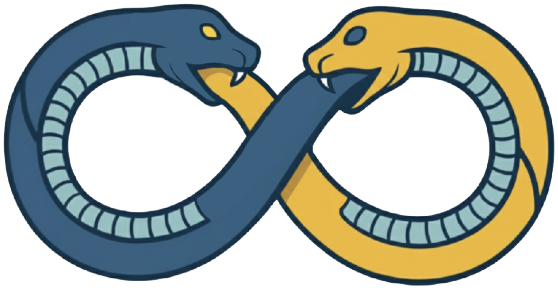


REVERTY

Ingegneria dei Linguaggi di Programmazione

https://github.com/Gianpyy/Reverty

Studenti

Claudio Buono - NF22500051

Gianpio Silvestri - NF22500083

Docenti

Gennaro Costagliola

ANNO ACCADEMICO 2025/2026

Indice

1 Introduzione

Il progetto si propone di realizzare un'**architettura multi-agente** basata su Large Language Models (LLM) per la generazione automatica di codice. Per validare rigorosamente le capacità di ragionamento e generalizzazione degli agenti, abbiamo scelto di non utilizzare un linguaggio standard, bensì di progettare un linguaggio ad-hoc denominato **Reverty** (Reverse-Python).

1.1 Cos'è Reverty

Reverty è un linguaggio progettato appositamente per fungere da nucleo dell'architettura proposta. Si tratta di un linguaggio costruito su misura, concepito non come un linguaggio di programmazione completamente nuovo, ma come una versione speculare di Python. In altre parole, **Reverty** riproduce la struttura sintattica e semantica di Python invertendone la direzione di lettura, con l'obiettivo di esplorare un modello alternativo di rappresentazione del codice mantenendo piena compatibilità concettuale con il linguaggio originale.

1.2 Come funziona Reverty

L'idea alla base di Reverty è semplice ma efficace: separare il *significato* del codice dalla sua *forma*.

- **La logica (Semantica):** Il comportamento del programma rimane invariato rispetto a Python. Le operazioni matematiche, la logica dei dati e il flusso di esecuzione funzionano esattamente come ci si aspetta in un normale programma Python.
- **La scrittura (Sintassi):** Ciò che cambia radicalmente è il modo in cui il codice si presenta visivamente. Abbiamo applicato una regola di inversione sistematica a tutte le parole chiave del linguaggio e all'ordine delle parole chiave nei costrutti del linguaggio.

In termini pratici, questo significa che un programmatore (o in questo caso, un modello di Intelligenza Artificiale) deve scrivere le istruzioni al contrario.

Ad esempio, il classico costrutto condizionale `if` viene trasformato in `fi`, il ciclo `for` diventa `rof`, e la definizione di una funzione, normalmente introdotta da `def`, in Reverty si scrive `fed`.

1.3 Motivazioni per una sintassi invertita

L'adozione di una sintassi invertita non rappresenta una scelta puramente estetica, ma risponde a precise esigenze di analisi e valutazione. In particolare, essa consente di testare due aspetti fondamentali del comportamento dei Large Language Models.

1. Ragionamento effettivo contro riproduzione a memoria

I modelli linguistici di grandi dimensioni sono stati addestrati su enormi quantità di codice Python e, di conseguenza, tendono spesso a generare output corretti facendo affidamento su schemi appresi, in modo simile a un correttore automatico, piuttosto che attraverso un reale processo di ragionamento. L'utilizzo di parole inventate, in questo caso ottenute tramite inversione, impedisce al modello di affidarsi a tali automatismi. Poiché questi termini non fanno parte del suo addestramento, il modello non può "riconoscerli" o completarli per analogia, ma è costretto ad applicare consapevolmente le regole sintattiche definite, ragionando su ogni singola istruzione.

2. Controllo oggettivo dell'output

L'integrazione di strumenti formali come il parser *Lark* introduce un livello di controllo rigoroso e verificabile sull'output prodotto dai modelli linguistici. Mentre la valutazione di un testo in linguaggio naturale è intrinsecamente soggettiva e difficilmente misurabile, l'approccio adottato con Reverty consente di spostare l'analisi su un piano puramente logico e deterministico. In questo contesto, la correttezza dell'output non dipende da interpretazioni qualitative, ma dal rispetto formale delle regole sintattiche definite, rendendo la valutazione riproducibile e oggettiva.

2 Specifiche del Linguaggio

2.1 Definizione Lessicale

2.1.1 Keywords

Le keyword di Reverty derivano dall'inversione delle parole chiave di Python, mantenendo inalterata la loro funzione logica.

Token	Definizione
DEF	`"fed"`
RETURN	`"nruter"`
IF	`"fi"`
ELIF	`"file"`
ELSE	`"esle"`
WHILE	`"elihw"`
FOR	`"rof"`
IN	`"ni"`
TRUE	`"eurT"`
FALSE	`"eslaF"`
NONE	`"enoN"`
AND	`"dna"`
OR	`"ro"`
NOT	`"ton"`

2.1.2 Tipi Primitivi Riservati

Oltre alle keyword di controllo flusso, Reverty riserva un set specifico di token per i tipi di dato primitivi. Questi token sono soggetti alla regola di inversione lessicale per forzare il modello a una mappatura semantica esplicita.

Token	Tipo Reverty	Tipo Python
TYPE_INT	`tni`	`int`
TYPE_FLOAT	`taolf`	`float`

Token	Tipo Reverty	Tipo Python
TYPE_STR	`rts`	`str`
TYPE_BOOL	`loob`	`bool`
TYPE_NONE	`enoN`	`None`

Nota tecnica: In Python, `None` è propriamente un valore (singleton della classe `NoneType`). Tuttavia, nel contesto dei Type Hints, il token `None` è ammesso come alias per indicare il tipo di ritorno nullo. Reverty eredita questa convenzione permettendo l'uso di `enoN` sia come valore letterale nelle espressioni, sia come indicatore di tipo nelle firme di funzione.

2.1.3 Operatori e Delimitatori

Qui vengono definiti i simboli statici utilizzati per le operazioni e la punteggiatura.

Token	Simbolo
PLUS	`"+"`
MINUS	`"-"`
TIMES	`"*"`
DIV	`"/"`
MOD	`"%"`
ASSIGN	`"="`
EQ	`"=="`
NE	`"!="`
LT	`"<"`
GT	`">"`
LE	`"<="`
GE	`">="`
LPAR	`"("`
RPAR	`")"`
COLON	`":"`
COMMA	`","`
ARROW	`"->"`

2.1.4 Pattern Lessicali e Literals

A differenza delle keyword e degli operatori fissi, i seguenti token sono definiti tramite **Espressioni Regolari** per gestire identificatori e valori costanti.

Token	Espressione Regolare
ID	`[a-zA-Z_][a-zA-Z0-9_]*`
NUMBER	`[0-9]+(.[0-9]+)?`
STRING	`".*?"`
COMMENT	`#[^\n]*`

2.1.5 Gestione dell'Indentazione

Essendo derivato da Python, Reverty gestisce allo stesso modo la definizione dei blocchi. Il lexer tiene traccia del livello di indentazione e genera token virtuali:

Token Speciale	Descrizione
NEWLINE	Fine di una riga logica (`\n`).
INDENT	Generato quando il livello di indentazione aumenta rispetto alla riga precedente (inizio blocco).
DEDENT	Generato quando il livello di indentazione diminuisce (fine blocco).

