Università degli Studi di Salerno
Corso di Ingegneria del Software: Tecniche Avanzate

CodeSmile
Report Iniziale
Versione 1.0

Team:
Antonio Caiazzo
Emanuele Iovane
Salvatore Di Martino

Progetto: CodeSmile	Versione: 1.0
Documento: Report Iniziale	Data: 13/11/2025

Partecipanti:

Nome	**Matricola**
Antonio Caiazzo	NF22500205
Salvatore Di Martino	NF22500114
Emanuele Iovane	NF22500162

Scritto da:	Antonio Caiazzo, Salvatore Di Martino, Emanuele Iovane

Indice

1. Descrizione del sistema

1.1. Introduzione

L'evoluzione dei sistemi software basati su tecniche di *Machine Learning* (ML-enabled systems) ha introdotto nuove sfide in termini di qualità, manutenibilità e affidabilità del codice. Questi sistemi, caratterizzati da una forte dipendenza da pipeline di dati, modelli e configurazioni sperimentali, tendono a generare forme specifiche di debito tecnico note come **AI-Specific Technical Debt**. In tale contesto si inserisce **CodeSmile**, uno strumento di analisi statica e AI-based sviluppato per individuare e classificare difetti ricorrenti nei progetti che integrano componenti di Machine Learning, con l'obiettivo di supportare la qualità del codice e ridurre l'accumulo di debito tecnico.

1.1.1. AI-specific Technical Debt e ML Code Smells

Il concetto di **Technical Debt**, introdotto da Cunningham (1992), rappresenta il compromesso tra scelte di implementazione rapide e la qualità a lungo termine di un sistema software.

Nei sistemi che incorporano moduli di intelligenza artificiale, questa problematica assume una forma particolare, denominata **AI-Specific Technical Debt**. Essa si manifesta attraverso pratiche di sviluppo subottimali che compromettono la robustezza e la riproducibilità delle pipeline di apprendimento automatico, generando inefficienze legate ai dati, ai modelli e alle configurazioni.

Un aspetto chiave dell'AI Technical Debt è rappresentato dai **Machine Learning Specific Code Smells (ML-CSs)**: pattern ricorrenti di cattive pratiche implementative nelle pipeline di ML, che possono degradare le prestazioni e la manutenibilità del sistema. I *ML Code Smells* si distinguono dai code smells tradizionali in quanto derivano dall'uso improprio di librerie come *Pandas*, *TensorFlow* e *PyTorch*, e incidono direttamente sul comportamento del modello o sulla gestione dei dati.

Lo strumento CodeSmile è stato progettato per rilevare automaticamente **16 differenti ML-specific Code Smells**, suddivisi in due categorie principali:

- **API-Specific Smells**, legati all'uso scorretto di funzioni o metodi propri delle librerie ML;
- **Generic Smells**, che rappresentano inefficienze generali nelle strutture di codice ma con impatti diretti sulla pipeline di ML.

Queste tipologie di smells rappresentano diversi punti critici della pipeline ML, dalla **fase di preprocessing dei dati**, all'**addestramento del modello**, fino alla **gestione della memoria** e alla **riproducibilità sperimentale**. La loro individuazione automatica consente di analizzare in modo sistematico il debito tecnico nei sistemi ML-enabled, supportando strategie di miglioramento continuo della qualità.

1.1.2. Tool CodeSmile: Scopo, Obiettivi e Contesto

CodeSmile è uno strumento di analisi statica e AI-based progettato per individuare automaticamente *ML-specific Code Smells* all'interno di progetti Python che utilizzano librerie di Machine Learning. Il tool nasce nell'ambito di una ricerca empirica su larga scala volta a comprendere la diffusione, l'introduzione e la rimozione dei code smells nei sistemi ML-enabled.

L'idea alla base del progetto è che i tradizionali strumenti di *software quality assurance* non siano in grado di cogliere appieno le problematiche introdotte dalle complesse pipeline ML. In questo scenario, **CodeSmile** si propone come un *AI Technical Debt Detector*, capace di misurare in modo oggettivo il debito tecnico legato alle librerie di ML.

Gli **obiettivi principali** del tool sono:

- **Rilevamento automatico** dei 16 ML-specific code smells mediante regole statiche derivate dalla letteratura e da esempi concreti;
- **Supporto alla ricerca empirica**, analizzare il ciclo di vita dei code smells (introduzione, rimozione, sopravvivenza), attraverso dataset di grandi dimensioni, comprendenti oltre 400.000 commit, provenienti da 337 progetti open source;
- **Contributo alla qualità del software**, facilitando la manutenzione e la leggibilità dei progetti che integrano componenti ML;
- **Riduzione e gestione del debito tecnico legato all'intelligenza artificiale**, con particolare riferimento alle seguenti categorie individuate:
    - **Data Debt:** problematiche legate alla qualità, gestione o trasformazione dei dati di input e dei DataFrame;
    - **Model Debt:** inefficienze strutturali nei modelli ML, come l'uso errato delle API di PyTorch o TensorFlow, la mancata inizializzazione corretta dei gradienti o la gestione impropria della memoria;
    - **Configuration Debt:** configurazioni non ottimali o non riproducibili, derivanti da parametri non esplicitamente dichiarati o da algoritmi non deterministici;
    - **Ethics Debt:** debito etico.
- **Integrazione nel contesto SQA4AI (Software Quality Assurance for AI)**, come strumento di supporto alla valutazione e alla prevenzione dei difetti legati all'uso di modelli di apprendimento automatico.

1.1.3. Analisi Preliminare del Sistema

Il funzionamento di CodeSmile si basa su due approcci complementari di analisi statica: **AST-Based Detection** e **Rule-Based Detction**.

