UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

Ingegneria del Software: Tecniche Avanzate

Progetto ISTA – Evoluzione di CodeSmile

Reverse Engineering

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

Elenco delle tabelle

Elenco delle figure

1 Introduzione

Il presente documento descrive il reverse engineering del sistema *CodeSmile*, con l'obiettivo di documentarne la struttura, i componenti principali e il funzionamento complessivo nello stato attuale. La descrizione è basata sulle informazioni fornite nel lavoro di Recupito et al. [1], in cui CodeSmile viene introdotto e utilizzato come strumento di supporto per l'analisi empirica dei *machine learning-specific code smells*.

Il documento fornisce una descrizione tecnica del sistema esistente, senza proporre modifiche, e costituisce la base per le successive attività di pianificazione dei test, analisi dell'impatto ed evoluzione del sistema.

2 Descrizione Generale del Sistema

CodeSmile è uno strumento progettato per la rilevazione automatica di un insieme predefinito di *machine learning-specific code smells* (ML-CSs) in progetti Python. Il sistema opera mediante analisi statica del codice sorgente e consente di individuare pattern di implementazione considerati subottimali nel contesto di applicazioni che utilizzano librerie e framework di Machine Learning.

Il tool implementa un insieme di regole di detection dedicate alla rilevazione degli ML-specific code smells e consente di individuarne le occorrenze nel codice sorgente senza richiedere l'esecuzione dei programmi analizzati. L'analisi viene effettuata a livello di file sorgente ed è basata sulla costruzione e sull'ispezione dell'Abstract Syntax Tree (AST).

2.1 Machine Learning-Specific Code Smells

Nel lavoro di Recupito et al. viene definito un insieme di sedici *machine learning-specific code smells*, individuati attraverso un'analisi sistematica delle pratiche di sviluppo di applicazioni di Machine Learning. Tali smell rappresentano pratiche di implementazione potenzialmente problematiche e sono stati suddivisi in due categorie principali: *Generic ML Code Smells* e *API-Specific Code Smells*.

2.1.1 Generic ML Code Smells. I *Generic ML Code Smells* comprendono smell che possono manifestarsi in applicazioni di Machine Learning indipendentemente dallo specifico framework utilizzato. Essi sono legati all'uso improprio di strutture dati, API o pattern di programmazione comunemente impiegati nello sviluppo di sistemi ML-enabled.

2.1.2 API-Specific Code Smells. Gli *API-Specific Code Smells* sono smell strettamente legati all'uso di API specifiche fornite dai framework e dalle librerie per l'implementazione, l'addestramento e l'esecuzione di modelli di Machine Learning, come PyTorch, TensorFlow e NumPy. Essi riguardano l'uso improprio di metodi, funzioni o costrutti messi a disposizione da tali librerie e possono influire sul corretto funzionamento o sull'efficienza dei modelli implementati.

2.2 Detection Rules

La rilevazione dei machine learning-specific code smells in CodeSmile è basata su un insieme di *detection rules*, ciascuna associata a uno specifico code smell. Le regole formalizzano pattern sintattici e strutturali osservabili nel codice sorgente Python e consentono di individuare le occorrenze degli smell mediante analisi statica, senza richiedere l'esecuzione del programma.

Le detection rules operano sull'Abstract Syntax Tree (AST) e verificano condizioni relative all'uso di API, alla struttura delle chiamate di funzione e alla composizione delle operazioni, in particolare in riferimento alle librerie Pandas, NumPy, PyTorch e TensorFlow. Ogni regola è progettata per individuare un singolo ML-specific code smell e produce informazioni relative al tipo di smell rilevato e alla sua localizzazione nel codice.

Tabella 2.1. Generic ML Code Smells

Nome	Descrizione
Broadcasting Feature Not Used	Operazioni su tensori che non sfruttano la funzionalità di broadcasting di TensorFlow.
Columns and DataType Not Explicitly Set	Creazione di DataFrame senza specificare esplicitamente nomi delle colonne e tipi di dato.
Deterministic Algorithm Option Not Used	Mancata rimozione dell'opzione `torch.use_deterministic_algorithms(True)` quando non necessaria.
Empty Column Misinitialization	Inizializzazione impropria di colonne di DataFrame con valori predefiniti come zeri o stringhe vuote.
Hyperparameters Not Explicitly Set	Mancata definizione esplicita degli iperparametri di un modello o di una pipeline di training.
In-Place APIs Misused	Uso errato di API che non modificano la struttura dati in-place, assumendo che la modifica avvenga senza riassegnazione.
Memory Not Freed	Mancata liberazione della memoria associata a modelli o strutture allocate in cicli, con rischio di crescita non controllata dell'uso di memoria.
Merge API Parameter Not Explicitly Set	Uso di operazioni di merge in Pandas senza specificare esplicitamente i parametri `how` e `on`.
NaN Equivalence Comparison Misused	Confronti non corretti con valori NaN (es. uso dell'uguaglianza diretta).
Unnecessary Iteration	Uso di iterazioni esplicite quando sono disponibili operazioni vettoriali/aggregate più appropriate.

Tabella 2.2. API-Specific Code Smells

Nome	Descrizione
Chain Indexing	Uso inefficiente di indicizzazione concatenata su DataFrame (es. accessi del tipo `df["col"][0]`).
DataFrame Conversion API Misused	Conversione impropria dei DataFrame in array NumPy (es. uso di `values()` al posto di API dedicate).
Gradients Not Cleared	Mancata pulizia dei gradienti prima della backpropagation (es. omissione di `zero_grad`).
Matrix Multiplication API Misused	Uso improprio delle API per la moltiplicazione di matrici (es. impiego di primitive non appropriate).
PyTorch Call Method Misused	Invocazione diretta di `forward()` invece dell'invocazione del modello come callable (es. `self.net()`).
TensorArray Not Used	Uso inefficiente di costrutti dentro cicli (es. uso di costanti/tensori al posto di strutture dedicate come `TensorArray`).

