
Università degli Studi di Salerno

Dipartimento di Informatica

---

Corso di Ingegneria del Software: Tecniche Avanzate

# CodeSmile


## Initial Report Document

Versione 1.0

## Team

Choaib Goumri
Mattia Gallucci

---

Anno Accademico 2025–2026

# Indice


**1 Descrizione del sistema** **1**
- 1.1 Introduzione . . . 1
  - 1.1.1 L'emersione dell'AI-Specific Technical Debt . . . 1
  - 1.1.2 I Machine Learning Specific Code Smells (ML-CSs) . . . 1
  - 1.1.3 La soluzione proposta: CodeSmile . . . 2

**2 Architettura del sistema** **3**
- 2.1 Struttura e Architettura del Sistema CodeSmile . . . 3
  - 2.1.1 Package del Sistema . . . 4
  - 2.1.2 Web App e Microservizi . . . 8

**3 Funzionamento del tool** **9**
- 3.1 Modalità di Utilizzo . . . 9
  - 3.1.1 Utilizzo tramite Linea di Comando (CLI) . . . 9
  - 3.1.2 Utilizzo tramite Interfaccia Grafica (GUI) . . . 9
  - 3.1.3 Utilizzo tramite Web App . . . 10
  - 3.1.4 Flusso di Funzionamento Dettagliato . . . 10

# 1 Descrizione del sistema

## 1.1 Introduzione

### 1.1.1 L'emersione dell'AI-Specific Technical Debt

L'integrazione pervasiva di tecnologie di intelligenza artificiale nello sviluppo software ha generato una nuova categoria di applicazioni, note come *ML-enabled systems*, che pongono sfide inedite rispetto alla programmazione tradizionale. La natura di questi sistemi, che intreccia in modo indissolubile codice sorgente, pipeline di dati e modelli predittivi, crea un terreno particolarmente fertile per problematiche di manutenzione specifiche.

Spesso, le stringenti esigenze di *time-to-market* inducono i team di sviluppo a scelte implementative rapide ma poco lungimiranti. Tali compromessi danno origine a un fenomeno critico definito **AI-Specific Technical Debt**, un debito tecnico peculiare che, se non gestito, rischia di compromettere irreversibilmente:

- La stabilità del sistema;
- La riproducibilità degli esperimenti;
- L'evoluzione futura del software.

### 1.1.2 I Machine Learning Specific Code Smells (ML-CSs)

In questo scenario, la necessità di rilasciare prodotti in tempi rapidi porta spesso all'adozione di pratiche di implementazione subottimali che si manifestano sotto forma di **Machine Learning Specific Code Smells (ML-CSs)**.

Questi difetti si distinguono dai code smell tradizionali per le seguenti caratteristiche chiave:

- **Origine specifica:** Derivano dall'uso improprio di librerie dedicate come *Pandas*, *TensorFlow* e *PyTorch*.
- **Impatto diretto:** Influiscono direttamente sul comportamento del modello o sulla gestione dei dati, degradando prestazioni e robustezza nel lungo periodo.
- **Natura architetturale:** Rappresentano pattern ricorrenti di cattiva progettazione all'interno delle pipeline di ML.

### 1.1.3 La soluzione proposta: CodeSmile

Il progetto **CodeSmile** nasce per rispondere a queste criticità, proponendosi come uno strumento di analisi statica (*AI Technical Debt Detector*) progettato per individuare e classificare automaticamente i difetti nei progetti Python che integrano componenti di Machine Learning.

L'obiettivo primario del sistema è ridurre l'accumulo di debito tecnico, focalizzandosi su quattro categorie principali :

- **Data Debt:** Problematiche legate alla qualità, gestione o trasformazione dei dati.
- **Model Debt:** Inefficienze strutturali nei modelli (es. gestione scorretta della memoria).
- **Configuration Debt:** Derivante da configurazioni non ottimali o non riproducibili.
- **Ethics Debt:** Potenziali rischi etici impliciti nel codice.

Dal punto di vista tecnico, CodeSmile opera attraverso un approccio basato sull'analisi degli *Abstract Syntax Trees* (AST) e su regole di rilevamento predefinite. Il tool è in grado di identificare **16 differenti tipologie** di code smells specifici (suddivisi in *API-Specific* e *Generic*), offrendo un supporto concreto per il monitoraggio della qualità software nei sistemi intelligenti.