Approccio AST-Based

L'analisi AST (*Abstract Syntax Tree*) consente di rappresentare la struttura sintattica del codice sorgente sotto forma di albero. CodeSmile utilizza questa tecnica per:

- analizzare la struttura dei file Python;
- identificare definizioni di variabili, funzioni e chiamate a metodi;
- riconoscere le dipendenze tra oggetti e librerie ML (Pandas, TensorFlow, PyTorch).

Durante la fase di parsing, il sistema costruisce un albero sintattico per ciascun file e applica un processo di estrazione semantica delle entità rilevanti: variabili, DataFrame, tensori, modelli e operazioni API. L'obiettivo è isolare le porzioni di codice potenzialmente problematiche per verificarne la conformità alle regole definite.

Approccio Rule-Based

A valle dell'analisi sintattica, il sistema applica **regole di rilevamento** (rule-based detection) derivate dalla letteratura e dalle specifiche API ML. Ogni regola corrisponde a una condizione logica che descrive il comportamento anomalo da intercettare (es. mancanza di *inplace=True*, uso scorretto di *np.dot*, assenza di *zero_grad()*, ecc.).
Il rilevamento combina quindi:

- **analisi strutturale**, per individuare pattern sintattici ricorrenti;
- **analisi semantica**, per validare il contesto e ridurre i falsi positivi.

Questa doppia logica consente a CodeSmile di riconoscere non solo gli errori espliciti nel codice, ma anche pratiche di programmazione inefficienti o non deterministiche.

2. Struttura e Architettura del Sistema

2.1. Architettura

CodeSmile adotta un'organizzazione a oggetti con pipeline di analisi statica che combina **traversing dell'AST** (Abstract Syntax Tree) e **regole di rilevamento** (rule-based) per identificare ML-specific code smells in file o progetti *Pyhton*.

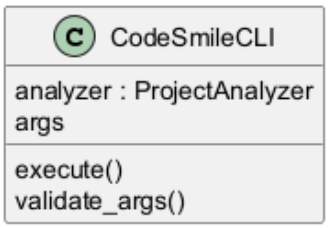
2.2.Package del Sistema

2.2.1. cli

Scopo: Interfaccia a riga di comando per avviare e configurare l'analisi.
Moduli principali:

- *cli_runner.py*: definisce la classe *CodeSmileCLI*. Gestisce l'interfaccia a linea di comando, per l'esecuzione di CodeSmile.
  In particolare gestisce parametri di input/output, esecuzione sequenziale/parallela, ripresa delle analisi e multi-progetto; orchestra l'esecuzione tramite *ProjectAnalyzer* e salva i risultati. Contiene l'entrypoint *main()* che istanzia la CLI.

Di seguito la definizione della classe *CodeSmileCLI*, contenuta nel modulo *cli_runner.py*:

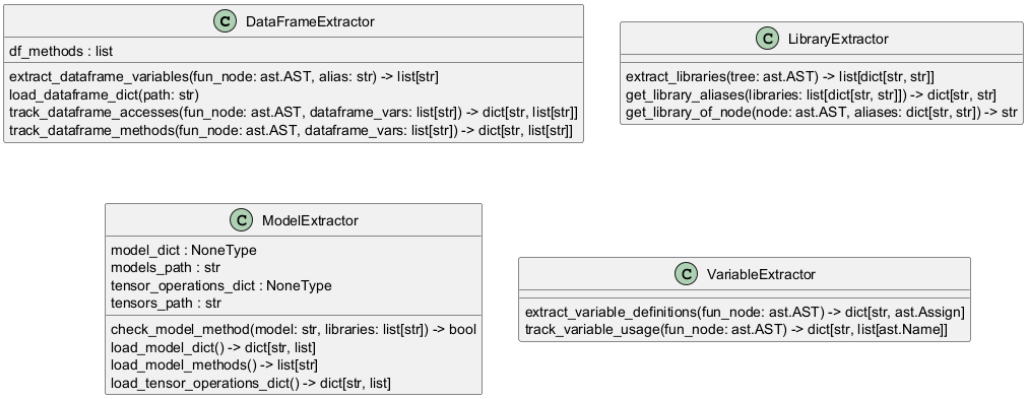


2.2.2. code_extractor

Scopo: Racchiude moduli per l'estrazione delle strutture semantiche, di contesto e costrutti tipici, a supporto della detection.
Moduli principali:

- *dataframe_extractor.py*: individua *DataFrame Pandas*, metodi richiamati e colonne accessibili via Abstract Syntax Trees (AST); usando un dizionario di metodi per affinare il rilevamento da CSV.
- *library_extractor.py*: rileva librerie importate e alias, mappa chiamate a funzione e metodi alla libreria di riferimento tramite AST.
- *model_extractor.py*: gestisce le informazioni su modelli e operazioni su tensori . Carica dizionari di modelli/tensori da CSV, filtra operazioni multi-tensore e verifica l'appartenenza di metodi a librerie note.
- *variable_extractor.py*: estrae definizioni/usi delle variabili, dal codice, collegandole ai nodi dell' AST per l'analisi statica.