2.2 Definizione sintattica

La definizione sintattica del linguaggio è processata mediante la libreria `Lark`, che consente di esprimere una grammatica context-free con supporto nativo all'indentazione significativa. La grammatica è progettata per descrivere in modo formale e non ambiguo la struttura dei programmi, includendo costrutti di controllo, definizione di funzioni, sistema di tipi e gestione delle espressioni.
In questa sezione la grammatica viene analizzata suddividendo i costrutti per area funzionale, con l'obiettivo di fornire una specifica sintattica chiara.

2.2.1 Struttura del programma e indentazione

Un programma è definito come una sequenza di istruzioni (`stmt`), eventualmente separate da righe vuote o commenti. Il simbolo iniziale della grammatica è `start`, che consente la presenza di newline isolati senza impatto sulla struttura del programma.

```
start: (_NEWLINE | stmt)*
```

La regola `suite` riveste un ruolo centrale, in quanto consente di rappresentare sia istruzioni su singola linea sia blocchi di istruzioni indentate, utilizzati nei costrutti composti come funzioni, cicli e condizionali.

```
suite: simple_stmt | _NEWLINE _INDENT stmt+ _DEDENT

?stmt: simple_stmt | compound_stmt
?simple_stmt: small_stmt _NEWLINE
?small_stmt: return_stmt | assign_stmt | expr_stmt

?compound_stmt: func_def | conditional_stmt | while_stmt |
   for_stmt
```

2.2.2 Funzioni, parametri e assegnamenti

La definizione delle funzioni segue una sintassi rigorosa, caratterizzata da una firma completamente esplicita. Ogni funzione specifica obbligatoriamente il tipo di ritorno, il nome identificativo, un elenco opzionale di parametri tipizzati e un corpo espresso come `suite`.
I parametri di funzione devono essere accompagnati da un *type hint*, mentre nelle assegnazioni il tipo è opzionale. L'istruzione `return`, invece, può comparire sia con che senza un'espressione associata.

```
func_def: ":" type_hint "->" "(" [params] ")" NAME "fed"
   suite

params: param ("," param)*
param: type_hint ":" NAME

return_stmt: "nruter" [expr]
assign_stmt: NAME (":" type_hint)? "=" expr
```

2.2.3 Sistema di tipi

Il sistema di tipi è definito sintatticamente tramite il non terminale `type_hint`. L'insieme dei tipi primitivi supportati è chiuso e comprende tipi interi, reali, booleani, stringhe e il tipo `None`.

```
type_hint: "tni"   -> type_int
| "rts"   -> type_str
| "loob"  -> type_bool
| "enoN"  -> type_none
| "taolf" -> type_float
```

2.2.4 Controllo del flusso

La grammatica supporta costrutti di controllo strutturati per l'esecuzione condizionale e iterativa del codice. In particolare, sono previsti costrutti condizionali completi e cicli a controllo esplicito.

Le istruzioni condizionali sono costituite da un costrutto `if`, seguito da zero o più `elif` e da un `else` opzionale. Ogni ramo è associato a una condizione e a un blocco di codice definito come `suite`.

```
conditional_stmt: if_stmt (elif_stmt)* [else_stmt]
if_stmt: ":" expr "fi" suite
elif_stmt: ":" expr "file" suite
else_stmt: ":" "esle" suite
```

In maniera simile, i cicli utilizzano una condizione ed un blocco di codice, sempre definito come `suite`. Il ciclo `for`, a differenza del `while`, permette di iterare su stringhe oppure su un'espressione `range`.

```
while_stmt: ":" expr "elihw" suite
for_stmt: ":" loop_expr "ni" NAME "rof" suite

loop_expr: STRING | range_expr
range_expr: "range" "(" expr ")"
```

2.2.5 Espressioni e gerarchia delle precedenze

Le espressioni costituiscono il nucleo computazionale del linguaggio e sono definite come una struttura ricorsiva unica. La grammatica impone esplicitamente la gerarchia delle precedenze attraverso la scomposizione in livelli sintattici distinti. In ordine di precedenza crescente, le espressioni sono organizzate come segue:

1. **Atom**: unità indivisibili come letterali, variabili e chiamate di funzione.
2. **Product**: operatori moltiplicativi.
3. **Sum**: operatori additivi.
4. **Comparison**: operatori di confronto.
5. **Logic**: operatori logici, con precedenza minima.

```
expr_stmt: expr
?expr: logic_or

?logic_or: logic_and ("ro" logic_and)*
?logic_and: logic_not ("dna" logic_not)*
?logic_not: "ton" logic_not -> not_expr
| comparison

?comparison: sum (comp_op sum)*

?sum: product (add_op product)*

?product: atom (mul_op atom)*

?atom: NUMBER         -> number
| STRING         -> string
| "eurT"       -> true
| "eslaF"      -> false
| "enoN"       -> none
| func_call
| NAME           -> var
| "(" expr ")"
| add_op atom    -> unary_op

func_call: NAME "(" [arguments] ")"
arguments: expr ("," expr)*
```

2.2.6 Elementi lessicali

La grammatica definisce esplicitamente gli elementi lessicali fondamentali del linguaggio, tra cui identificatori, numeri e stringhe.

```
NAME: /[a-zA-Z_]\w*/
NUMBER: /\d+(\.\d+)?/
STRING: /".*?"/
```

La gestione dei blocchi di codice non avviene tramite delimitatori espliciti, ma attraverso i token virtuali _INDENT e _DEDENT. Tali token vengono generati automaticamente da *Lark* durante la fase di tokenizzazione, analizzando i livelli di indentazione del codice sorgente.

```
%declare _INDENT _DEDENT
```

I commenti e gli spazi bianchi vengono ignorati dal parser, mentre i caratteri di fine linea rivestono un ruolo sintattico essenziale nella separazione delle istruzioni.

```
COMMENT: /#[^\n]*/
%ignore COMMENT
%ignore /[ \t\f]+/
%ignore /\\[\t \f]*\r?\n/

_NEWLINE: ( /\r?\n[\t ]*/ | COMMENT )+
```

2.3 Definizione Semantica

Reverty adotta un modello semantico semplificato, progettato per garantire determinismo e traducibilità diretta verso il linguaggio target (Python). Le specifiche semantiche si fondano su due pilastri: Scope Unico e Tipizzazione Pass-through.

2.3.1 Gestione dello scope

Reverty implementa un Singolo Scope Globale:

- Tutti gli identificatori (variabili e funzioni) risiedono in un unico spazio dei nomi.
- Non è supportato l'oscuramento delle variabili: non è possibile dichiarare una variabile locale con lo stesso nome di una globale o di un'altra funzione. Questa scelta progettuale è intesa a ridurre le ambiguità nella risoluzione dei nomi, forzando la produzione di codice con riferimenti univoci.

2.3.2 Tipizzazione pass-through

Il sistema di tipi di Reverty agisce secondo un principio di validazione differita.

- **Obbligo Sintattico:** A livello formale, Reverty impone la presenza di Type Hints espliciti in tutte le firme di funzione (es. `tni: x`). Questo vincolo serve a forzare l'Agente LLM a esplicitare le sue intenzioni.
- **Delega Semantica:** A livello sostanziale, queste annotazioni vengono trattate come metadati "pass-through". Non viene eseguita inferenza di tipo; le annotazioni vengono tradotte letteralmente nel codice Python risultante, delegando all'interprete ospite il compito di far emergere eventuali incoerenze a runtime.

2.3.3 Validità per traducibilità

Il concetto di ”errore semantico” in Reverty viene ridefinito in funzione della pipeline di generazione:

1. **Errori Bloccanti (Pre-Traduzione):** Violazioni che impediscono la generazione del codice (es. riferimenti a variabili non dichiarate nella tabella dei simboli).