Tabella 2.3. Detection Rules per ML-Specific Code Smells

Code Smell	Criterio di Detection
Chain Indexing	Presenza di accessi concatenati a DataFrame Pandas (es. `df["one"]["two"]`), che vengono interpretati come operazioni separate.
Columns and DataType Not Explicitly Set	Creazione di DataFrame senza specificare esplicitamente i nomi delle colonne e i tipi di dato (es. uso di `pd.read_csv` senza parametri dedicati).
DataFrame Conversion API Misused	Uso dell'API `values()` per convertire DataFrame in array NumPy, invece dell'uso di `to_numpy()`.
In-Place APIs Misused	Invocazione di API che possono operare in-place senza verificare se il metodo modifica l'oggetto o restituisce una copia.
Gradients Not Cleared	Assenza della chiamata a `optimizer.zero_grad()` prima della backpropagation in codice PyTorch.
Matrix Multiplication API Misused	Uso di `np.dot()` in contesti in cui il comportamento dipende dalle dimensioni degli input, causando ambiguità semantica.
Memory Not Freed	Allocazione ripetuta di modelli o tensori in cicli senza rilascio esplicito delle risorse (es. assenza di `clear_session()` o `detach()`).
Merge API Parameter Not Explicitly Set	Uso della funzione `merge` di Pandas senza specificare parametri fondamentali come `how`, `on` o `validate`.
NaN Equivalence Comparison Misused	Confronti diretti con valori NaN (es. `np.nan == np.nan`) che producono risultati non intuitivi.
PyTorch Call Method Misused	Uso diretto di `self.net.forward()` invece dell'invocazione del modello come callable (`self.net()`).
TensorArray Not Used	Uso di `tf.constant()` per la crescita di array in un loop, invece di strutture dedicate come `TensorArray`.
Unnecessary Iteration	Presenza di iterazioni esplicite su oggetti Pandas o TensorFlow invece di soluzioni vettorializzate.
Broadcasting Feature Not Used	Uso esplicito di `tf.tile` in operazioni aritmetiche di TensorFlow che potrebbero sfruttare il meccanismo di broadcasting automatico.
Deterministic Algorithm Option Not Used	Presenza di `torch.use_deterministic_algorithms(True)`, che forza l'uso di algoritmi deterministici indipendentemente dal contesto di utilizzo.
Empty Column Misinitialization	Inizializzazione di colonne di DataFrame Pandas con valori costanti come `0` o stringhe vuote, invece di valori nulli espliciti.
Hyperparameters Not Explicitly Set	Istanziazione di modelli di Machine Learning senza specificare esplicitamente gli iperparametri tramite argomenti o keyword.

3 Componenti del Sistema

CodeSmile è strutturato come una suite composta da tre componenti principali, ciascuno responsabile di una specifica fase del processo di rilevazione e gestione dei machine learning-specific code smells. Le componenti sono progettate per operare in modo indipendente, ma possono essere integrate all'interno di un flusso di analisi comune.

Nel seguito vengono descritte le tre componenti principali del sistema: lo *Static Analysis Tool*, l'*AI-Based Detection Tool* che è un componente di rilevazione basato su tecniche di apprendimento automatico e la *Web Application*.

3.1 Static Analysis Tool

Lo *Static Analysis Tool* rappresenta il componente centrale di CodeSmile ed è responsabile della rilevazione dei machine learning-specific code smells tramite analisi statica del codice sorgente Python. Il tool opera costruendo l'Abstract Syntax Tree (AST) dei file analizzati e applicando un insieme di detection rules, ciascuna associata a uno specifico code smell.

Il processo di analisi si basa sull'ispezione della struttura sintattica del codice e sull'identificazione di pattern relativi all'uso di API, alla composizione delle operazioni e alle chiamate di funzione, in particolare in riferimento alle librerie Pandas, NumPy, PyTorch e TensorFlow. L'analisi non richiede l'esecuzione del codice e consente di individuare le occorrenze degli smell direttamente a livello di sorgente.

L'output prodotto dallo Static Analysis Tool consiste in un insieme di informazioni strutturate che riportano, per ciascuna occorrenza rilevata, il tipo di code smell e la sua localizzazione nel codice.

3.2 AI-Based Detection Tool

Accanto allo strumento di analisi statica, CodeSmile include un componente di rilevazione basato su tecniche di apprendimento automatico. Nel lavoro di riferimento, tale componente è presentato come uno strumento sperimentale, utilizzato per valutare la fattibilità dell'impiego di modelli di Machine Learning nella classificazione dei code smells.

Nella versione attuale del sistema, come indicato nella documentazione tecnica, il componente AI-Based è implementato utilizzando modelli linguistici di grandi dimensioni (LLM) ed è finalizzato alla classificazione dei code smells a partire dal codice sorgente. Questo componente opera in modo complementare allo Static Analysis Tool e non fornisce informazioni di localizzazione a livello di AST.

3.3 Web Application

CodeSmile include una *Web Application* che fornisce un'interfaccia grafica per l'esecuzione delle analisi e la consultazione dei risultati prodotti dal sistema. La Web Application consente di avviare l'analisi del codice sorgente utilizzando i diversi componenti disponibili e di visualizzare le occorrenze dei code smells rilevati.

Nel sistema attuale, la Web Application svolge il ruolo di componente di orchestrazione e presentazione, facilitando l'utilizzo degli strumenti di analisi e l'accesso ai risultati.

4 Struttura e Architettura del Sistema

4.1 Organizzazione dei test dello Static Analysis Tool

La directory `test`, collocata nella root del repository, raccoglie la suite di verifica del sistema. Essa non costituisce un componente eseguito in produzione, ma contiene test automatizzati e casi di test utilizzati per validare il corretto funzionamento dello *Static Analysis Tool* lungo tre livelli: unità, integrazione e sistema.

- `unit_testing`: test di unità dei singoli package del core (`cli`, `gui`, `code_extractor`, `components`, `detection_rules`, `report`), mirati a verificare il comportamento delle classi principali in isolamento.
- `integration_testing`: test di integrazione che verificano la collaborazione tra componenti adiacenti della pipeline e includono casi di integrazione end-to-end con CLI e GUI.
- `system_testing`: casi di test organizzati in cartelle `TC*` contenenti progetti mock e file di input utilizzati per validare il comportamento del sistema su scenari realistici, inclusi input annidati e casi limite.

A supporto dell'attività di testing automatizzato, nella root del repository è presente il file `.coveragerc`, utilizzato per la configurazione della code coverage in combinazione con `pytest-cov`. La configurazione abilita il branch coverage e definisce in modo esplicito le parti del codice da escludere dal calcolo, concentrando la misurazione sui componenti core dello *Static Analysis Tool* testati tramite `pytest` e `unittest.mock`.

Sono esclusi dal calcolo della coverage i moduli non rilevanti ai fini della verifica funzionale (directory `test`, file di inizializzazione, ambienti virtuali, Web Application, dataset e componenti di fine-tuning).

4.2 Organizzazione dei test della Web Application

All'interno della directory `webapp` è presente una suite di test dedicata alla verifica della Web Application. Poiché il sottosistema è full-stack, i test risultano distribuiti su livelli differenti: unit testing del frontend, end-to-end testing dei flussi utente, e test di integrazione lato backend per verificare la comunicazione tra gateway e microservizi. Tali risorse non costituiscono componenti eseguiti in produzione, ma supportano la validazione del comportamento dell'applicazione nelle principali modalità d'uso.

- `_tests_`: suite di test di unità del frontend (Jest + Testing Library), focalizzata su componenti e pagine Next.js. I test verificano rendering, gestione dello stato e interazioni utente, isolando la UI tramite mocking delle dipendenze e delle chiamate alle API.
- `cypress`: suite di test end-to-end (Cypress) che valida i flussi principali dal punto di vista dell'utente finale. La directory include anche `fixtures` (dati e input di test) e `support` (utility e comandi riutilizzabili) a supporto degli scenari E2E.
- `integration_tests`: test di integrazione lato backend, orientati a verificare l'interazione tra il `gateway` e i servizi applicativi (servizio di report, servizio di analisi statica e servizio AI-based), controllando la corretta orchestrazione delle richieste e delle risposte lungo i confini API.