Di seguito la definizione dei diversi moduli contenuti nel package *code_extractor*:



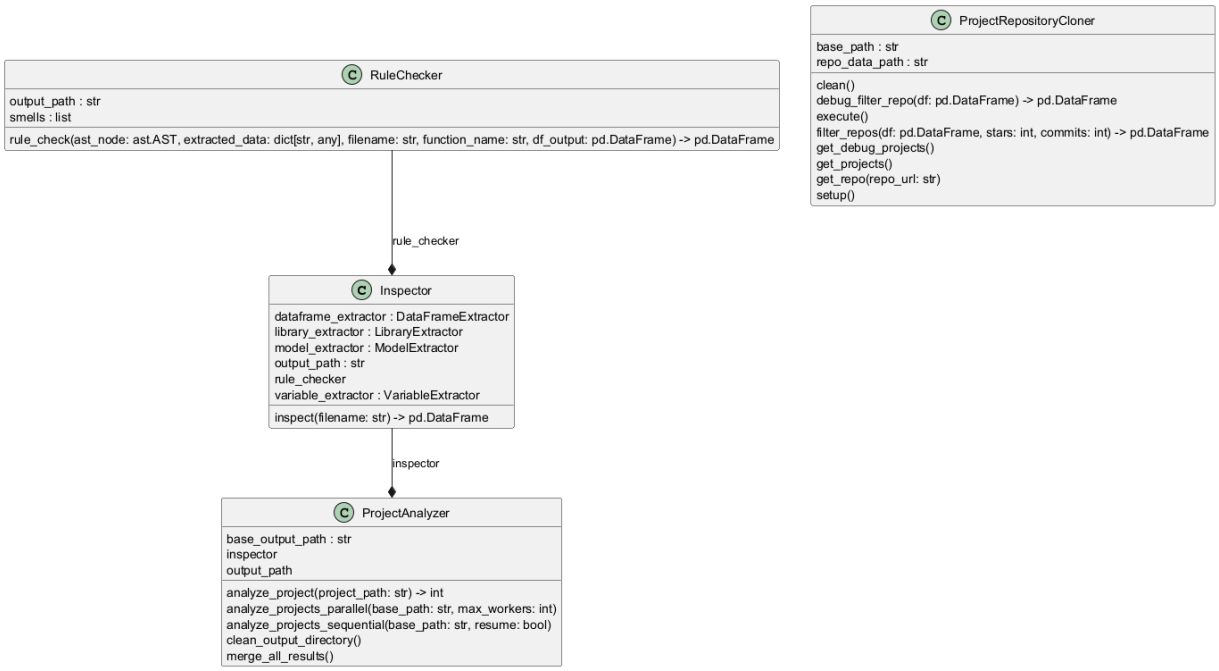
2.2.3. components

Scopo: Contiene i componenti core per: l'analisi dei file e dei progetti e l'applicazione delle regole.

Moduli principali:

- *inspector.py*: apre ciascun file, costruisce l'AST, coordina gli estrattori e passa i dati a *RuleChecker* producendo *DataFrame* di smell.
- *project_analyzer.py*: gestisce l'analisi di interi progetti, elaborando i file tramite esecuzione sequenziale o parallela dell'analisi e si occupa della generazione dei risultati tramite report.
- *project_repository_cloner.py*: clona e gestisce i repositories dal dataset ML, con filtri per diverse metriche e si occupa del setup e del cleanup dei progetti.
- *rule_checker.py*: si occupa dell'applicazione delle regole di rilevamento (API-specifiche di Machine Learning e generali) sull'AST, aggregando i riscontri in DataFrame.

Di seguito la definizione dei moduli principali del package *components*:



2.2.4. detection_rules

Scopo: Contiene la gerarchia delle regole AST per individuare smell specifici di librerie AI e pattern generici, basata sulla classe astratta *Smell.* Sono presenti due sotto-package: **api_specific_smells** (Regole di rilevamento di errori nelle API di Machine Learning)) e **generic_smells** (Regole di rilevamento per code smells comuni). Di seguito i file contenuti in questi package vengono specificati nel formato seguente: *sotto-package/file.*

Moduli principali:

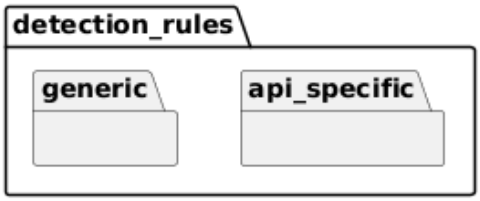
- *smell.py*: superclasse che definisce la struttura comune di uno smell.

- *api_specific/chain_indexing_smell.py*: rileva chained indexing su DataFrame Pandas.
- *api_specific/dataframe_conversion_api_misused.py: segnala* l'uso dell'attributo *values* sui *DataFrame* in *Pandas*.
- *api_specific/gradients_not_cleared_before_backward_propagation.py*: verifica che *zero_grad*() preceda *backward*() nei loop di addestramento *PyTorch*.
- *api_specific/matrix_multiplication_api_misused.py*: sconsiglia l'uso di *np.dot* per moltiplicazioni matriciali dove è preferibile *np.matmul*.
- *api_specific/pytorch_call_method_misused.py*: intercetta chiamate dirette a forward() nei modelli PyTorch. S8uggerisce di invocare il modello direttamente, tramite la sua istanza.
- *api_specific/tensor_array_not_used.py*: suggerisce *tf.TensorArray* quando un *tf.constant* viene aggiornato in loop tramite *tf.concat*, gestendo meglio le operazioni con *TensorFlow*.