2. **Errori Delegati (Post-Traduzione):** Incoerenze logiche (es. somma tra tipi incompatibili) che vengono lasciate passare affinché generino eccezioni tracciabili nella Sandbox. Questo meccanismo è fondamentale per fornire agli Agenti uno stack trace reale su cui basare il ragionamento correttivo.

2.4 Esempio di Codice

Di seguito viene riportato un esempio completo di programma Reverty che calcola il fattoriale.

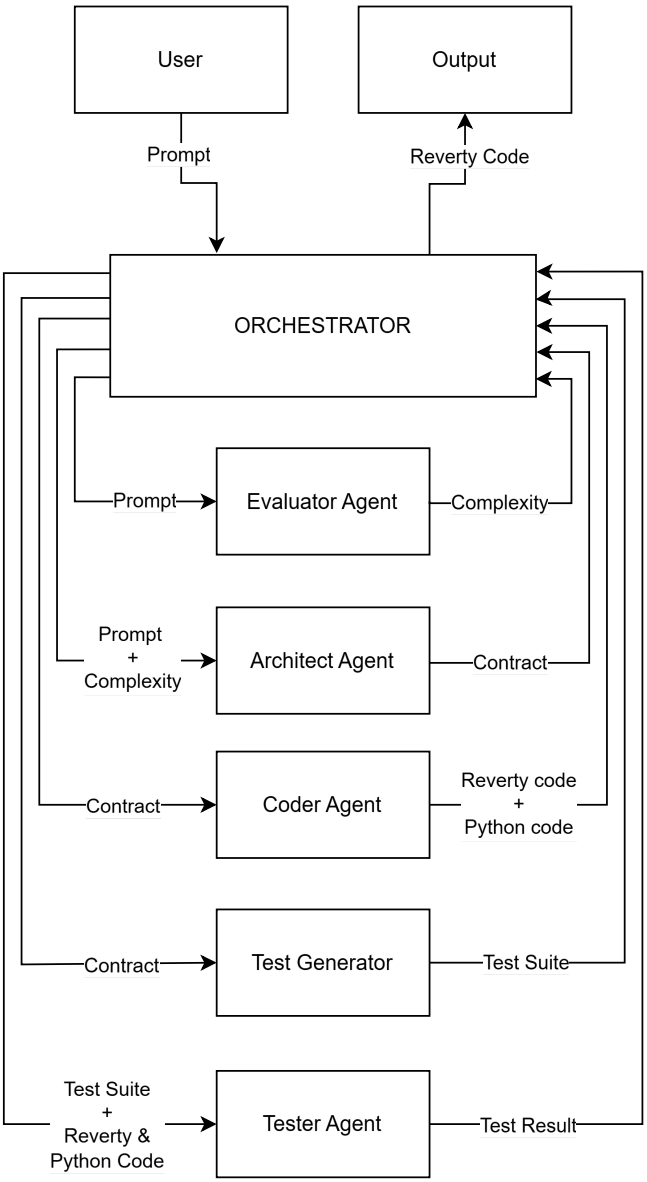
```
# Funzione per il calcolo del fattoriale
tni -> (tni: n) factorial fed
    res : tni = 1
        : n > 1 elihw
            res = res * n
            n = n - 1
        nruter res

# Entry point del programma
tni -> () main fed:
    result: tni = factorial(5)
    nruter result
```

3 Soluzione Proposta

3.1 Architettura del Sistema

L'architettura proposta implementa una pipeline completa di sviluppo software, simulata attraverso agenti specializzati. Il sistema è orchestrato centralmente e suddiviso in fasi funzionali che ricalcano il ciclo di vita del software.



3.1.1 Workflow principale

Il flusso di lavoro, governato dal componente centrale **Orchestrator**, si articola nei seguenti agenti specializzati:

1. **Fase di Analisi e Design**

    - **Evaluator Agent:** Analizza preliminarmente la richiesta in linguaggio naturale. Il suo compito è quantificare la complessità della soluzione (punteggio da 1 a 10). Questo punteggio viene utilizzato per calibrare dinamicamente le richieste per gli agenti successivi.
    - **Architect Agent:** Riceve in input la richiesta utente e la valutazione di complessità. Il suo obiettivo è formalizzare l'intento producendo un **Technical Contract**: una specifica tecnica rigorosa che definisce requisiti funzionali, firme delle funzioni e strutture dati, eliminando l'ambiguità tipica del linguaggio naturale.

2. **Fase di Implementazione**

    - **Coder Agent:** Opera come unità di sviluppo pura. A partire dalle specifiche del *Technical Contract* genera il codice sorgente nel linguaggio target **Reverty**. Successivamente, il codice prodotto viene sottoposto a validazione tramite strumenti di analisi statica dedicati e transpilato in **Python**. L'agente possiede capacità di auto-correzione: in caso di errori rilevati durante la generazione o il testing, itererà sul codice per risolverli.

3. **Fase di Testing e Validazione**

    - **Test Generator Agent:** Automatizza la creazione di una suite di test unitari derivando i casi d'uso direttamente dai requisiti funzionali esposti nel *Technical Contract* e dalla logica implementata nel codice transpilato.
    - **Tester Agent:** Esegue il codice Python generato all'interno di un ambiente isolato contro la Test Suite. Oltre a registrare gli esiti (Pass/Fail)

e gli stack trace, agisce come analista diagnostico: in caso di fallimento, esamina la coerenza tra il *Technical Contract*, codice e test per discernere se l'errore risieda nell'implementazione (bug del codice) e/o nella verifica (bug del test), indirizzando il feedback all'agente competente.

3.2 Interfaccia Utente

L'interfaccia grafica è stata progettata con l'obiettivo primario di rendere trasparente e interattivo il processo di generazione multi-agente. Il front-end funge da ponte tra l'utente e il backend di orchestrazione, offrendo strumenti per la configurazione dei modelli, il monitoraggio in tempo reale e l'analisi dei risultati prodotti.
L'organizzazione visiva è suddivisa in tre macro-aree funzionali:

3.2.1 Pannello di Configurazione

Posto lateralmente, questo modulo permette all'utente di definire i vincoli operativi e i parametri del sistema prima dell'avvio del processo:

- **Selezione del modello:** È possibile scegliere il modello linguistico da utilizzare tra quello locale e quello in cloud.
- **Controllo della Temperatura:** Uno slider dedicato permette di calibrare il grado di determinismo del modello, influenzando la creatività delle soluzioni proposte dagli agenti.
- **Gestione dei Limiti di Iterazione:** Per prevenire loop infiniti o costi eccessivi, l'utente può impostare soglie massime per i cicli di orchestrazione, valutazione e validazione, garantendo che il processo termini anche in caso di errori persistenti.
- **Sicurezza:** Un campo dedicato permette l'inserimento sicuro delle chiavi API, qualora il backend selezionato lo richieda.

3.2.2 Interazione e Monitoraggio

La sezione centrale è dedicata all'interazione. Qui l'utente inserisce la richiesta in linguaggio naturale o seleziona scenari predefiniti da una libreria di esempi.
Un elemento cruciale di questa sezione è il **sistema di logging in tempo reale:** l'interfaccia espone dinamicamente lo scambio di messaggi tra gli agenti man mano che avvengono. Questo flusso visivo offre un feedback immediato sul "ragionamento" del sistema, permettendo all'utente di comprendere come la richiesta iniziale venga interpretata, raffinata e tradotta.

3.2.3 Visualizzazione dei risultati

La colonna di destra è dedicata all'analisi dell'output. Al termine della generazione, il sistema presenta i risultati organizzati in schede per facilitare il confronto e il debugging:

- **Codice Reverty:** Visualizza il codice generato nel linguaggio target.
- **Codice Python:** Visualizza il codice ottenuto a partire dal linguaggio target.
- **Esploratore AST:** Una visualizzazione ad albero interattiva dell'Abstract Syntax Tree.