# 2 Architettura del sistema

## 2.1 Struttura e Architettura del Sistema CodeSmile

L'architettura di CodeSmile è stata progettata seguendo un approccio **orientato agli oggetti**, finalizzato a garantire modularità, manutenibilità e scalabilità. Il cuore del sistema si basa su una pipeline di analisi statica che combina l'attraversamento degli **AST (Abstract Syntax Trees)** con regole di rilevamento predefinite (rule-based detection) per identificare pattern problematici specifici per il Machine Learning.

Il sistema è suddiviso logicamente in due macro-componenti: il core del tool (organizzato in package Python) e l'infrastruttura Web (composta da Web App e Microservizi).

La fase iniziale del progetto è stata dedicata all'analisi approfondita del tool attraverso un processo di **reverse engineering**. Per ricostruire accuratamente l'architettura del sistema e i suoi flussi logici, sono stati prodotti diagrammi UML automatizzati tramite **Pyreverse**, successivamente raffinati con **Graphviz**.

L'indagine ha portato alla luce un'architettura modulare, organizzata in sottosistemi indipendenti ma strettamente interconnessi. Ciascun modulo presiede a una fase specifica del workflow: dall'estrazione delle informazioni dal codice sorgente, all'identificazione degli smell, fino alla produzione dell'output. Nelle sezioni successive, i singoli componenti verranno descritti nel dettaglio con il supporto dei relativi diagrammi.

### 2.1.1 Package del Sistema

Il backend del sistema è organizzato in package distinti, ognuno con una responsabilità specifica per l'analisi, l'interfaccia utente o la gestione dei dati.

#### Package `cli`

Questo package gestisce l'interfaccia a riga di comando (CLI).

- **Funzione:** Permette di configurare e avviare l'analisi (su singoli file o interi progetti), gestire input/output e supportare l'esecuzione parallela o la ripresa di analisi interrotte.
- **Componenti Chiave:** Il modulo `cli_runner.py` orchestra l'esecuzione interfacciandosi con il `Project Analyzer`.

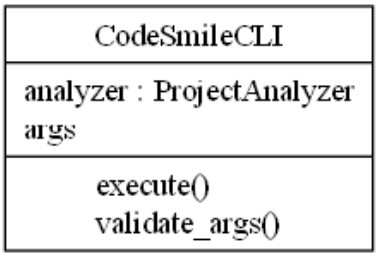


*Figura 2.1: Class Diagram del package cli*

#### Package `code_extractor`

Contiene i moduli dedicati all'estrazione semantica delle strutture del codice tramite analisi AST.

`dataframe_extractor.py`**:** Analizza l'uso dei DataFrame Pandas, tracciando variabili, metodi e accessi alle colonne.

`library_extractor.py`**:** Identifica librerie importate e alias per mappare correttamente le chiamate alle funzioni.

`model_extractor.py`**:** Gestisce informazioni sui modelli ML e operazioni sui tensori, filtrando operazioni e verificando l'appartenenza alle librerie.

`variable_extractor.py`**:** Traccia definizioni e utilizzi delle variabili collegandole ai nodi AST.

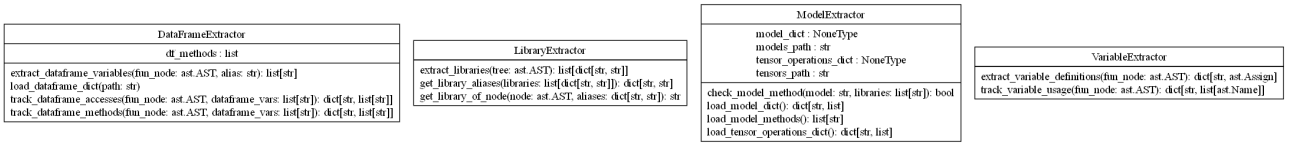


*Figura 2.2: Class Diagram del package code$_e$xtractor*

**Package** `components`

Rappresenta il nucleo operativo per l'analisi e il rilevamento.

`inspector.py`: Coordina gli estrattori per analizzare i singoli file e costruire l'AST.

`project_analyzer.py`: Gestisce l'analisi a livello di progetto (sequenziale o parallela) e la generazione dei risultati.

`rule_checker.py`: Applica le regole di rilevamento (specifiche ML e generali) all'AST.