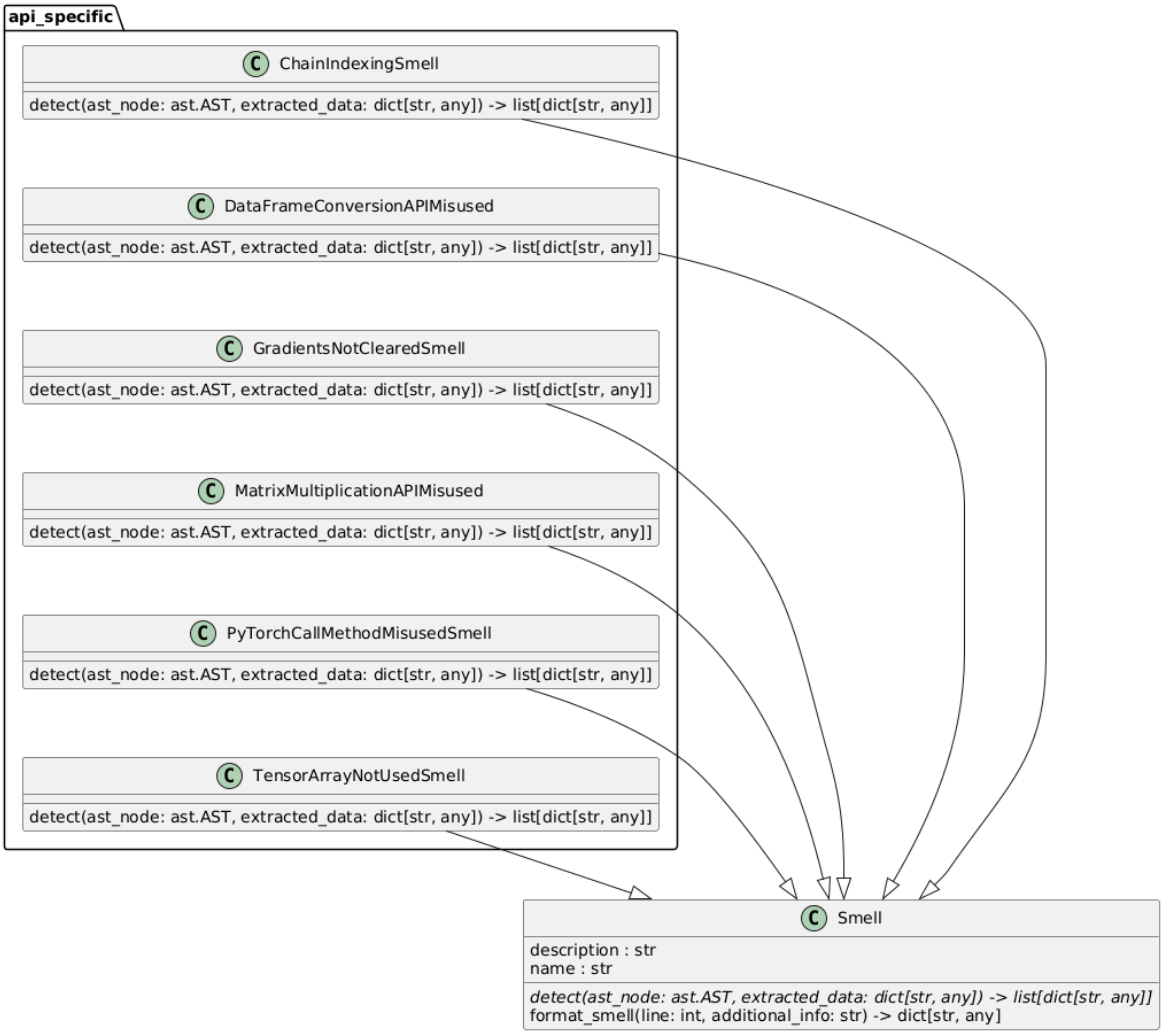
- *generic/broadcasting_feature_not_used.py*: rileva uso inutile di *tf.tile* invece del broadcasting in *TensorFlow*, per la replicazione di tensori.
- *generic/columns_and_datatype_not_explicitly_set.py*: richiede l'esplicita impostazione di dtype per *DataFrame* o per la lettura da *CSV*.
- *generic/deterministic_algorithm_option_not_used.py*: avvisa sull'attivazione di *torch.use_deterministic_algorithms*(*True*) in *PyTorch* per l'impatto sulle prestazioni.
- *generic/empty_column_misinitialization.py*: vieta inizializzare colonne con 0 o stringhe vuote consigliando *NaN*.
- *generic/hyperparameters_not_explicitly_set.py*: segnala modelli istanziati senza l'uso di iperparametri espliciti.
- *generic/in_place_apis_misused.py*: controlla l'uso coerente del parametro *inplace* e l'assegnazione del risultato nelle API *Pandas*, segnalando quando viene omesso o impostato a *false*.
- *generic/memory_not_freed.py*: impone l'uso di *tf.keras.backend.clear_session*() quando si creano modelli in loop, per liberare la memoria esplicitamente.
- *generic/merge_api_parameter_not_explicitly_set.py*: *richiede how/on/validate* nelle chiamate a *merge*() di *Pandas*, per evitare comportamenti anomali.
- *generic/nan_equivalence_comparison_misused.py*: evita confronti diretti con *NaN*, suggerendo *np.isna0n*.
- *generic/unnecessary_iteration.py*: rileva loop/operazioni inefficienti su *DataFrame* (*iterrows/apply/applymap*).