3.3 TOON

https://github.com/toon-format/toon

Il **Token-Oriented Object Notation** è un formato di serializzazione dei dati progettato specificamente per l'era dell'IA Generativa. Mentre formati come JSON o XML sono nati per la comunicazione tra macchine (server-to-server) o per la leggibilità umana classica, TOON nasce con un terzo destinatario in mente: il Tokenizer degli LLM.
In sintesi, TOON è una rappresentazione 1:1 del modello dati JSON, ma liberata da tutto il "rumore sintattico" che aumenta il conteggio dei token e distrae i meccanismi di attenzione dei modelli.

JSON

```json
{
  "context": {
    "task": "Our favorite hikes",
    "location": "Boulder",
    "season": "spring_2025"
  },
  "friends": ["ana", "luis", "sam"],
  "hikes": [
    {
      "id": 1,
      "name": "Blue Lake Trail",
      "dist": 7.5,
      "elev": 320,
      "with": "ana",
      "sunny": true
    },
    {
      "id": 2,
      "name": "Ridge Overlook",
      "dist": 9.2,
      "elev": 540,
      "with": "luis",
      "sunny": false
    }
  ]
}
```

TOON

```
context:
  task: Our favorite hikes
  location: Boulder
  season: spring_2025
friends[3]: ana,luis,sam
hikes[2]{id,name,dist,elev,with,sunny}:
  1,Blue Lake Trail,7.5,320,ana,true
  2,Ridge Overlook,9.2,540,luis,false
```

Figura 3.1: Confronto diretto tra JSON e TOON

3.3.1 Il Problema del JSON negli LLM

- **Spreco di Token**: Caratteri come {, }, ", :, sono spesso tokenizzati separatamente o causano la frammentazione delle parole adiacenti.
- **Sintassi Rigida**: La sintassi rigida costringe il modello a "sprecare" attenzione sulla struttura (chiudere le parentesi, mettere le virgole) invece che sul contenuto semantico.
- **Consumo di spazio**: Un JSON verboso consuma prezioso spazio nella finestra di contesto, limitando la quantità di dati che può inviare.

3.3.2 Vantaggi di TOON

L'integrazione di questa notazione porta quattro vantaggi immediati:

1. **Riduzione dei Costi (API)**: Meno token di input e meno token di output significano una riduzione diretta dei costi delle chiamate API.
2. **Minore Latenza**: Generare meno token richiede meno tempo. Le risposte del modello saranno più veloci.
3. **Maggiore Accuratezza**: Rimuovendo la complessità sintattica del JSON (es. il modello che dimentica una virgola o una parentesi graffa finale), riduce gli errori di parsing. Il modello si concentra sui dati, non sulla formattazione.
4. **Parsing "Lossless"**: Essendo una mappatura diretta del modello dati JSON, può convertire JSON a TOON per l'input dell'LLM e convertire l'output TOON a JSON per il codice, senza perdere informazioni.

Formato	**Efficienza** (acc%/1K tok)	**Accuratezza** (%)	**Token** (Count)
TOON	**26.9**	**73.9%**	**2,744**
JSON compact	22.9	70.7%	3,081
JSON	15.3	69.7%	4,545

Confronto delle Performance: TOON vs JSON.

4 Integrazione degli LLM

4.1 Modelli utilizzati

Il sistema supporta due principali modelli:

- **Ollama:** Rappresenta l'interfaccia verso modelli LLM eseguiti localmente tramite il framework `Ollama`.
- **GitHub Models:** Interfaccia remota verso i modelli linguistici ospitati da GitHub Models (es. modelli basati su GPT o Claude).

4.2 Prompt Engineering

Ogni agente è dotato di un proprio *system prompt*, progettato per specializzare il comportamento del modello in base al compito da svolgere.

4.2.1 System Prompts dedicati

- `EVALUATOR_SYSTEM_PROMPT`: istruisce il modello a stimare la complessità di un compito in modo numerico e strutturato.
- `ARCHITECT_SYSTEM_PROMPT_SIMPLE` e `ARCHITECT_SYSTEM_PROMPT_COMPLEX`: definiscono due modalità operative per generare contratti tecnici, una per compiti semplici e una per problemi complessi.
- `CODER_SYSTEM_PROMPT`: guida la generazione del codice Reverty, includendo la grammatica formale del linguaggio.
- `TESTER_GENERATOR_SYSTEM_PROMPT`: orienta il modello nella scrittura di test unitari basati su `pytest`.
- `TESTER_SYSTEM_PROMPT`: utilizza il modello per analizzare i fallimenti dei test e individuare la causa degli errori.

4.2.2 Tecniche di ottimizzazione dei prompt

Durante lo sviluppo del progetto, i *system prompt* sono stati progressivamente raffinati al fine di migliorare l'aderenza dei modelli — sia locali che Cloud — agli obiettivi di generazione prefissati.
In particolare, sono state adottate le seguenti tecniche di prompt engineering:

- definizione di **ruoli funzionali** espliciti per il modello (ad esempio *"You are an expert Software Architect"* o *"You are an expert QA Engineer"*), al fine di orientarne il comportamento;
- utilizzo di istruzioni chiare e non ambigue per guidare il processo di generazione;
- impiego selettivo delle **maiuscole** per enfatizzare vincoli o requisiti critici del prompt;
- adozione di **template strutturati** per il formato dell'output, così da ridurre la variabilità sintattica;
- fornitura di esempi di input/output tramite tecniche di *few-shot prompting*;
- concatenazione del *system prompt* con la grammatica formale del linguaggio Reverty, allo scopo di vincolare la generazione al rispetto delle regole sintattiche;
- introduzione di **vincoli espliciti e penalizzazioni semantiche** nel prompt per scoraggiare deviazioni dal formato e dalla sintassi richiesti.

5 Implementazione

5.1 Tecnologie Utilizzate

Il sistema è stato sviluppato interamente in **Python**, sfruttando librerie open-source per garantire modularità e manutenibilità.
Le componenti tecnologiche principali sono le seguenti:

- **Streamlit:** Framework principale per la costruzione della web application.
- **Lark:** Parser utilizzato per definire la grammatica formale del linguaggio *Reverty* e generare l'Abstract Syntax Tree (AST), fondamentale per la fase di transpilazione.
- **TOON Format for Python:** Una libreria oper-source utilizzata per gestire la serializzazione degli oggetti di scambio tra agenti nel formato TOON.
- **Flake8:** Strumento di analisi statica (linter) che combina *PyFlakes* e *pycodestyle* per garantire la conformità agli standard PEP 8.
- **Mypy:** Static Type Checker utilizzato per introdurre un controllo rigoroso dei tipi nel codice Python.
- **Pytest:** Framework di testing utilizzato per l'esecuzione automatizzata delle suite di test e per la validazione funzionale del codice generato.

5.2 `Orchestrator`

La classe `Orchestrator` costituisce il nucleo di coordinamento del sistema multi-agente, validazione e correzione di codice a partire dal prompt fornito dall'utente. Il suo compito principale è gestire l'intero *workflow* di generazione, compilazione e verifica, orchestrando in maniera sequenziale e iterativa l'interazione tra i diversi agenti specializzati che compongono l'architettura.