A completamento dell'infrastruttura di testing della Web Application, sono presenti file di configurazione dedicati ai framework utilizzati. Il file `jest.config.ts` definisce l'ambiente di esecuzione per i test di unità del frontend, mentre `cypress.config.ts` configura l'esecuzione dei test end-to-end, mirati alla validazione dei principali flussi di interazione utente dell'applicazione web.

4.3 Organizzazione dei Package

L'implementazione di CodeSmile è organizzata in un insieme di package che riflettono le principali componenti funzionali del sistema. Sebbene la struttura fisica del repository non separi rigidamente tali componenti in directory distinte, è possibile individuare una chiara organizzazione logica.

Dall'analisi della struttura del repository emergono tre macro-componenti principali:

- lo **Static Analysis Tool**;
- l'**AI-Based Detection Tool**;
- la **Web Application**.

I package dello *Static Analysis Tool* costituiscono il nucleo del sistema e includono i moduli per l'analisi statica basata su AST, l'implementazione delle detection rules e la gestione dei risultati.

Il componente *AI-Based Detection Tool* è composto da package dedicati alla preparazione dei dati, all'addestramento e all'esecuzione di modelli di Machine Learning, ed è logicamente separato dal flusso di analisi statica.

La *Web Application*, interamente contenuta nella directory `webapp`, comprende il frontend, un gateway di orchestrazione e servizi backend per l'esecuzione delle analisi e la generazione dei report.

4.3.1 Package dello Static Analysis Tool. Lo *Static Analysis Tool* è implementato attraverso un insieme di package collocati nella root del repository. Tali package realizzano il core di analisi statica del sistema e sono responsabili della costruzione dell'AST, dell'applicazione delle detection rules e della produzione dei risultati.

A supporto dell'analisi statica, il sistema utilizza inoltre risorse statiche esterne al codice, raccolte nella directory `obj_dictionaries`. Tale directory non costituisce un package Python, ma contiene file in formato CSV che forniscono dizionari e metadati relativi a costrutti tipici delle applicazioni di Machine Learning (ad esempio DataFrame, modelli e tensori), impiegati per facilitare il riconoscimento di pattern rilevanti durante l'analisi basata su AST.

I package che compongono lo Static Analysis Tool sono i seguenti:

- `cli` Implementa l'interfaccia a linea di comando per l'avvio dell'analisi statica.
- `gui` Fornisce un'interfaccia grafica locale per l'esecuzione dello Static Analysis Tool.
- `code_extractor` Contiene i moduli responsabili dell'estrazione delle informazioni dal codice sorgente e della costruzione dell'AST.
- `detection_rules` Implementa le detection rules per i machine learning-specific code smells, suddivise in `generic` e `api_specific`.
- `components` Include componenti di supporto utilizzati durante il processo di analisi.

- `report` Gestisce la generazione e l'organizzazione dei risultati prodotti dall'analisi statica.
- `utils` Include funzionalità di utilità condivise tra i diversi package.

Package `cli`. Il package `cli` include i seguenti file:

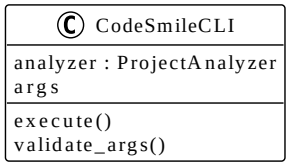

Fig. 4.1. Diagramma del package `cli`.

- `cli_runner.py`: punto di ingresso per l'esecuzione dell'analisi statica tramite linea di comando.

Package `gui`. Il package `gui` include i seguenti file:

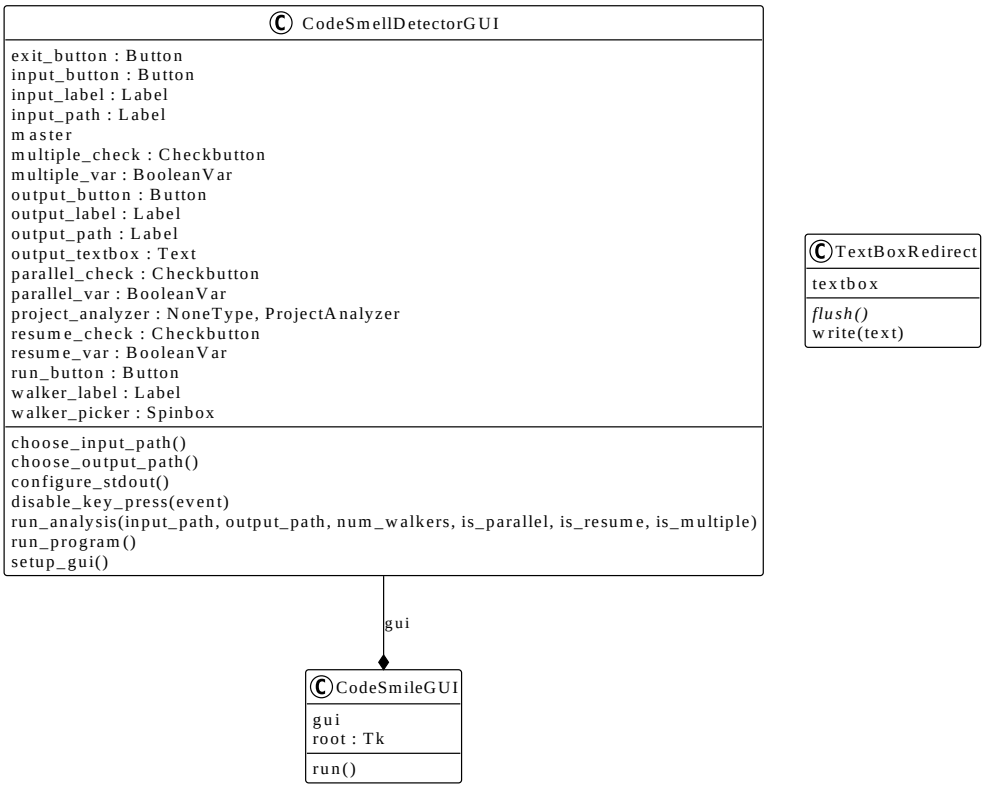

Fig. 4.2. Diagramma del package `gui`.

- `code_smell_detector_gui.py`: implementa l'interfaccia grafica principale basata su `tkinter`, consentendo all'utente di configurare i parametri di esecuzione dell'analisi e di avviare lo Static Analysis Tool.
- `gui_runner.py`: rappresenta il punto di ingresso per l'esecuzione dell'interfaccia grafica locale.
- `textbox_redirect.py`: fornisce una utility per il reindirizzamento dello standard output verso l'interfaccia grafica, permettendo la visualizzazione dei messaggi di esecuzione.

Package `code_extractor`. Il package `code_extractor` include i seguenti file:

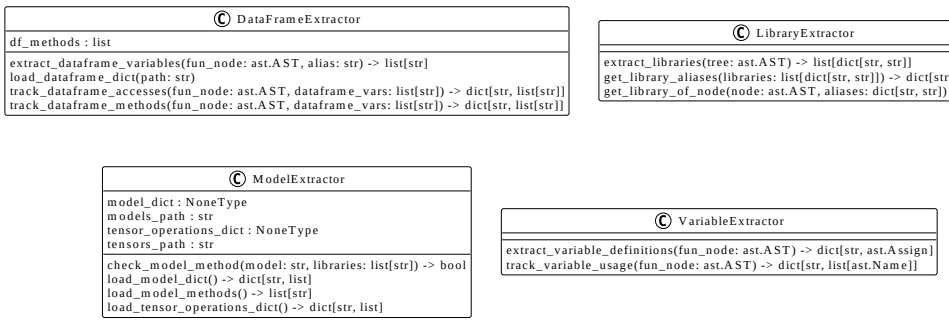

Fig. 4.3. Diagramma del package `code_extractor`.