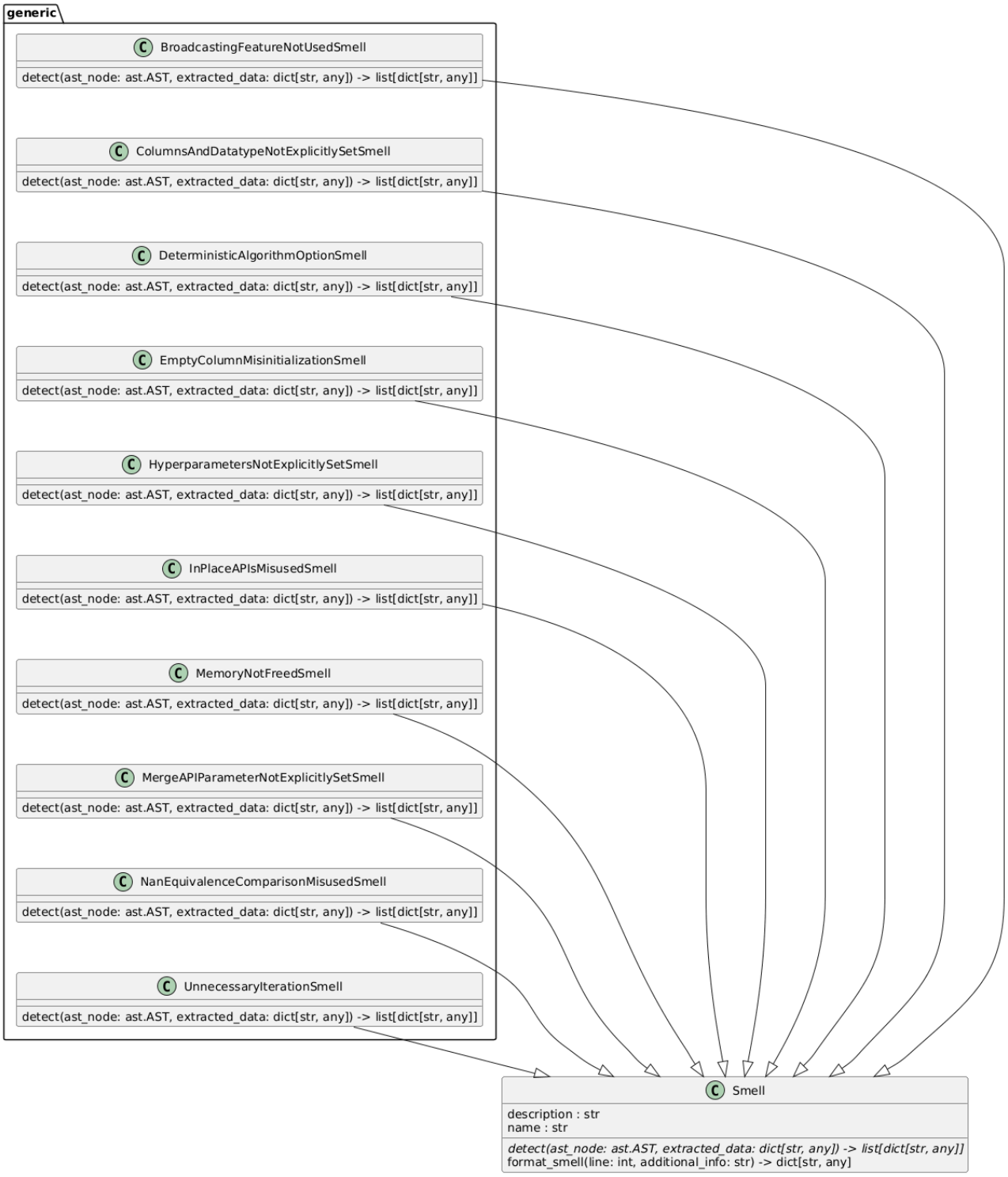
Di seguito, la definizione dei packages e dei moduli presenti nel package *detection_rules*:

Packages



Modules




generic
BroadcastingFeatureNotUsedSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
ColumnsAndDatatypeNotExplicitlySetSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
DeterministicAlgorithmOptionSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
EmptyColumnMisinitializationSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
HyperparametersNotExplicitlySetSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
InPlaceAPIsMisusedSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
MemoryNotFreedSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
MergeAPIParameterNotExplicitlySetSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
NanEquivalenceComparisonMisusedSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
UnnecessaryIterationSmell
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
Smell
description : str
name : str
detect(ast_node: ast.AST, extracted_data: dict[str, any]) -> list[dict[str, any]]
format_smell(line: int, additional_info: str) -> dict[str, any]

2.2.5. gui

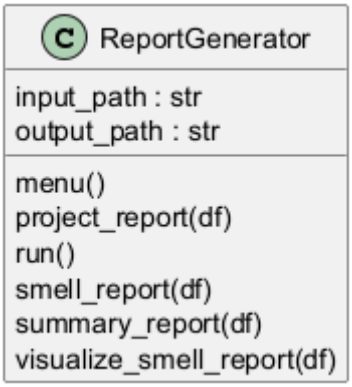
Scopo: Interfaccia grafica per configurare ed eseguire l'analisi con visualizzazione dei log.
Moduli principali:

- *code_smell_detector_gui.py*: costruisce l'interfaccia grafica per l'analisi dei code smells basata su *Tkinter*, per la raccolta di percorsi e parametri, e lancia l'analisi su thread dedicato tramite *ProjectAnalyzer*, permettendo la scelta tra esecuzione parallela e sequenziale.
- *gui_runner.py*: crea l'istanza *Tkinter* e avvia il mainloop tramite l'utilizzo di *CodeSmellDetectorGUI*.
- *textbox_redirect.py*: implementa un writer *io.StringIO* che redirige stdout nel widget *Text* della GUI. Consente la visualizzazione realtime dei diversi log generati dal tool.

2.2.6. report

Scopo: Generazione di report aggregati ed esportazioni.
Moduli principali:

- *report_generator.py*: carica i CSV di dettaglio, produce report globali e per un progetto. Inoltre, esporta un workbook riassuntivo e un grafico a barre, includendo un semplice menu testuale.



2.2.7. utils

Scopo: Fornise utility di file I/O condivise.
Moduli principali:

- *file_utils.py*: classe statica con funzioni per ripulire cartelle, enumerare file *Python*, unire risultati dell'analisi da più file CSV, gestire log sequenziali o sincronizzati tra thread.

2.3. Web App e Microservizi

È disponibile una **Web App** (sviluppata in *React* e *Node.js*) che si appoggia a un **API Gateway** (*FastAPI*) per instradare le richieste verso tre servizi principali. Questo permette di interfacciarsi con il tool tramite un'interfaccia web. L'applicazione è basata su un'architettura a microservizi, esponendo i moduli principali attraverso endpoint. Di seguito vengono mostrati e analizzati i diversi package dell'applicazione.

2.3.1. Package app

Implementa il front-end *Next.js* definisce il layout globale, pagine pubbliche, viste interattive per gli upload di codice o di progetti, e la generazione dei report.

2.3.2. Package components

Raccolta di componenti *React* riusabili per il layout, per la navigazione , input progetto, grafici e widget di supporto.