5.2.1 Panoramica

L'Orchestrator riceve in input un *prompt* testuale, ossia la descrizione del problema o della funzionalità che si desidera implementare. A partire da questo input, il modulo coordina le seguenti fasi principali:

1. **Valutazione della complessità**: tramite l'agente `EvaluatorAgent`, il sistema stima la complessità del compito richiesto. Questa informazione guida le scelte successive di generazione del codice e delle strutture di test.

2. **Progettazione del *Technical Contract***: l'agente `ArchitectAgent` elabora un *Technical Contract* che formalizza i requisiti funzionali e non funzionali del programma, traducendo il linguaggio naturale in una rappresentazione strutturata e interpretabile dagli altri agenti e basandosi sulla complessità valutata dall'agente valutatore.

3. **Generazione o correzione del codice**: l'agente `CoderAgent` produce il codice sorgente in due forme parallele:

   - il codice `Reverty`
   - il corrispettivo codice `Python`, eseguibile e testabile.

   In caso di errori, la stessa fase può operare in modalità di correzione, tramite la specifica `RequestType` (`FIX_CODE` o `FIX_BOTH`).

4. **Generazione e validazione dei test**: l'agente `TestGeneratorAgent` costruisce i casi di test coerenti con il *Technical Contract*, mentre il `TesterAgent` li esegue effettivamente contro il codice Python generato.

5. **Analisi dei risultati e ciclo di revisione**: se vengono individuati errori nel codice o nei test, l'Orchestrator aggiorna il tipo di richiesta (`RequestType`) e ripete il ciclo di generazione/correzione fino al raggiungimento di un esito positivo o del numero massimo di iterazioni consentito.

5.2.2 Architettura interna

L'Orchestrator istanzia e collega tutti gli agenti necessari, impostando inoltre il tipo di client LLM (`Mock`, `Ollama`, `GitHub Models`) responsabile dell'elaborazione linguistica e generativa. Ogni agente opera come un modulo autonomo ma coordinato, e l'Orchestrator agisce come un *controller* di alto livello, garantendo coerenza, logging e gestione dello stato condiviso.

5.2.3 Orchestration Loop

Il flusso operativo dell'Orchestrator può essere rappresentato come un ciclo iterativo:

$\rightarrow$ `Generate Code` $\rightarrow$ `Generate Tests` $\rightarrow$ `Execute Tests` $\rightarrow$ `(Fix if needed)`

Ogni iterazione termina solo quando il codice passa con successo tutti i test oppure viene raggiunto il limite massimo di iterazioni (`MAX_ORCHESTRATOR_ITERATIONS`). In questo modo, il sistema realizza un vero e proprio processo di *auto-compilazione e auto-validazione* controllata.

5.3 Client LLM

5.3.1 Interfaccia astratta `LLMClient`

La comunicazione tra gli agenti e i modelli linguistici è mediata da un'interfaccia astratta denominata `LLMClient`. Tale classe fornisce l'implementazione del metodo astratto `generate()`, che costituisce il punto di accesso unificato alle chiamate dei modelli. Il metodo ha i seguenti parametri:

- **prompt**: La richiesta principale fornita all'LLM che descrive il compito da svolgere, come la richiesta dell'utente o le specifiche del *Technical Contract*.
- **system_prompt**: Un prompt aggiuntivo che fornisce istruzioni al modello su come comportarsi o sul contesto in cui operare. Viene utilizzato per guidare lo stile di generazione, la formattazione del codice e il livello di dettaglio delle risposte.
- **model**: Specifica quale modello linguistico deve essere utilizzato per la generazione.

5.3.2 Tipologie di Client LLM

A partire dall'interfaccia `LLMClient`, il sistema implementa tre versioni concrete, ognuna con scopi e caratteristiche differenti:

- `MockLLMClient`: Simula il comportamento di un modello linguistico restituendo risposte predefinite. Utilizzato in fase di sviluppo e test per verificare il corretto funzionamento dell'orchestrazione senza dipendere da servizi esterni.
- `OllamaClient`: Interfaccia con modelli linguistici eseguiti localmente tramite la piattaforma `Ollama`. Il client comunica con un'istanza attiva di Ollama accessibile tramite `localhost:11434`, inviando richieste HTTP REST alla sua API interna. Per poter funzionare correttamente è necessario che il modello scelto (ad esempio `llama3.2`) sia preventivamente scaricato e disponibile

nel runtime locale di Ollama. Questo consente un'esecuzione completamente offline, tempi di risposta ridotti e l'indipendenza dalla disponibilità di token.

- `GitHubModelsClient`: Effettua una connessione remota ai modelli linguistici ospitati su `GitHub Models` (ad esempio varianti di GPT o Claude). L'accesso avviene tramite chiamate API autenticate mediante una **API Key**. Tale client offre maggiore capacità di ragionamento e contesto rispetto alle esecuzioni locali, risultando particolarmente adatto alle fasi di generazione del codice o di creazione di contratti complessi, dove è richiesta una comprensione semantica più profonda.

Grazie all'astrazione fornita dalla classe `LLMClient`, gli agenti possono essere istanziati con qualunque client senza modificare il codice applicativo, favorendo così estendibilità e sostituibilità dei modelli.

5.3.3 Selezione dinamica del client LLM

La selezione del client è gestita centralmente dall'`Orchestrator` tramite l'enumerazione `LLMClientType`, che può assumere i valori: `MOCK`, `OLLAMA` o `GITHUB_MODELS`. In base alla configurazione dell'utente, l'Orchestrator istanzia automaticamente il client appropriato e lo propaga a tutti gli agenti del sistema.

5.4 Agents

5.4.1 Classe base `Agent`

La classe `Agent` costituisce la **superclasse astratta** da cui ereditano gli altri *Agents* del sistema. Fornisce l'infrastruttura comune per la comunicazione con il modello linguistico, la gestione dei log e il parsing delle risposte, garantendo coerenza operativa e riduzione della ridondanza nel codice.

Ruolo e responsabilità

L'obiettivo principale della classe è definire un'interfaccia standard e un insieme di utilità condivise, in modo che ogni agente possa concentrarsi sulle proprie funzioni specifiche senza dover implementare ripetutamente la logica di base per:

- la comunicazione con il client LLM (`self.client`);
- la registrazione dei log (`self.log()`);
- l'estrazione robusta del contenuto dalle risposte del modello (`extract_response()`).

Struttura generale

La classe si compone di tre blocchi principali di funzionalità:

1. **Inizializzazione e configurazione del client:** Il costruttore riceve un oggetto client, che rappresenta l'interfaccia verso il modello linguistico (ad esempio `MockLLMClient`, `OllamaClient`, `GitHubModelsClient`). Ogni agente eredita tale client e lo utilizza per generare risposte ai propri prompt specializzati.
2. **Sistema di logging centralizzato:** La funzione `set_logger(on_log)` consente di collegare un callback di logging condiviso, utile per visualizzare o registrare i messaggi in un'interfaccia grafica (*frontend*) o in un file di log.

Il metodo `log(message)` verifica se il callback è disponibile e, in caso contrario, stampa il messaggio su standard output, assicurando un tracciamento uniforme del flusso di esecuzione tra gli agenti.

3. **Parsing e normalizzazione delle risposte JSON:** Il metodo `_extract_json(response)` costituisce il componente più sofisticato della classe. Esso implementa una serie di strategie progressive per estrarre in modo robusto un oggetto JSON valido dalle risposte del modello, anche quando queste contengono formattazioni Markdown o testo spurio.