- `dataframe_extractor.py`: fornisce funzionalità per l'individuazione delle variabili associate a DataFrame Pandas e per il tracciamento dei metodi e degli accessi alle colonne tramite analisi AST.
- `library_extractor.py`: estrae le informazioni relative alle librerie importate nel codice sorgente, gestendo alias e associazione tra nodi AST e librerie utilizzate.
- `model_extractor.py`: gestisce l'accesso a dizionari di modelli e operazioni tensoriali definiti tramite file CSV, fornendo supporto all'identificazione di modelli e operazioni rilevanti ai fini delle detection rules.
- `variable_extractor.py`: consente l'estrazione delle definizioni e degli utilizzi delle variabili all'interno delle funzioni, attraverso l'analisi dei nodi AST.

Package `detection_rules`. La rappresentazione grafica del package è suddivisa in tre diagrammi distinti: uno relativo alle classi comuni, uno per le regole dei *Generic ML Code Smells* e uno per le regole degli *API-Specific Code Smells*.

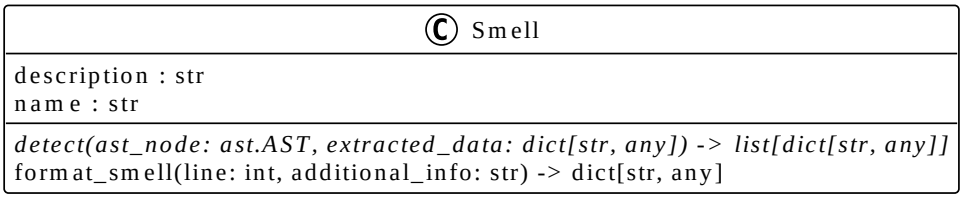

Fig. 4.4. Diagramma del file `smell.py`.

Fig. 4.5. Diagramma del package `generic`.

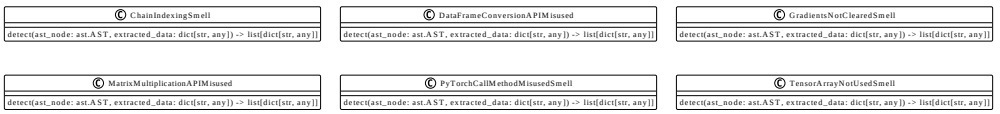

Fig. 4.6. diagramma del package `api_specific`.

Il package `detection_rules` include i seguenti file:

- `smell.py`: definisce la classe astratta `Smell`, che fornisce l'interfaccia comune a tutte le detection rules.

File che implementano una detection rule specifica per un *Generic ML Code Smell*, secondo i criteri di rilevazione descritti in 2.3:

- `broadcasting_feature_not_used.py`
- `columns_and_datatype_not_explicitly_set.py`
- `deterministic_algorithm_option_not_used.py`
- `empty_column_misinitialization.py`
- `hyperparameters_not_explicitly_set.py`
- `in_place_apis_misused.py`
- `memory_not_freed.py`
- `merge_api_parameter_not_explicitly_set.py`
- `nan_equivalence_comparison_misused.py`
- `unnecessary_iteration.py`

File che implementano una detection rule specifica per un *API-Specific Code Smell*, secondo i criteri di rilevazione descritti in 2.3:

- `chain_indexing_smell.py`
- `dataframe_conversion_api_misused.py`

- `gradients_not_cleared_before_backward_propagation.py`
- `matrix_multiplication_api_misused.py`
- `pytorch_call_method_misused.py`
- `tensor_array_not_used.py`

Package `components`. Il package `components` include i seguenti file:

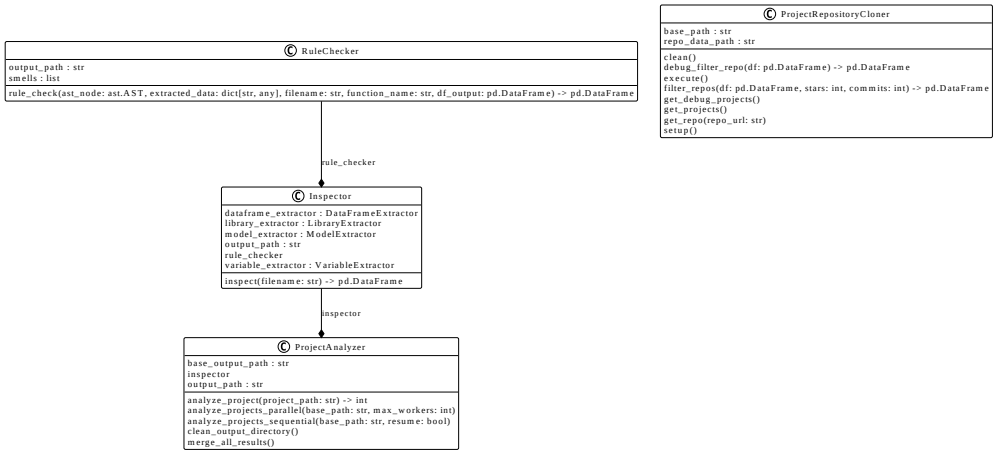

Fig. 4.7. Diagramma del package `components`.

- `inspector.py`: esegue l'analisi di un singolo file Python costruendo l'AST, raccogliendo i dati estratti (librerie, variabili, modelli, operazioni) e delegando l'applicazione delle detection rules al componente `RuleChecker`.
- `project_analyzer.py`: coordina l'analisi di un progetto (o di più progetti), gestendo la scansione dei file Python, l'eventuale esecuzione parallela e la produzione dei risultati aggregati in formato CSV.
- `project_repository_cloner.py`: fornisce funzionalità di supporto per il reperimento di progetti di input tramite clonazione di repository GitHub e filtraggio basato su metadati di dataset.
- `rule_checker.py`: inizializza e applica l'insieme delle detection rules (generic e API-specific) ai nodi AST analizzati, raccogliendo le occorrenze rilevate in una struttura dati uniforme.

Package `report`. Il package `report` include i seguenti file:

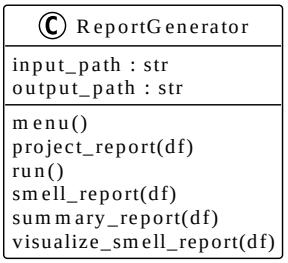

Fig. 4.8. Diagramma del package `report`.

- `report_generator.py`: implementa la generazione di report a partire dai file CSV prodotti dall'analisi. Supporta la produzione di: (i) report aggregati per tipo di smell, (ii) report aggregati per progetto, (iii) un report riassuntivo in formato `.xlsx` e (iv) una visualizzazione grafica (bar chart) delle occorrenze.