2.3.3. Package context

Espone un contesto *React* per memorizzare la lista di progetti, aggiornare i dati e condividere gli handler tra le diverse pagine.

2.3.4. Package utils

Wrapper client per le chiamate http al gateway, con gestione degli errori e configurazione comune di *Axios*.

2.3.5. Package types

Tipizzazioni condivise tra front-end e servizi(smell, risposte API, contesto *React*).

2.3.6. Package gateway

API Gateway FastAPI che espone endpoint unificati verso i servizi AI, statico e report, gestendo il *CORS* e l'inoltro delle richieste. Contiene il modulo *main.py* che configura *FastAPI* con middleware *CORS,* instrada le API verso i micro-servizi, gestendo anche timeout ed errori.

2.3.7. Package services

Definisce tre moduli principali costruiti come micro-servizi Python indipendenti (FastAPI), che implementano le API di: analisi AI, analisi statica e reportistica. Pensati sia per essere eseguiti localmente che in container. Di seguito vengono descritti i diversi servizi della web app, con i loro componenti.

AI service
Questo modulo si occupa dell'analisi dei code smell utilizzando AI.

- *main.py*: istanzia l'app FastAPI "AI Analysis Service" e include il router dedicato.
- *routers/detect_smell.py*: definisce gli endopoint per l'analisi. Valida l'input con AST, invoca il modello AI e restituisce *DetectSmellResponse*, loggando eventuali errori.
- *schemas/requests.py*: definisce lo schema per le richieste e il formato dei dati in entrata.
- *schemas/responses.py*: definisce lo schema per le risposte fornite dal servizio, includendo i dettagli sull'analisi tramite AI.
- *utils/model.py*: client verso Ollama (streaming), normalizza la risposta e filtra smell riconosciuti.

Static analysis service
Questo modulo si occupa del rilevamento dei code smells tramite analisi statica.

- *main.py*: istanzia l'app *FastAPI* "Static Analysis Service" con router registrato.
- *routers/detect_smell.py*: espone l'endpoint */detect_smell_static*, Riceve codice in input, esegue l'analisi statica e restituisce i code smells che vengono rilevati.
- *schemas/requests.py*: definisce lo schema per le richieste e il formato dei dati in entrata.
- *schemas/responses.py*: definisce lo schema per le risposte fornite dal servizio, includendo i dettagli sull'analisi tramite analisi statica
- *utils/static_analysis.py*: configura il processo di analisi statica tramite l'*Inspector*, crea un file temporaneo dal codice ricevuto, esegue l'analisi e restituisce i risultati, gestendo cleanup ed errori.

Report Service

Questo modulo gestisce l'aggregazione degli esiti, e la generazione dei report per la visualizzazione e l'esportazione.

- *main.py*: istanzia l'app *FastAPI* "Report Service" con router registrato.
- *routers/report.py*: espone l'endpoint */generate_report* che aggrega smells e gestisce eccezioni ed errori.
- *schemas/requests.py*: definisce lo schema per le richieste e il formato dei dati in entrata, includendo le informazioni sui progetti analizzati e gli smells rilevati.
- *schemas/responses.py*: definisce lo schema per le risposte fornite dal servizio, fornendo dati aggregati per la creazione dei grafici e delle statistiche.
- *utils/report_generator.py*: contiene la logica per l'aggregazione dei risultati e per la generazione dei report. Inoltre, costruisce *DataFrame Pandas* dagli smells e restituendo conteggi aggregati.

3. Flusso d'Uso e di Funzionamento di CodeSmile

3.1. Utilizzo

Si illustra di seguito il flusso di utilizzo del tool nelle tre modalità disponibili

3.1.1. Utilizzo tramite Linea di Comando (CLI)

La CLI consente di analizzare singoli file o interi progetti, impostando directory di input/output e scegliendo l'esecuzione sequenziale o parallela per velocizzare l'elaborazione su più file. Questa modalità è mirata a sviluppatori/ricercatori che necessitano di analisi rapide e ripetibili. Il runner orchestra l'analisi di progetto e salva i risultati, supportando anche la ripresa di analisi interrotte e l'elaborazione multi-progetto.

3.1.2. Utilizzo tramite Interfaccia Grafica (GUI)

La GUI fornisce un'interfaccia desktop per selezionare parametri, configurare le cartelle di input/output, avviare l'analisi con parallelismo e resume, visualizzando log in tempo reale nella finestra dell'applicazione. È pensata per un'interazione più guidata rispetto alla CLI.