`project_repository_cloner.py`: Gestisce la clonazione, il filtraggio e la pulizia dei repository da analizzare.

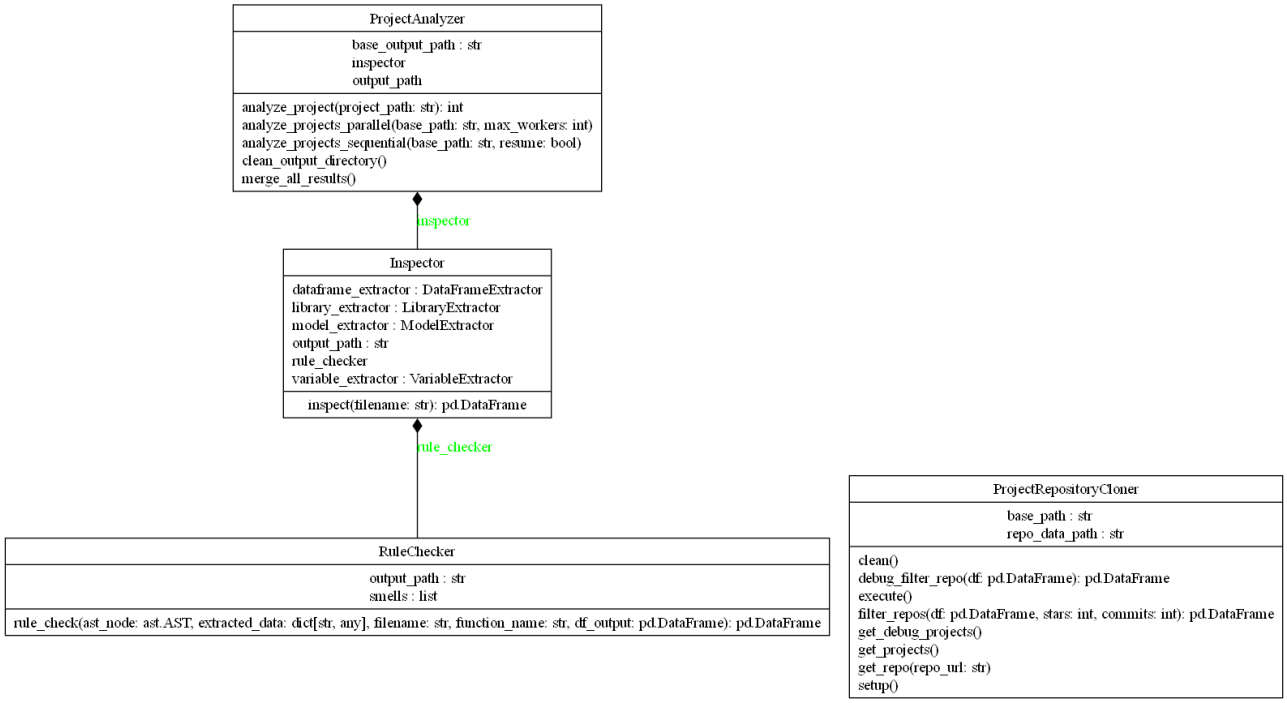


*Figura 2.3: Class Diagram del package components*

**Package** `detection_rules`

Definisce la gerarchia delle regole per i code smells, basata sulla superclasse `Smell`. È suddiviso in due sotto-package:

1. `api_specific_smells`: Regole per errori specifici di librerie come Pandas (es. *chain indexing*), PyTorch (es. mancato `zero_grad`) e TensorFlow.
2. `generic_smells`: Regole per inefficienze generali con impatto ML, come iterazioni non necessarie, inizializzazioni errate o parametri non espliciti.

*Per ragioni di spazio e per garantire una migliore leggibilità, il diagramma completo è disponibile nel repository GitHub del progetto* Link al Repository GitHub.

**Package** `gui`

Gestisce l'interfaccia grafica desktop basata su **Tkinter**.

- Permette la configurazione visuale dei parametri e visualizza i log in tempo reale reindirizzando l'output della console nella finestra dell'applicazione tramite `textbox_redirect.py`.

