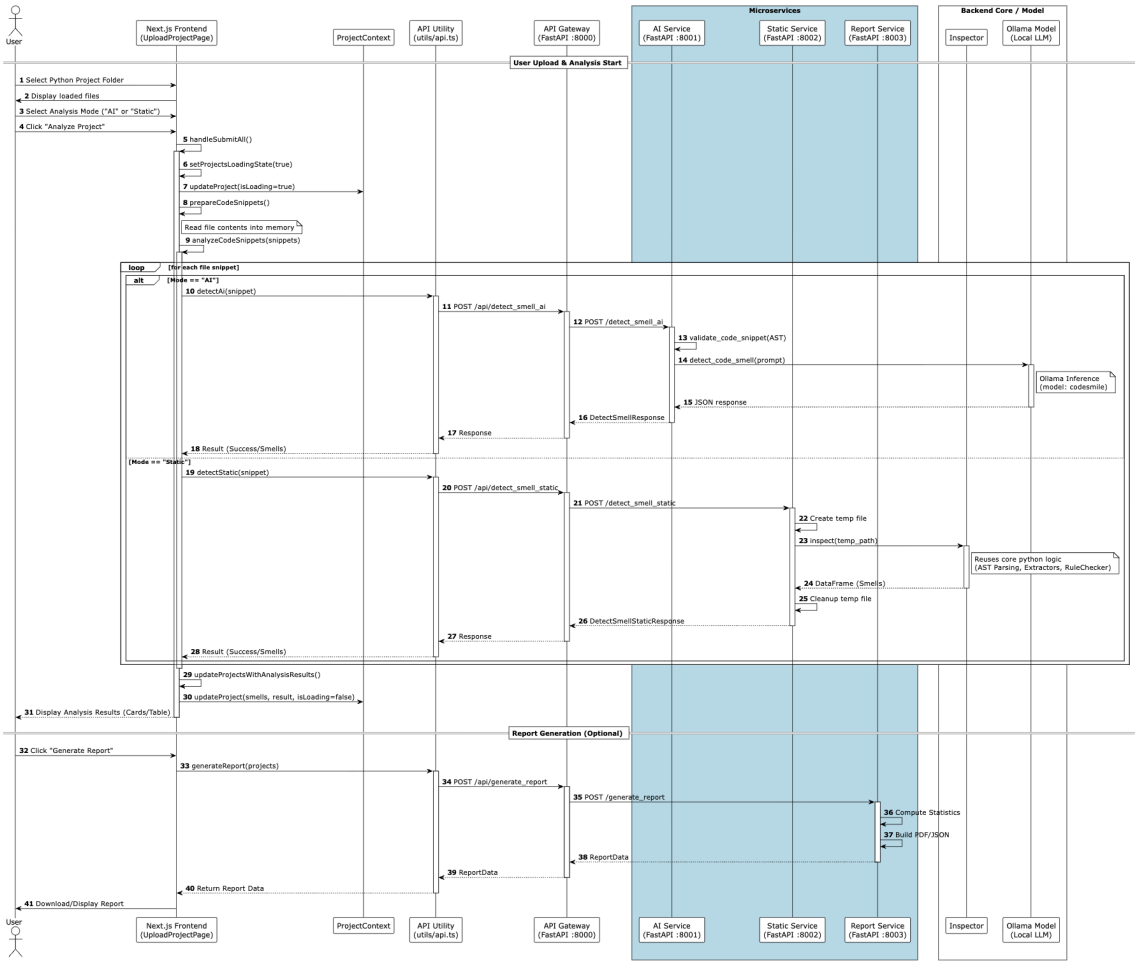


**Figura 3.5:** Sequence Diagram Webapp.