Algoritmo di estrazione della reponse

La procedura di estrazione è costruita in più fasi, applicate in ordine di priorità:

1. Tentativo diretto di `json.loads(response)`. Se la stringa è un JSON valido, il parsing termina immediatamente.
2. Ricerca di blocchi di codice Markdown come ```` ```json``` ````, ```` ```python``` ````, ```` ```reverty``` ```` o ```` ```toon``` ````, da cui viene estratto il contenuto e decodificato.
3. Ricerca euristica del primo e dell'ultimo carattere graffa ({ e }) per delimitare manualmente l'oggetto JSON principale.
4. Analisi iterativa con bilanciamento di parentesi graffe, utile per individuare l'oggetto JSON più esterno in risposte complesse o nidificate.

Se nessuno dei metodi ha successo, la funzione restituisce la risposta grezza, che viene gestita a livello superiore dall'agente chiamante.

Estendibilità e riuso

Tutti gli agenti del sistema estendono la classe `Agent`, ridefinendo i propri metodi di elaborazione specifica ma riutilizzando:

- il client LLM condiviso;
- il sistema di logging uniforme;
- il meccanismo di parsing resiliente.

Questa architettura favorisce la manutenibilità, l'estendibilità e la coerenza operativa, rendendo semplice l'aggiunta di nuovi agenti o la modifica del comportamento esistente senza duplicare codice.

5.4.2 Evaluator Agent

L'agente `EvaluatorAgent` ha il compito di stimare la complessità della richiesta dell'utente (*user prompt*) al fine di fornire all'**Architect Agent** un'informazione quantitativa utile per calibrare la progettazione del *Technical Contract*. Si tratta del primo passo logico del flusso di elaborazione, che consente di tradurre un'esigenza descritta in linguaggio naturale in un livello di difficoltà computazionale interpretabile dal sistema.

Funzionalità principale

La funzione principale `evaluate_request()` riceve come input il prompt dell'utente e interroga il modello linguistico (LLM) attraverso il metodo `client.generate()`, utilizzando un *system prompt* specifico (`EVALUATOR_SYSTEM_PROMPT`) che istruisce il modello a valutare la complessità concettuale e algoritmica del compito.
Per garantire robustezza e affidabilità del processo, l'agente implementa una fase di **validazione iterativa** della risposta del modello. Dopo il primo tentativo di parsing, il metodo verifica che la chiave `complexity` esista e sia un valore intero. Se la risposta non rispetta tale condizione, l'agente ripete la richiesta al modello fino a un numero massimo di tentativi definito da `MAX_EVALUATION_RETRIES`.

Il risultato atteso è una risposta in formato TOON, contenente almeno la chiave `complexity`, il cui valore numerico (un intero compreso tra 1 e 10) rappresenta la difficoltà stimata della richiesta. Tale valore orienta la fase successiva di costruzione del *Technical Contract* da parte dell'agente `ArchitectAgent`.

Ruolo nel sistema

Dal punto di vista del flusso complessivo, l'`EvaluatorAgent` rappresenta la fase di *analisi semantica preliminare* del prompt. Determinando la complessità della richiesta, esso fornisce un parametro guida che influenza:

- la granularità del *Technical Contract* generato dall'architetto;
- la profondità e il livello di astrazione del codice prodotto dal `CoderAgent`;
- la quantità e la difficoltà dei test generati successivamente.

5.4.3 Architect Agent

L'`ArchitectAgent` svolge il ruolo di *progettista del Technical Contract* all'interno del sistema multi-agente. La sua funzione è quella di tradurre il prompt dell'utente, con l'aggiunta della complessità valutata dall'**Evaluator Agent**, in una specifica formale strutturata. Tale *technical contract* rappresenta una descrizione ad alto livello dei requisiti funzionali e delle caratteristiche del programma da generare.

Funzionalità principale

Il metodo principale `create_contract()` riceve due parametri:

- `user_prompt`: la descrizione testuale del compito o della funzionalità da implementare;
- `complexity`: un valore numerico che rappresenta il livello di complessità del problema, compreso tra 1 e 10.

Sulla base del valore di complessità, l'agente seleziona dinamicamente quale *system prompt* utilizzare:

- `ARCHITECT_SYSTEM_PROMPT_SIMPLE`, per richieste con complessità bassa o media (complessità ≤ 5);
- `ARCHITECT_SYSTEM_PROMPT_COMPLEX`, per richieste di complessità superiore.

Tale distinzione consente di adattare il comportamento del modello linguistico al livello di astrazione necessario: per richieste semplici viene utilizzata una struttura sintetica e diretta, mentre per problemi più complessi viene attivata una guida più dettagliata che incoraggia la generazione di specifiche articolate.

Struttura e contenuto del *Technical Contract*

Il *technical contract* prodotto dall'`ArchitectAgent` costituisce una rappresentazione formale intermedia che descrive:

- gli obiettivi funzionali del programma;
- le specifiche di input e output;
- i vincoli logici e strutturali (ad esempio tipologie di dati, precondizioni, invarianti);
- eventuali linee guida per la generazione del codice e dei test.

Questa struttura funge da "ponte semantico" tra la descrizione in linguaggio naturale fornita dall'utente e la generazione effettiva del codice a cura del `CoderAgent`.

5.4.4 Coder Agent

Il `CoderAgent` è responsabile della generazione effettiva del codice a partire dal *Technical Contract* fornito dall'`ArchitectAgent`. È concepito come un *agente di compilazione e validazione*, capace non solo di produrre codice, ma anche di verificarne la correttezza sintattica, semantica e stilistica in modo iterativo.

Funzionalità principale

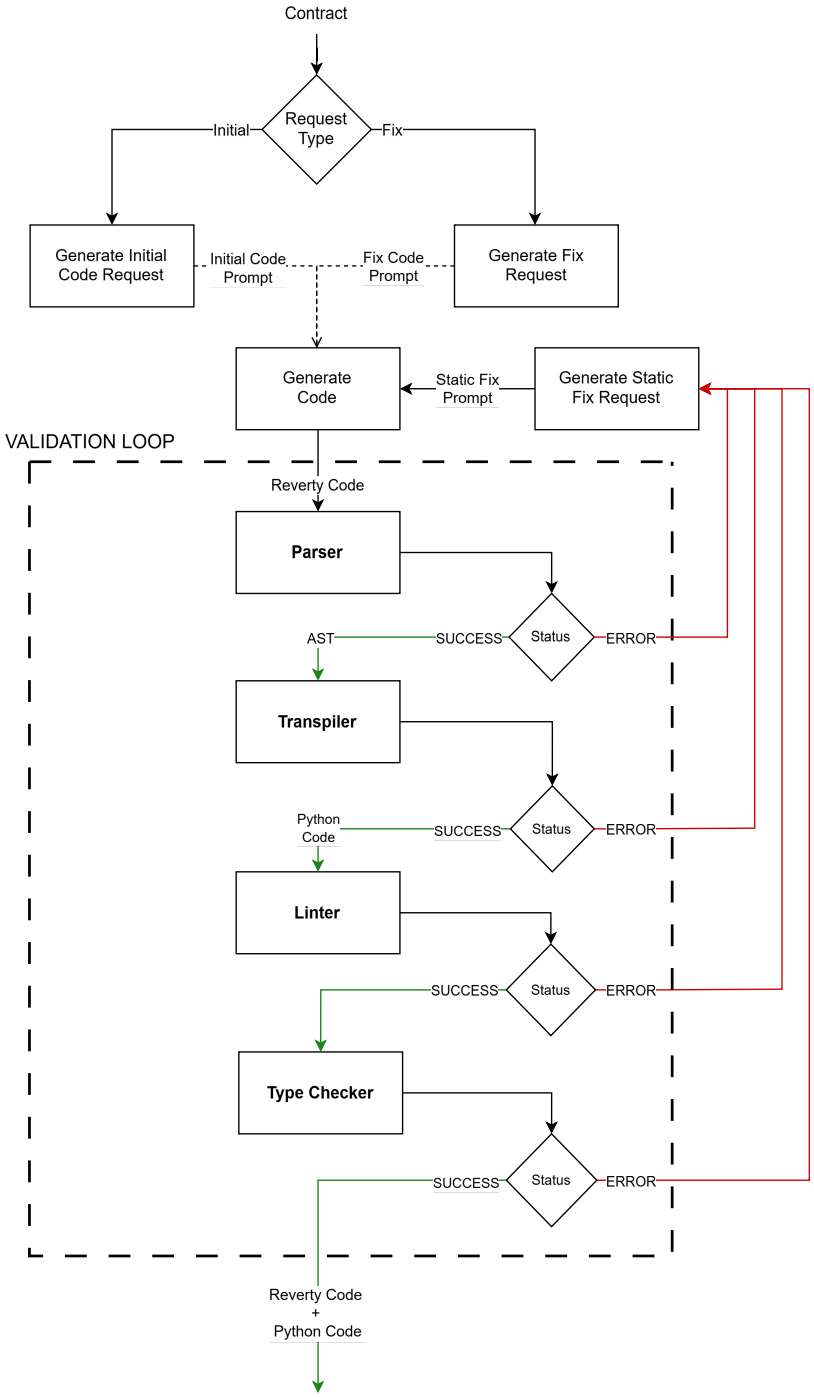
L'agente riceve come input il *Technical Contract* e genera due forme parallele di codice:

- il **codice Reverty**;
- il **codice Python**, ottenuto tramite traspilazione del codice Reverty, pronto per test ed esecuzione.

A tale scopo, il metodo principale `build_initial_code()` costruisce un *prompt* mirato utilizzando la funzione `generate_initial_code_request()`, lo invia al LLM con un system prompt dedicato (`CODER_SYSTEM_PROMPT`) e riceve in risposta un blocco di codice Reverty. Segue una fase di validazione che garantisce la coerenza del risultato prima di passarlo agli agenti successivi.

Pipeline di validazione

Il `CoderAgent` integra al proprio interno diversi sottosistemi specializzati



Il metodo interno `_validate_code()` realizza un ciclo iterativo di validazione multipla fino a un numero massimo di tentativi (`MAX_VALIDATION_ITERATIONS`). Ad ogni iterazione vengono eseguite in sequenza le seguenti operazioni:

1. **Parsing**: traduzione del codice Reverty in un *Abstract Syntax Tree* (AST);
2. **Transpiling**: conversione dell'AST in codice Python eseguibile;
3. **Linting**: verifica della conformità stilistica e della leggibilità del codice;
4. **TypeChecking**: controllo della coerenza dei tipi e individuazione di eventuali errori semantici.

Ogni fase produce un oggetto `AnalysisResult` che indica l'esito (`Status.SUCCESS` o `Status.ERROR`) e il messaggio associato. In caso di errore, l'agente è in grado di attivare meccanismi di auto-correzione automatica basati su richieste statiche all'LLM, costruite tramite la funzione `generate_static_fix_request()`. Se una fase fallisce, il codice Reverty viene corretto e il processo di validazione viene ripetuto. Solo quando tutte le fasi restituiscono uno stato di successo, l'agente produce in output la coppia `(Reverty, Python)` e lo stato `SUCCESS`.

Modalità di correzione

Oltre alla generazione iniziale, il `CoderAgent` gestisce la modalità di *code fixing* tramite il metodo `fix_code()`, che utilizza prompt specifici per correggere errori rilevati durante la fase di testing (`generate_test_fix_request()`). Le correzioni avvengono sempre nel contesto del *Technical Contract*, garantendo coerenza semantica tra specifiche e codice.

5.4.5 Test Generator Agent

Il `TestGeneratorAgent` si occupa della generazione automatica dei test unitari basati sul framework `pytest`, a partire dal *technical contract* e dal codice Python prodotto dal `CoderAgent`. Questa componente consente di garantire che ogni implementazione generata sia accompagnata da un insieme coerente e completo di test, rendendo possibile la validazione automatica dell'intero sistema.

Funzionalità principale

Il metodo `build_tests()` riceve in ingresso il *technical contract* e il codice Python. Tramite la funzione ausiliaria `generate_test_generator_request()`, viene costruito un prompt mirato contenente:

- la descrizione della funzione e dei requisiti funzionali presenti nel *Technical Contract*;
- la firma della funzione e i suoi tipi di input/output;
- il codice Python da testare.

Questo prompt viene poi inviato al LLM con un *system prompt* specifico (`TESTER_GENERATOR_SYSTEM_PROMPT`), progettato per guidare il modello nella generazione di test conformi a `pytest`.

Correzione automatica dei test

Oltre alla generazione iniziale, l'agente è in grado di eseguire una *correzione automatica* dei test falliti attraverso il metodo `fix_tests()`.
In questa modalità, il sistema costruisce un nuovo prompt tramite la funzione `generate_test_generator_fix_request()`, che include:

- il *technical contract* di riferimento;
- il codice Python corrente;
- il log degli errori (`test_errors`) riscontrati durante l'esecuzione precedente.

Il modello linguistico riceve quindi istruzioni per analizzare le cause dei fallimenti e produrre una versione corretta della suite di test. Questo approccio iterativo consente di mantenere allineati codice e test, migliorando progressivamente la copertura e la robustezza del sistema.

5.4.6 Tester Agent

Il `TesterAgent` rappresenta la componente di *validazione dinamica* del sistema multi-agente.
Il suo compito è eseguire i casi di test generati dall'agente `TestGeneratorAgent` sul codice Python prodotto dal `CoderAgent`, analizzare i risultati e classificare gli errori individuati. In base all'esito dei test, il sistema è in grado di decidere se sia necessario correggere il codice, i test stessi o entrambi.

Funzionalità principale

Il metodo centrale `test()` riceve come parametri:

- il *technical contract*, che definisce i requisiti funzionali;
- il **codice Reverty** generato dal `CoderAgent`;
- il corrispondente **codice Python**;
- la suite di **test automatici**.

L'agente esegue i test effettivi utilizzando la classe ausiliaria `TestExecutor`, che si occupa di lanciare i casi di test in un ambiente di esecuzione controllato.

Esecuzione e raccolta dei risultati

L'output del modulo di esecuzione viene rappresentato come un oggetto di tipo `ExecutionResult`, che contiene:

- lo `status` complessivo dell'esecuzione (`SUCCESS` o `ERROR`);
- eventuali `code_failures`, ossia errori derivanti dal codice sorgente;
- eventuali `failed_tests`, ossia test non superati o mal definiti.

Analisi degli errori

Nel caso in cui i test falliscano, l'agente costruisce un prompt di analisi specifico tramite la funzione `generate_tester_request()`. Questo prompt include il *technical contract*, il codice Python e Reverty, la suite di test, e l'output degli errori. L'agente interroga quindi l'LLM utilizzando il `TESTER_SYSTEM_PROMPT`, richiedendo una classificazione strutturata dei fallimenti riscontrati.
Il modello deve restituire un TOON che distingue tra:

- `code_failures`: errori logici o di implementazione presenti nel codice;
- `test_failures`: problemi o incoerenze nella definizione dei test.

In questo modo, l'Orchestrator è in grado di stabilire se avviare una nuova iterazione di correzione del codice, dei test, o di entrambi (richiedendo una nuova esecuzione).