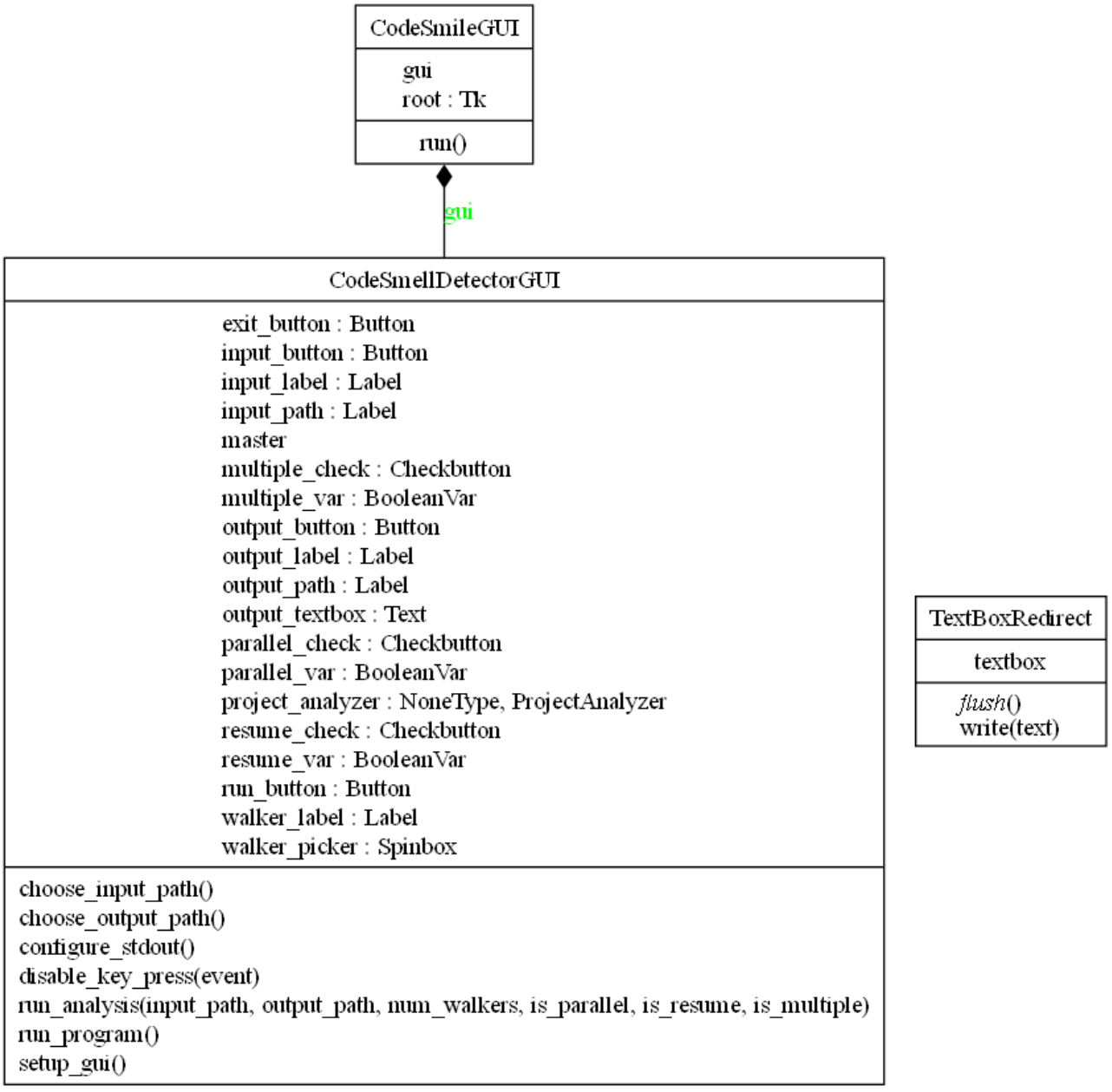


*Figura 2.4: Class Diagram del package gui*

**Package** `report`

Responsabile della generazione degli output finali.

- Produce report dettagliati (CSV, Excel) e visualizzazioni grafiche, aggregando i risultati per progetto o globalmente.

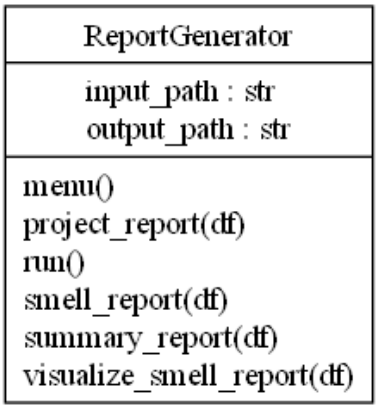


*Figura 2.5: Class Diagram del package report*

**Package** `utils`

Fornisce utilità trasversali, come la gestione del file system, la pulizia delle directory e la gestione sicura dei log in ambienti multi-thread.