3.1.3. Utilizzo tramite Web App

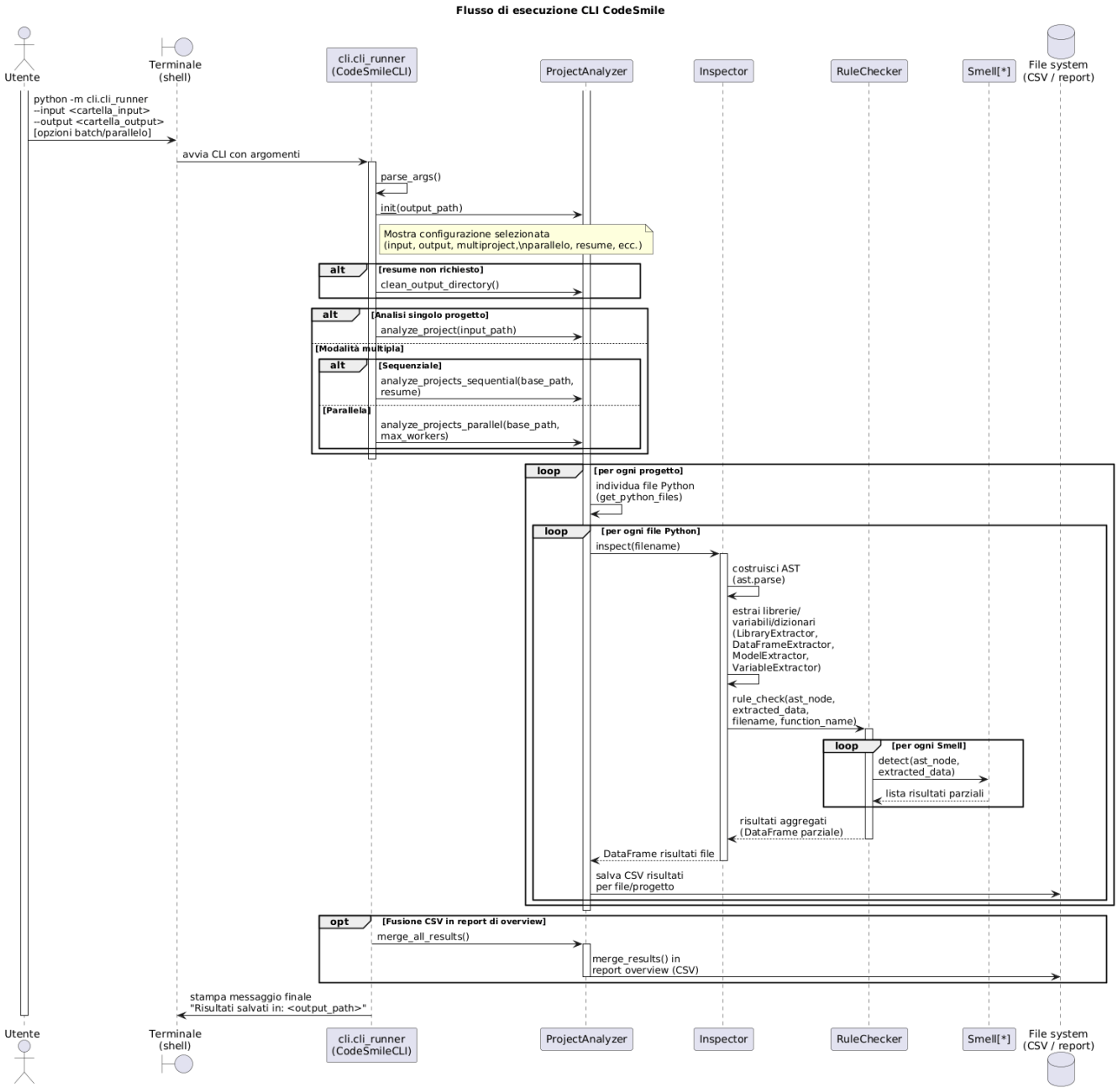
La WebApp, permette di caricare file o progetti dal browser, visualizzare gli esiti di rilevamento e esportare i report, senza scrivere comandi. È ideale per profili meno tecnici o contesti in cui serve un'interazione visuale e collaborativa. L'interfaccia comunica con un gateway basato su FastAPI che espone servizi dedicati per analisi AI, analisi statica e generazione report.

3.2. Funzionamento

Si illustra di seguito il flusso di funzionamento semplificato delle tre modalità di funzionamento del tool CodeSmile.