5.5 Logger e Utils

- **Logger**: Sistema di monitoraggio ereditato da tutti gli agenti, tramite la classe base Agent, registra l'intero flusso di esecuzione (scambio di messaggi tra gli agenti) per consentire il debugging e la visualizzazione in tempo reale della conversazione nel frontend.
- **Utils**: Raccolta di funzioni di supporto che gestiscono operazioni trasversali al sistema, come il caricamento di file, la formattazione di stringhe e, soprattutto, la visualizzazione dell'AST in forma leggibile per facilitare l'analisi e il debugging del codice generato.

6 Testing del Sistema

Data la natura del sistema, che integra componenti deterministici (parser, validatore sintattico) e componenti non deterministici (Large Language Models), il testing è stato progettato con l'obiettivo di garantire correttezza, riproducibilità e isolamento delle singole parti.

6.1 Strategia di Testing

La strategia adottata combina differenti livelli di testing:

- **Unit testing:** verifica dei singoli componenti in isolamento;
- **Integration testing:** interazione tra i componenti principali;

Questa suddivisione consente di individuare con precisione eventuali errori e di mantenere separata la validazione della logica di sistema dal comportamento stocastico degli LLM.

6.2 Unit Testing

La strategia di testing unitario si è concentrata sulla validazione isolata delle componenti fondamentali del sistema. Data la natura ibrida dell'architettura, è stato necessario distinguere nettamente tra il testing delle componenti deterministiche (Tools) e la verifica della logica di controllo degli agenti (Agents).

6.2.1 Tools

Le componenti che non dipendono dall'inferenza di un modello linguistico sono state sottoposte a una copertura di test rigorosa. Grazie alla natura formale del linguaggio Reverty, è stato possibile definire suite di test positivi e negativi con esito univoco.
In particolare, l'attività di validazione ha coperto:

- `Parser`: Verifica del corretto riconoscimento della grammatica. I test assicurano che ogni regola sintattica generi l'Abstract Syntax Tree (AST) atteso e che il parser sollevi le eccezioni corrette in presenza di codice malformato.

- `Transpiler`: Verifica della fedeltà semantica nella traduzione da AST Reverty a codice Python, garantendo che la logica originale non venga alterata durante la conversione.
- `Linter`: Validazione del motore di analisi statica per il riconoscimento di violazioni stilistiche e sintattiche nel codice generato.
- `TypeChecker`: Verifica del sistema di controllo dei tipi, assicurando che le incongruenze nelle firme delle funzioni o nell'uso delle variabili vengano correttamente segnalate prima dell'esecuzione.
- `TestExecutor`: Verifica del runner di test automatizzati, controllando la corretta esecuzione delle suite, il parsing dei risultati (Pass/Fail) e la cattura degli stack trace in caso di errore.

6.2.2 Agents

Per isolare la logica di gestione del dialogo e di elaborazione delle risposte, è stato implementato un componente di simulazione denominato `SequentialMockLLM`. Questo modulo sostituisce il client LLM reale restituendo sequenze di risposte predeterminate e deterministiche. Ciò ha permesso di:

- Simulare scenari di successo in cui l'agente riceve output conformi al formato di output atteso.
- Simulare scenari di fallimento (es. risposta malformata, violazioni di protocollo) per verificare che la logica di *retry* e gestione degli errori dell'agente reagisca correttamente, senza consumare token reali o introdurre latenza di rete.

6.3 Integration Testing

Sono stati definiti due scenari critici di integrazione:

- **Validation Loop:** Il test simula la generazione di codice sintatticamente errato per confermare che i tool di analisi statica intercettino l'errore tramite ed invochino il feedback di correzione segnalando l'errore in maniera adeguata, iterando fino alla risoluzione o al raggiungimento del limite di tentativi.
- **Orchestrator Loop:** Questo scenario verifica l'intero flusso di vita della richiesta, dall'input utente alla generazione finale. Testa il corretto passaggio di contesto tra gli agenti del sistema, assicurando che non vi sia perdita di informazioni durante le transizioni di stato.

L'obiettivo di questa fase non è valutare la correttezza logica del codice prodotto, ma garantire che ogni componente reagisca in modo coerente e controllato agli output delle altre parti del sistema.

6.4 MockLLM Client

Per ovviare ai problemi introdotti dall'utilizzo di un LLM, è stato introdotto un client simulato che sostituisce l'LLM reale durante le fasi di test. Il `MockLLMClient` restituisce risposte predefinite e controllate, consentendo di:

- simulare output sintatticamente corretti;
- introdurre errori intenzionali;
- testare casi limite;
- isolare il comportamento dell'architettura dalla generazione linguistica.

7 Conclusioni

7.1 Valutazione del sistema

L'architettura multi-agente sviluppata ha dimostrato di essere coerente con gli obiettivi prefissati, fornendo una pipeline stabile e testabile per la generazione automatica di codice. La suddivisione del sistema in moduli indipendenti ha facilitato il controllo del flusso di esecuzione e l'isolamento delle singole responsabilità, rendendo il sistema estendibile, scalabile e manutenibile.

L'utilizzo di Reverty come linguaggio si è rivelato particolarmente efficace per ridurre il ricorso a meccanismi di riproduzione mnemonica da parte dei modelli linguistici. La sintassi invertita, unita alla validazione formale tramite parser, ha costretto gli agenti ad applicare esplicitamente le regole sintattiche, consentendo una valutazione più oggettiva delle loro capacità di ragionamento strutturato.

Dal punto di vista sperimentale, gli agenti hanno mostrato capacità limitate di adattamento al linguaggio proposto. I casi di successo riguardano prevalentemente task di natura semplice, quali il calcolo del fattoriale o l'implementazione di una calcolatrice elementare, quest'ultima ottenuta solo in maniera discontinua e principalmente tramite modelli Cloud.

I modelli locali, in particolare, hanno evidenziato difficoltà significative anche nella risoluzione di problemi banali, come la definizione di una funzione per la somma di due numeri. Tale comportamento è attribuibile al fatto che i modelli non risultano addestrati sul linguaggio Reverty, ma ne incontrano la grammatica per la prima volta all'interno del *system prompt*, rendendo necessario un processo di adattamento che eccede le loro capacità di generalizzazione sintattica.

Nel complesso, i risultati ottenuti confermano la validità dell'approccio adottato e ne evidenziano il potenziale come strumento di analisi e sperimentazione nel contesto dei sistemi di generazione del codice basati su Large Language Models.

7.2 Sviluppi Futuri

Nonostante i risultati ottenuti confermino l'efficacia dell'architettura multi-agente proposta e il suo corretto funzionamento, il sistema presenta ampi margini di evoluzione.

- **Espansione della grammatica e dei costrutti supportati:** L'introduzione di tipi di dato quali **liste** e **dizionari**, unitamente all'implementazione della gestione delle eccezioni tramite blocchi **try-catch**, rappresenterà un passo fondamentale per rendere il sistema capace di gestire logiche applicative di livello professionale.
- **Integrazione di un sistema conversazionale real-time:** Una delle evoluzioni più significative riguarda l'implementazione di un'interfaccia chatbot interattiva basata sul paradigma *Human-in-the-loop*. Tale componente permetterebbe all'utente di intervenire direttamente nel ciclo di generazione, fornendo chiarimenti semantici o correggendo interpretazioni errate degli agenti in tempo reale.
- **Allocazione dinamica dei modelli:** Al fine di ottimizzare il bilanciamento tra prestazioni e costi computazionali, si prevede la possibilità di assegnare LLM differenti a ciascun agente in base alla specificità del compito. Si potrebbe, ad esempio, optare per modelli leggeri a bassa latenza per task di formattazione e logging, riservando modelli Cloud con maggiori capacità di ragionamento per l'agente *Orchestrator* e per le unità di generazione logica.