Package `utils`. Poiché il package `utils` contiene esclusivamente una classe di supporto con metodi statici e non presenta relazioni strutturali significative con altre classi del sistema, non è stato realizzato un diagramma delle classi dedicato.

Il package `utils` include i seguenti file:

- `file_utils.py`: raccoglie metodi di utilità per operazioni su file e directory, tra cui: pulizia/creazione della directory di output, individuazione dei file `.py` da analizzare, merge dei risultati CSV, e gestione di un file di log per supportare esecuzioni multiple e modalità di resume.

4.3.2 Package dell'AI-Based Detection Tool. Il componente *AI-Based Detection Tool* è implementato attraverso un insieme di package collocati nella root del repository e dedicati alle attività di preparazione dei dati, addestramento e validazione di modelli di Machine Learning. Tale componente supporta l'individuazione dei *ML-Specific Code Smells* tramite approcci basati su apprendimento automatico ed è logicamente separato dal flusso di analisi statica basata su AST.

I package dell'AI-Based Detection Tool non operano direttamente sul codice sorgente, ma lavorano su dataset derivati dai risultati dell'analisi statica o da collezioni di progetti, realizzando pipeline di preprocessing, training e valutazione dei modelli.

Oltre ai package Python, il repository include diverse directory di supporto che contengono dati di input, output intermedi e dataset utilizzati durante le sperimentazioni. Tali directory non costituiscono package Python e non contengono codice eseguibile, ma rappresentano risorse dati necessarie all'esecuzione delle pipeline di Machine Learning. In particolare:

- `input`: contiene dataset di riferimento e progetti di esempio, utilizzati come sorgente dei dati di partenza per le analisi e le fasi di addestramento.
- `output`: raccoglie i risultati prodotti dall'analisi di progetti reali, organizzati per repository e salvati in formato CSV.

- `datasets`: include dataset intermedi e finali utilizzati per l'addestramento e la validazione dei modelli, nonché file di supporto generati durante le fasi di preprocessing e data injection.

Le directory sopra elencate non vengono trattate come package e non sono oggetto di diagrammi UML, in quanto non contengono classi o relazioni strutturali rilevanti dal punto di vista architetturale.

I package che compongono l'AI-Based Detection Tool sono i seguenti:

- `data_preparation` Contiene i moduli dedicati alla preparazione e al preprocessing dei dati, inclusa la trasformazione dei risultati dell'analisi statica in feature utilizzabili dai modelli di Machine Learning.
- `finetuning` Include le componenti responsabili dell'addestramento e della validazione dei modelli, organizzate nelle sottodirectory `train` e `validation`.

Package `data_preparation`. Il package `data_preparation` include i seguenti file:

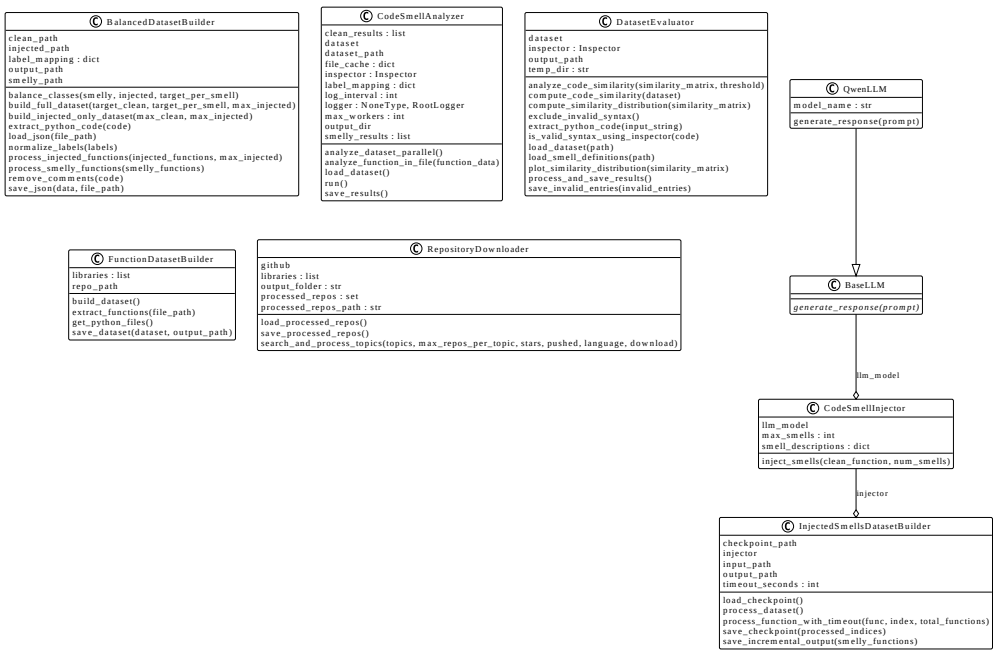

Fig. 4.9. Diagramma del package `data_preparation`.

- `balanced_dataset_builder.py`: costruisce dataset unificati e bilanciati combinando funzioni pulite, funzioni con smell rilevati e funzioni con smell iniettati, producendo output in formato JSON.
- `base_llm.py`: definisce un'interfaccia astratta comune per i modelli linguistici utilizzati nel sistema.
- `code_smell_analyzer.py`: analizza automaticamente un dataset di funzioni invocando lo `Static Analysis Tool` e classifica le funzioni come *clean* o *smelly*.

- `code_smell_injector.py`: implementa l'iniezione artificiale di code smells in funzioni pulite tramite un modello LLM, generando versioni modificate del codice che preservano la funzionalità originale.
- `dataset_creation_runner.py`: coordina l'esecuzione a step dell'intera pipeline di costruzione del dataset, includendo download dei repository, estrazione delle funzioni, analisi statica, iniezione degli smell e generazione dei dataset finali.
- `dataset_evaluator.py`: fornisce funzionalità di validazione e analisi del dataset, includendo controlli di sintassi e analisi di similarità tra blocchi di codice.
- `function_dataset_builder.py`: estrae funzioni Python da repository e directory di progetto, selezionando quelle rilevanti per il dominio del Machine Learning e producendo un dataset JSON annotato.
- `injected_smells_dataset_builder.py`: costruisce un dataset di funzioni con smell iniettati a partire da funzioni pulite, supportando l'elaborazione incrementale e il recupero in caso di interruzioni.
- `qwen_llm.py`: implementa un modello linguistico concreto basato su *Qwen2.5-Coder*, utilizzato per la generazione di codice durante la fase di iniezione automatica dei code smell.
- `repository_downloader.py`: gestisce la ricerca e il download automatico di repository GitHub rilevanti per il dominio del Machine Learning, filtrandoli in base a criteri di linguaggio, popolarità e dipendenze.

Package `finetuning`. Il package `finetuning` è diviso nelle sottodirectory `train` e `validation`.