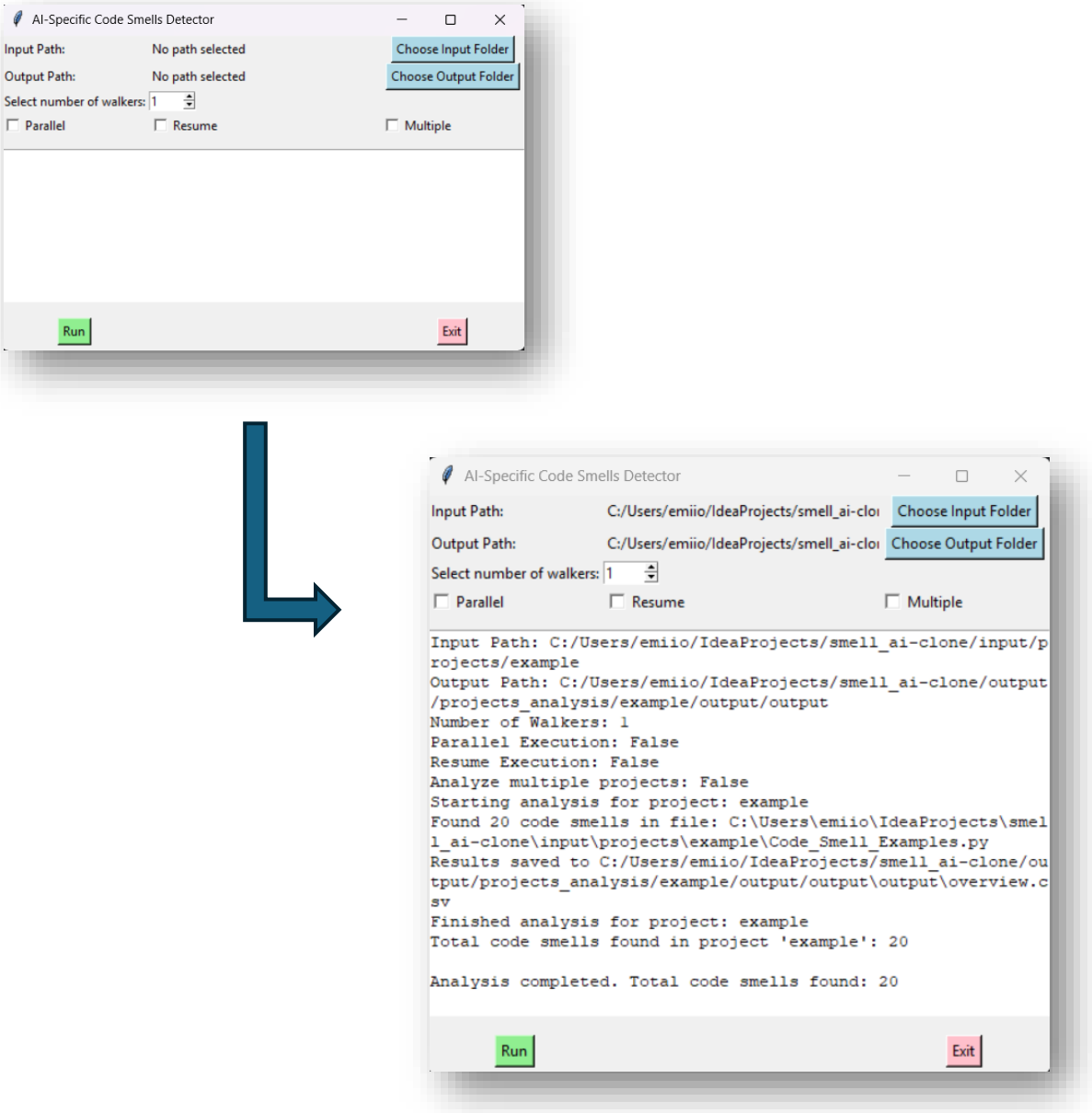
3.2.1. Flusso CLI

1. L'utente avvia l'analisi da terminale eseguendo *python -m cli.cli_runner* e passando cartelle di input/output ed eventuali opzioni per il batch o l'elaborazione parallela.
2. Il modulo CLI effettua il parsing degli argomenti, crea un *ProjectAnalyzer* puntando alla directory di output e mostra la configurazione selezionata prima di procedere.
3. Se non è richiesto il *resume*, l'output viene pulito; quindi la CLI richiama l'analisi di un singolo progetto o, in caso di modalità multipla, attiva la scansione sequenziale o parallela dei sottoprogetti, demandando il lavoro al *ProjectAnalyzer*.
4. Per ogni file Python individuato, il *ProjectAnalyzer* usa l'*Inspector* per costruire l'AST, estrarre librerie/variabili/dizionari e applicare tutte le regole, aggregando i risultati in *DataFrame* che vengono salvati su disco.
5. A fine corsa, l'analizzatore può fondere i CSV dettagliati dei diversi progetti in un unico report di overview e il processo CLI termina segnalando dove sono stati salvati i risultati.


Flusso di esecuzione CLI CodeSmile
Utente
Terminale (shell)
cli.cli_runner (CodeSmileCLI)
ProjectAnalyzer
Inspector
RuleChecker
Smell[*]
File system (CSV / report)
python -m cli.cli_runner --input <cartella_input> --output <cartella_output> [opzioni batch/parallelo]
avvia CLI con argomenti
parse_args()
init(output_path)
Mostra configurazione selezionata (input, output, multiproject,\nparallelo, resume, ecc.)
alt [resume non richiesto]
clean_output_directory()
alt [Analisi singolo progetto]
analyze_project(input_path)
[Modalità multipla]
alt [Sequenziale]
analyze_projects_sequential(base_path, resume)
[Parallela]
analyze_projects_parallel(base_path, max_workers)
loop [per ogni progetto]
individua file Python (get_python_files)
loop [per ogni file Python]
inspect(filename)
costruisci AST (ast.parse)
estrai librerie/ variabili/dizionari (LibraryExtractor, DataFrameExtractor, ModelExtractor, VariableExtractor)
rule_check(ast_node, extracted_data, filename, function_name)
loop [per ogni Smell]
detect(ast_node, extracted_data)
lista risultati parziali
risultati aggregati (DataFrame parziale)
DataFrame risultati file
salva CSV risultati per file/progetto
opt [Fusione CSV in report di overview]
merge_all_results()
merge_results() in report overview (CSV)
stampa messaggio finale "Risultati salvati in: <output_path>"

3.2.2. Flusso GUI

1. L'utente lancia python *-m gui.gui_runner*, ottenendo una finestra *Tkinter* che permette di scegliere cartelle di input/output, numero di worker e flag per parallelismo, resume e batch.
2. L'interfaccia reindirizza stdout in un widget testuale tramite *TextBoxRedirect*, così ogni log dell'analisi appare in tempo reale all'interno della GUI.
3. Premendo "Run", la GUI valida i percorsi e avvia un thread che istanzia *ProjectAnalyzer*, applica le stesse logiche della CLI (pulizia output, analisi singola o multipla, modalità parallela) e stampa su schermo conteggi, errori ed esito finale.
4. il feedback testuale resta visibile finché l'utente non chiude la finestra, facilitando l'utilizzo in contesti non a riga di comando.



3.2.3. Flusso Web App

1. Nel front-end *Next.js* (es. pagina “Upload Python”) l’utente carica uno snippet .py, o un progetto completo, sceglie il motore (AI o statico), avvia l’analisi e osserva stato/progresso e risultati direttamente nella pagina.
2. La pagina richiama le funzioni client *detectAi* o *detectStatic*, che inviano il codice al gateway specifico tramite richieste HTTP con timeout prolungato e gestiscono eventuali errori mostrando toast di notifica.
3. Il *gateway FastAPI* riceve le richieste, applica *CORS* e le inoltra ai micro-servizi specializzati (AI, statico o reportistica) usando *httpx*, restituendo al front-end la risposta unificata.
4. Nel caso dell’analisi statica, il micro-servizio crea un file temporaneo con lo snippet, usa *l’Inspector* condiviso per eseguire le stesse regole del tool desktop e converte il *DataFrame* in oggetti Smell prima di rispondere al gateway.
5. Una volta ricevuta la risposta, la pagina aggiorna l’interfaccia con l’elenco dei code smell o con un messaggio di successo, mantenendo i dati nel contesto client per eventuali workflow successivi come la generazione di report.