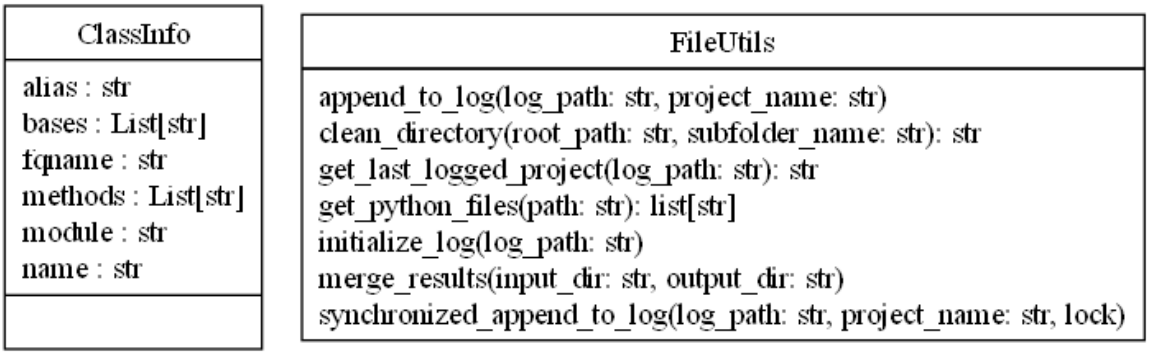


*Figura 2.6: Class Diagram del package utils*

### 2.1.2 Web App e Microservizi

L'architettura web di CodeSmile è moderna e distribuita, composta da un frontend interattivo e un backend a microservizi.

#### Frontend (Web App)

Sviluppata in **React** / **Next.js** e **Node.js**, l'applicazione offre un'interfaccia per caricare progetti, visualizzare risultati ed esportare report. I package principali del frontend includono:

- `app`: Gestisce il layout globale, le pagine e le viste interattive.
- `components`: Una raccolta di elementi UI riutilizzabili (grafici, input, widget).
- `context`: Gestisce lo stato globale dell'applicazione (es. lista progetti).
- `utils` & `types`: Gestiscono le chiamate HTTP (Axios) e le definizioni dei tipi condivisi.

#### Backend (Microservizi e Gateway)

Il backend è strutturato su microservizi Python indipendenti che comunicano tramite un **API Gateway** basato su **FastAPI**.

**1. API Gateway** (`package gateway`) Funge da punto di ingresso unico. Gestisce il routing delle richieste, il CORS e l'instradamento verso i servizi specifici, esponendo endpoint unificati come `/api/detect_smell_ai` o `/api/generate_report`.

**2. I Servizi** (`package services`) I servizi sono suddivisi in tre moduli principali:

- **AI Analysis Service:**
  - Si occupa del rilevamento semantico dei code smells utilizzando modelli di linguaggio (LLM).
  - Utilizza un client per analizzare snippet di codice validati e restituire i risultati in formato JSON.
- **Static Analysis Service:**
  - Esegue l'analisi basata su AST e regole.
  - Crea file temporanei dal codice ricevuto e utilizza l'infrastruttura core (`Inspector`) per rilevare gli smells, garantendo coerenza con la versione CLI/Desktop.
- **Report Service:**
  - Aggrega i risultati delle analisi provenienti da più progetti o file.
  - Utilizza Pandas per elaborare i dati e generare statistiche e strutture adatte alla visualizzazione nel frontend.

# 3 Funzionamento del tool

Il tool CodeSmile è stato progettato per adattarsi a scenari differenti, offrendo diverse modalità di interazione a seconda delle esigenze dell'utente, spaziando dalla riga di comando all'interfaccia web.

## 3.1 Modalità di Utilizzo

Il sistema supporta tre modalità principali di utilizzo, ognuna pensata per un target specifico di utenza.