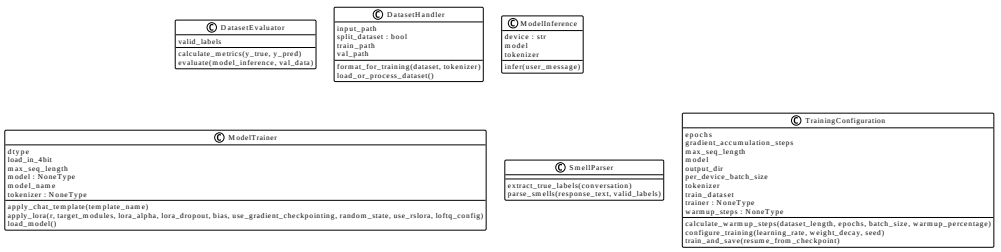

Fig. 4.10. Diagramma del package `finetuning`.

Il package `finetuning.train` include i seguenti file:

- `dataset_handler.py`: gestisce il caricamento e la preparazione del dataset per il fine-tuning, supportando sia l'uso di split già disponibili sia la loro generazione automatica, e formattando i dati secondo il chat template richiesto dal modello.
- `model_trainer.py`: si occupa del caricamento del modello pre-addestrato e dell'applicazione delle tecniche di adattamento (LoRA), configurando tokenizer e parametri necessari all'addestramento.
- `training_configuration.py`: centralizza la configurazione del processo di training, definendo iperparametri, strategia di ottimizzazione e modalità di addestramento, e gestendo il salvataggio finale del modello nei formati previsti.

- `training_runner.py`: rappresenta il punto di ingresso del fine-tuning, coordinando le diverse fasi della pipeline di addestramento e consentendo la selezione delle modalità operative tramite CLI.

Il package `finetuning.validation` include i seguenti file:

- `dataset_evaluator.py`: gestisce la valutazione del modello sul dataset di validazione, confrontando le predizioni con le etichette di ground truth e calcolando metriche globali di accuratezza e classificazione.
- `model_inference.py`: incapsula la logica di inferenza del modello fine-tuned, occupandosi della preparazione dell'input, della generazione delle risposte e della loro decodifica.
- `smell_parser.py`: fornisce funzioni di supporto per l'estrazione e la validazione delle etichette di code smell, sia dalle risposte del modello sia dai dati di ground truth.
- `validation_runner.py`: rappresenta il punto di ingresso del processo di validazione, coordinando il caricamento del modello, l'esecuzione dell'inferenza e il calcolo delle metriche finali sul dataset di validazione.

4.3.3 Package della Web Application. La *Web Application* costituisce il livello di interazione utente del sistema ed è progettata come applicazione full-stack, combinando un frontend web basato su *React / Next.js* e un backend a microservizi accessibile tramite API REST. Essa non replica la logica di analisi, ma funge da orchestratore e punto di accesso ai servizi di analisi statica, detection AI-based e generazione dei report.

All'interno della directory `webapp` sono presenti sia package eseguibili in produzione sia risorse di configurazione, testing e supporto allo sviluppo, che non costituiscono package applicativi veri e propri.

In particolare, le seguenti directory e file svolgono un ruolo infrastrutturale o di supporto:

- `app`, `public`, `components`, `context`, `types`, `utils`: risorse del frontend Next.js, comprensive di pagine, componenti React, contesto applicativo e utility condivise.
- `cypress`, `_tests_`: directory dedicate al testing end-to-end e al testing di unità del frontend.
- File di configurazione quali `next.config.js`, `tailwind.config.ts`, `tsconfig.json`, `eslint.config.mjs`, `jest.config.ts`, necessari alla build, allo stile e al testing dell'applicazione web.
- File di containerizzazione e deploy (`Dockerfile`, `package.json`, `package-lock.json`), utilizzati per l'esecuzione e la distribuzione dell'applicazione.

I package che costituiscono il backend applicativo della Web Application, responsabili dell'orchestrazione delle funzionalità di analisi, sono invece i seguenti:

- `gateway` Implementa il punto di ingresso backend della Web Application. Espone API REST utilizzate dal frontend e coordina l'invocazione dei servizi sottostanti (analisi statica, detection AI-based e generazione dei report).
- `services.staticanalysis` Incapsula il servizio backend dedicato allo *Static Analysis Tool*, fornendo endpoint REST per l'analisi del codice e l'esecuzione delle detection rules.

- `services.aiservice` Implementa il servizio di *AI-Based Detection*, responsabile dell'inferenza tramite modelli LLM fine-tuned e della classificazione dei code smells.
- `services.report` Gestisce la generazione e l'aggregazione dei risultati delle analisi, producendo dati strutturati per la visualizzazione dei report all'interno della Web Application.

Per la Web Application non sono stati prodotti i class diagram, poiché l'implementazione è distribuita su più tecnologie e paradigmi e l'architettura è descritta in modo più informativo tramite il *sequence diagram* del flusso di esecuzione (5.3).

Package `gateway`. Il package `gateway` include i seguenti file:

- `main.py` Definisce l'applicazione FastAPI del gateway, configura il middleware CORS e implementa gli endpoint REST che fungono da proxy verso i servizi `aiservice`, `staticanalysis` e `report`, gestendo l'inoltro delle richieste e il recupero delle risposte.
- `Dockerfile` Specifica l'ambiente di esecuzione containerizzato del gateway, includendo l'installazione delle dipendenze e l'avvio del servizio tramite `uvicorn`.
- `requirements.txt` Elenca le dipendenze Python necessarie all'esecuzione del gateway.

Package `services.staticanalysis`. Il package `services.staticanalysis` include i seguenti file:

- `app/main.py` Inizializza il servizio FastAPI, configura il middleware CORS e registra il router che espone gli endpoint REST del servizio di analisi statica.
- `app/routers/detect_smell.py` Definisce l'endpoint `/detect_smell_static` (REST) che riceve uno snippet di codice, invoca la logica di analisi statica e restituisce i risultati serializzati.
- `app/utils/static_analysis.py` Implementa la logica di analisi statica, creando un file temporaneo a partire dallo snippet di codice, invocando l'`Inspector` e trasformando l'output in oggetti di dominio.
- `app/schemas/requests.py` Definisce lo schema della richiesta utilizzata dall'endpoint di analisi statica.
- `app/schemas/responses.py` Definisce gli schemi di risposta utilizzati per rappresentare gli smell rilevati dal servizio.
- `Dockerfile` Specifica l'ambiente di esecuzione containerizzato del servizio e il comando di avvio tramite `uvicorn`.
- `requirements.txt` Elenca le dipendenze Python necessarie all'esecuzione del servizio.

Package `services.aiservice`. Il package `services.aiservice` include i seguenti file:

- `app/main.py` Inizializza il servizio FastAPI, abilita il CORS e registra il router che espone l'endpoint di AI-based detection.
- `app/routers/detect_smell.py` Definisce l'endpoint REST `/detect_smell_ai`: valida lo snippet (parsing AST), invoca il modello di inferenza e serializza la risposta secondo gli schemi definiti.

- `app/utils/model.py` Implementa il wrapper del modello: costruisce il prompt, interroga l'API di Ollama per la generazione e parsifica l'output per estrarre le label dei code smells.
- `app/schemas/requests.py` Definisce lo schema della richiesta per la detection AI-based.
- `app/schemas/responses.py` Definisce gli schemi di risposta (inclusa la rappresentazione degli smell) prodotti dal servizio.
- `Dockerfile` Specifica l'ambiente di esecuzione containerizzato del servizio e il comando di avvio tramite `uvicorn`.
- `requirements.txt` Elenca le dipendenze Python necessarie all'esecuzione del servizio.

Package `services.report`. Il package `services.report` include i seguenti file:

- `app/main.py` Inizializza il servizio FastAPI, abilita il CORS e registra il router per l'endpoint di generazione report.
- `app/routers/report.py` Definisce l'endpoint REST `/generate_report` che riceve i risultati di analisi per più progetti e restituisce i dati aggregati del report.
- `app/utils/report_generator.py` Implementa la logica di aggregazione: combina gli smell dei progetti e costruisce un output strutturato utile alla visualizzazione nel frontend.
- `app/schemas/requests.py` Definisce gli schemi della richiesta (progetti, file e smell) usati per invocare la generazione del report.
- `app/schemas/responses.py` Definisce lo schema di risposta contenente `report_data` (dati per grafici e riepiloghi).
- `Dockerfile` Specifica l'ambiente di esecuzione containerizzato del servizio e il comando di avvio tramite `uvicorn`.
- `requirements.txt` Elenca le dipendenze Python necessarie all'esecuzione del servizio.

5 Flussi di Esecuzione del Sistema

In questa sezione vengono descritti i flussi di esecuzione di *CodeSmile*, al fine di illustrare il comportamento dinamico del sistema e l'interazione tra i suoi componenti durante l'utilizzo.

I flussi sono rappresentati tramite *sequence diagram*, utilizzati per mostrare l'ordine temporale delle operazioni e lo scambio di dati tra i moduli coinvolti.

Il sistema prevede tre modalità di esecuzione:

(1) esecuzione tramite **Command Line Interface (CLI)**;
(2) esecuzione tramite **interfaccia grafica locale (GUI)**;
(3) esecuzione tramite **Web Application**.

Per ciascuna modalità viene presentato un sequence diagram dedicato, accompagnato da una descrizione sintetica delle fasi che compongono il flusso di esecuzione.

5.1 Flusso di esecuzione tramite CLI

Il flusso di esecuzione tramite interfaccia a linea di comando descrive il funzionamento dello *Static Analysis Tool* quando viene avviato direttamente dall'utente tramite terminale.

(1) L'utente avvia l'analisi da terminale, specificando i parametri di input e output tramite la CLI.
(2) Il modulo `cli_runner.py` effettua il parsing degli argomenti e inizializza l'istanza di `CodeSmileCLI`, validando le opzioni fornite.
(3) Il `ProjectAnalyzer` avvia l'analisi del progetto, individuando ricorsivamente i file Python presenti nella directory di input.
(4) Per ciascun file individuato, l'`Inspector` avvia il processo di analisi statica.
(5) Il codice sorgente viene parsificato tramite `ast.parse`, con la costruzione dell'albero sintattico astratto (AST).
(6) I moduli di estrazione analizzano l'AST per individuare librerie, variabili, DataFrame, modelli e operazioni rilevanti.
(7) Il `RuleChecker` applica l'insieme delle detection rules (generic e API-specific), individuando le occorrenze di code smell.
(8) I risultati dell'analisi vengono raccolti e salvati in formato CSV nella directory di output.
(9) Al termine dell'elaborazione, il `ReportGenerator` produce eventuali report aggregati e riepilogativi.
(10) L'esecuzione termina con la generazione dell'output finale e la notifica di completamento all'utente.

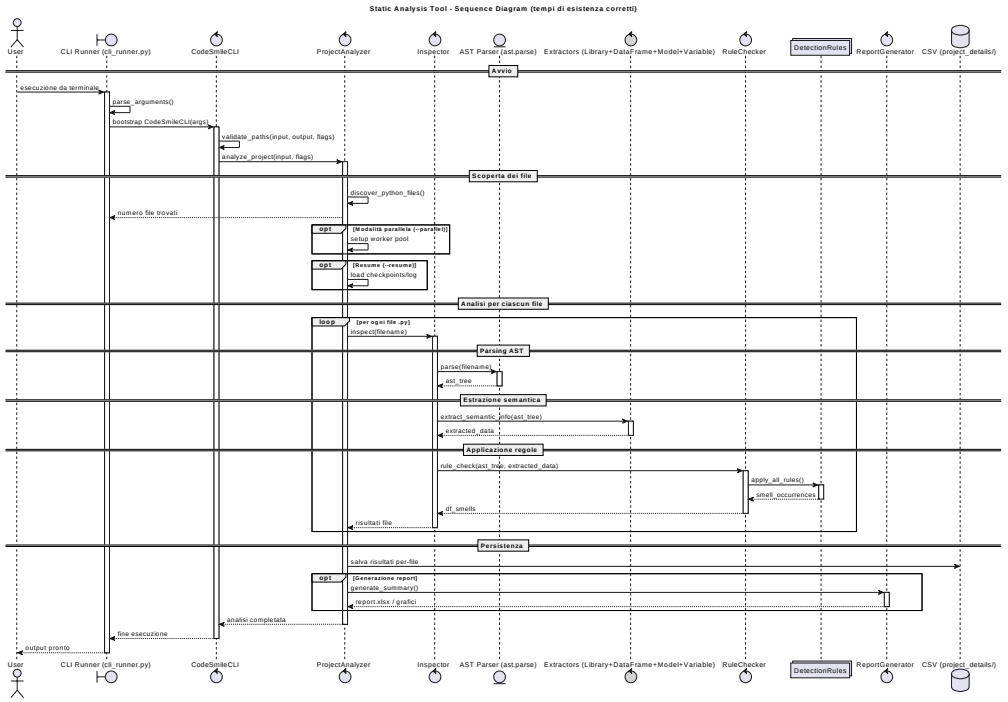

Fig. 5.1. *Sequence Diagram* CLI.

5.2 Flusso di esecuzione tramite GUI

Il flusso di esecuzione tramite interfaccia grafica consente all'utente di avviare l'analisi statica mediante una GUI locale basata su `tkinter`. L'esecuzione dell'analisi avviene in un thread separato, al fine di mantenere l'interfaccia responsiva durante l'elaborazione.

(1) L'utente avvia l'applicazione eseguendo `gui_runner.py`, che inizializza l'interfaccia grafica principale.
(2) Il componente `CodeSmellDetectorGUI` crea i widget grafici (pulsanti, checkbox, campi di testo) e configura il reindirizzamento dello standard output verso la GUI tramite `TextboxRedirect`.
(3) L'utente configura i parametri di esecuzione (directory di input e output, flag di parallelizzazione e resume); la GUI valida i percorsi selezionati.
(4) Alla pressione del pulsante `Run Analysis`, la GUI avvia l'analisi statica su un thread separato per evitare il blocco dell'interfaccia.
(5) Il thread invoca il metodo di analisi del `ProjectAnalyzer`, che esplora ricorsivamente la directory di input per individuare i file Python da analizzare.
(6) Per ciascun file Python individuato, il `Inspector` costruisce l'AST ed estrae le informazioni semantiche rilevanti (variabili, librerie, modelli e operazioni).
(7) Il `RuleChecker` applica l'insieme delle detection rules all'AST analizzato, producendo un DataFrame contenente gli smell rilevati per il singolo file.