### 3.1.1 Utilizzo tramite Linea di Comando (CLI)

Questa modalità è orientata a sviluppatori e ricercatori che necessitano di eseguire analisi mirate in maniera rapida, efficiente e ripetibile. Attraverso la CLI è possibile:

- Analizzare singoli file o interi progetti Python.
- Specificare manualmente le directory di input e di output.
- Configurare l'esecuzione in modalità sequenziale o parallela per velocizzare l'elaborazione su più file.
- Gestire la ripresa di analisi interrotte e l'elaborazione multi-progetto.

### 3.1.2 Utilizzo tramite Interfaccia Grafica (GUI)

La GUI fornisce un'interfaccia desktop basata su Tkinter che consente un'interazione più guidata rispetto alla CLI, ideale per utenti che preferiscono non scrivere comandi manualmente. Le funzionalità principali includono:

- Selezione visuale dei parametri e configurazione delle cartelle di input/output.
- Avvio dell'analisi con supporto al parallelismo e al resume.
- Visualizzazione dei log in tempo reale direttamente nella finestra dell'applicazione.

### 3.1.3 Utilizzo tramite Web App

La Web App offre un'interfaccia accessibile via browser, eliminando la necessità di installazione locale e facilitando l'interazione visuale e collaborativa. Tramite questa modalità gli utenti possono:

- Caricare file (snippet) o interi progetti direttamente dal browser.
- Scegliere tra due motori di analisi: **AI-Based** (basata su modelli LLM) o **Static Analysis** (basata su AST).
- Visualizzare i risultati ed esportare report per analisi successive.

### 3.1.4 Flusso di Funzionamento Dettagliato

Di seguito viene approfondita la logica tecnica di esecuzione per le tre modalità, evidenziando come i diversi componenti architetturali interagiscono tra loro.

#### Workflow CLI (Command Line Interface)

L'esecuzione via riga di comando è orchestrata dal modulo `cli_runner`, che agisce come entry point del sistema. Il processo segue una pipeline strutturata:

- **Inizializzazione e Configurazione:** Il sistema interpreta gli argomenti passati (directory, flag di parallelismo) e istanzia il `ProjectAnalyzer`, preparando l'ambiente di lavoro (es. pulizia delle directory di output).
- **Core Analysis Loop:** Il controllo viene demandato al motore di analisi che, operando in modalità sequenziale o parallela, itera sui file sorgente. Per ogni unità di codice, viene invocato l'`Inspector` per generare l'Abstract Syntax Tree (AST), estrarre le entità semantiche e applicare la batteria di regole di rilevamento (Detection Rules).
- **Aggregazione dei Risultati:** Al termine dell'elaborazione, i risultati parziali salvati su disco vengono consolidati in report strutturati (CSV/Excel) pronti per la consultazione.

#### Workflow GUI (Graphical User Interface)

L'interfaccia grafica funge da wrapper visuale intorno al motore di analisi, migliorando l'accessibilità senza sacrificare le prestazioni:

- **Astrazione e Redirezione:** Il modulo `gui_runner` inizializza l'ambiente Tkinter e configura un meccanismo di intercettazione dello standard output (`stdout`). Questo permette di proiettare i log tecnici generati dal backend direttamente nei widget testuali dell'interfaccia in tempo reale.
- **Esecuzione Asincrona:** Alla conferma dell'utente, l'analisi viene lanciata su un thread separato per non bloccare l'interfaccia. Il thread invoca le medesime API del `ProjectAnalyzer` utilizzate dalla CLI, garantendo così una totale coerenza nei risultati tra le due modalità.

**Workflow Web App e Microservizi**

Il funzionamento in ambiente web si basa su un'architettura distribuita a microservizi, dove il flusso dati attraversa diversi layer logici:

- **Comunicazione Client-Gateway:** L'interazione inizia dal frontend (Next.js) che invia il codice sorgente in modo asincrono all'API Gateway (FastAPI). Quest'ultimo funge da router intelligente, gestendo la sicurezza (CORS) e smistando la richiesta verso il servizio competente.
- **Elaborazione dei Servizi:**
  - Il **Static Analysis Service** riceve il payload, crea un contesto di esecuzione temporaneo e riutilizza la logica core dell'`Inspector` (condivisa con la versione desktop) per l'analisi AST.
  - L'**AI Analysis Service** interroga il modello di linguaggio (LLM) per un'analisi semantica avanzata.
- **Serializzazione e Risposta:** I code smells identificati vengono serializzati in formato JSON e restituiti al frontend attraverso il Gateway, permettendo la visualizzazione dinamica delle problematiche direttamente nel browser.