(8) I risultati dell'analisi vengono progressivamente salvati in formato CSV all'interno della directory di output del progetto.
(9) Al termine dell'analisi di tutti i file, il thread segnala la conclusione del processo e la GUI aggiorna lo stato dell'interfaccia, riabilitando i controlli e mostrando il percorso dei risultati generati.

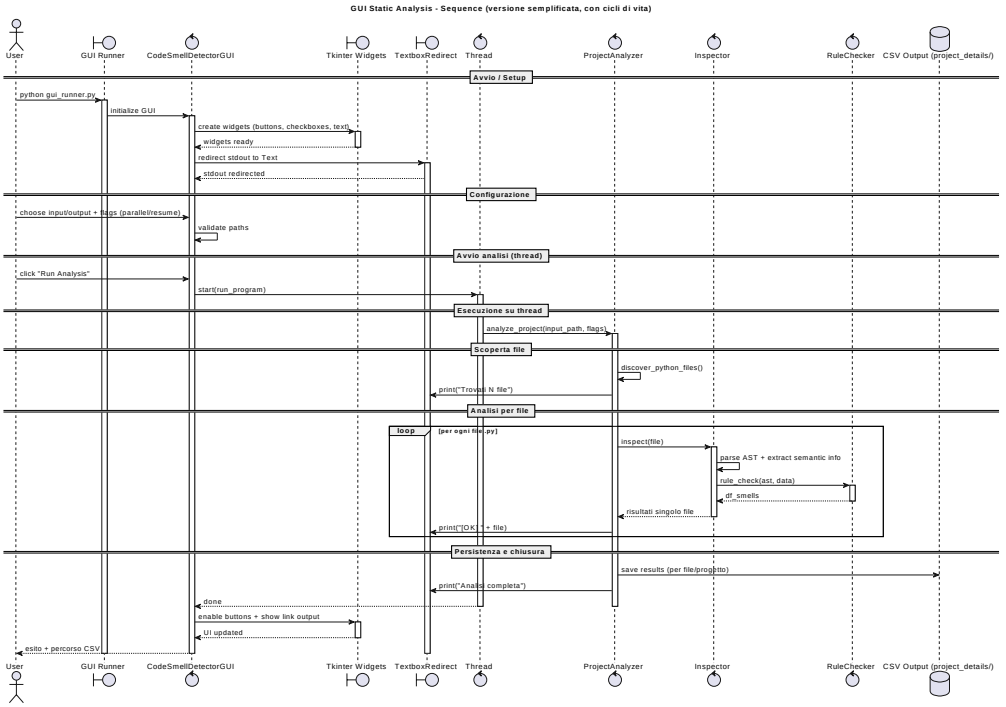

Fig. 5.2. *Sequence Diagram* GUI.

5.3 Flusso di esecuzione tramite Web Application

Il flusso di esecuzione della Web Application descrive l'interazione tra utente, frontend web e backend a microservizi per l'analisi dei code smells e la generazione dei report. Il processo supporta sia l'analisi AI-based sia l'analisi statica, selezionabili dall'utente.

(1) L'utente accede alla Web Application e apre la pagina di upload del codice Python tramite il frontend Next.js.
(2) Il frontend renderizza l'interfaccia di caricamento, consentendo la selezione del file `.py` e della modalità di analisi (AI-based o Static Analysis).
(3) Dopo la validazione dell'input, il frontend invia una richiesta POST al `gateway` tramite l'endpoint `/api/detect_smell_(ai|static)`, includendo lo snippet di codice.
(4) Il `gateway` riceve la richiesta e la inoltra al servizio backend appropriato in base alla modalità selezionata.

(5) In modalità *AI-based*, il servizio `aiservice` costruisce il prompt e interroga il modello LLM fine-tuned tramite Ollama, parsificando l'output per estrarre i code smells rilevati.
(6) In modalità *Static Analysis*, il servizio `staticanalysis` analizza lo snippet creando un file temporaneo, applica le detection rules tramite `Inspector` e raccoglie gli smell individuati.
(7) Il servizio backend restituisce al `gateway` l'elenco degli smell rilevati, che viene a sua volta inoltrato al frontend.
(8) Il frontend visualizza i risultati all'utente, mostrando i code smells rilevati con informazioni descrittive e riferimenti alle linee di codice.
(9) In modo opzionale, l'utente richiede la generazione di un report aggregato; il frontend invia una richiesta POST a `/api/generate_report` tramite il `gateway`.
(10) Il `report service` aggrega i risultati delle analisi, genera i dati strutturati del report e li restituisce al frontend, che rende disponibile il download o la visualizzazione del report.

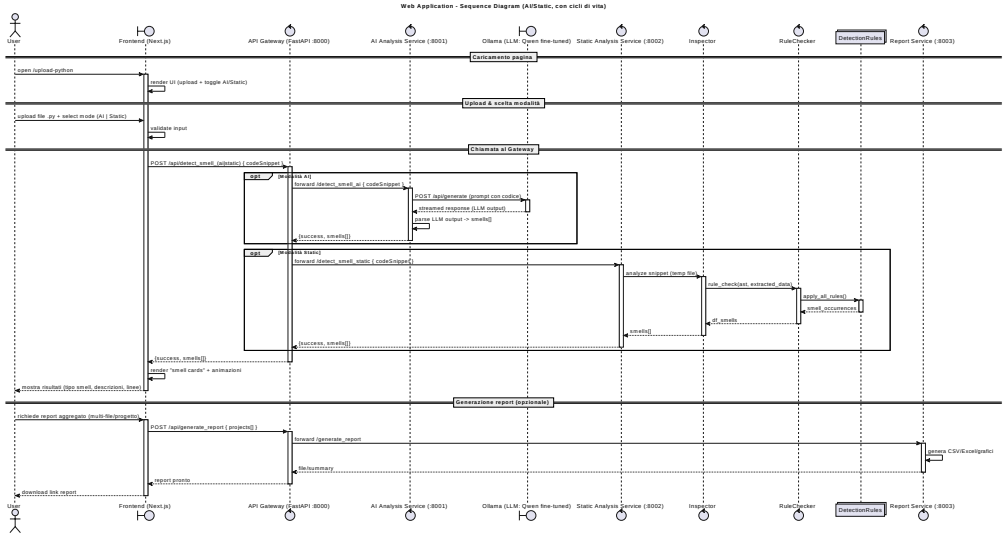

Fig. 5.3. *Sequence Diagram* Web Application.

Riferimenti bibliografici

[1] Giuseppe Recupito, Giulia Giordano, Filomena Ferrucci, et al. 2025. When code smells meet ML: on the lifecycle of ML-specific code smells in ML-enabled systems. *Empirical Software Engineering* 30, 139 (2025). doi:10.1007/s10664-025-10676-4