Code Rebirth: modernizzazione di applicazioni Legacy

A cura di:

Alessandro Iannone

Giuseppe Simone

Domenico de Gioia

Sommario

1. Introduzione

Le sfide della modernizzazione dei sistemi legacy

I sistemi legacy, fondati su tecnologie ormai obsolete, rappresentano una sfida crescente per molte imprese, a causa delle difficoltà legate alla manutenzione, all'aggiornamento e all'integrazione con soluzioni più moderne. Questi sistemi, sviluppati anni o decenni fa, sono stati per lungo tempo essenziali per le operazioni aziendali, ma oggi richiedono competenze tecniche specializzate sempre più difficili da reperire. La scarsità di professionisti capaci di gestire linguaggi e tecnologie superate comporta un aumento dei costi operativi, così come la necessità di mantenere hardware obsoleto, spesso non più supportato dai produttori.

Oltre alla complessità e ai costi di manutenzione, i sistemi legacy presentano limitazioni in termini di scalabilità. Spesso non sono in grado di gestire grandi volumi di dati o transazioni, compromettendo la capacità dell'azienda di espandersi e di rispondere alle esigenze del mercato globale. La scarsa integrazione con tecnologie avanzate, come il cloud computing, l'intelligenza artificiale e l'automazione dei processi, limita l'agilità delle imprese e ostacola l'adozione di soluzioni innovative che potrebbero migliorare l'efficienza e la competitività.

Un ulteriore aspetto critico è la vulnerabilità alla sicurezza informatica. I sistemi legacy, non ricevendo aggiornamenti di sicurezza regolari, diventano facili bersagli per attacchi informatici e malware, mettendo a rischio non solo l'integrità dei dati aziendali, ma anche la continuità operativa.

Nonostante la migrazione da un sistema legacy possa apparire complessa e costosa, è fondamentale per garantire la competitività e la sostenibilità a lungo termine. Le organizzazioni possono optare per un approccio graduale, modernizzando progressivamente le singole componenti, oppure adottare un'architettura ibrida, che consente di integrare le tecnologie legacy con nuove soluzioni. Sebbene sia un processo impegnativo, la modernizzazione offre vantaggi significativi, migliorando l'efficienza operativa, la sicurezza e la capacità di adattarsi rapidamente ai cambiamenti del mercato.

Processo di modernizzazione

Come riportato nel paper scientifico “Contemporary Software Modernization: Perspectives and Challenges to Deal with Legacy Systems”[1] il processo di modernizzazione di un sistema legacy, è suddiviso in diverse fasi che conducono gradualmente al passaggio verso un sistema moderno.

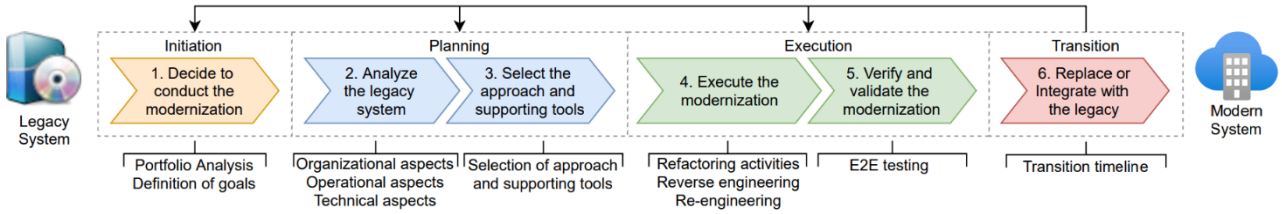

Figura 1 - Preliminary multi-perspective modernization workflow in the context of contemporary software development.

1. **Inizializzazione** – L’organizzazione effettua un’analisi del portfolio dei sistemi legacy e definisce gli obbiettivi da raggiungere con la modernizzazione.
2. **Pianificazione** – Si analizza il sistema legacy e si scelgono l’approccio e gli strumenti di supporto in base all’analisi condotta precedentemente.
3. **Esecuzione** – Vengono implementati i cambiamenti pianificati.
    a. **Modernizzazione** – Attraverso attività di refactoring, reverse-engineering o re-engineering, si adatta o aggiorna il sistema ai nuovi requisiti.
    b. **Validazione** – Si verifica che il sistema aggiornato funzioni correttamente.
4. **Transizione** – Consiste nel passaggio vero e proprio da sistema legacy a sistema modernizzato.

La nostra iniziativa si inserisce nella seconda fase. Questo progetto mira a fornire una soluzione innovativa che sfrutta l’intelligenza artificiale per modernizzare e ottimizzare il codice legacy, con particolare attenzione alla migrazione automatizzata dal COBOL a Java. Questo approccio riduce significativamente il tempo e i costi rispetto a una riscrittura manuale, minimizzando il rischio di errori. Il progetto non solo preserva il valore del codice esistente, ma lo rende più moderno e robusto per future evoluzioni aziendali.

[1] https://doi.org/10.48550/arXiv.2407.04017

2. Scopo e Obiettivi

Scopo

Lo scopo principale è semplificare e velocizzare la transizione da linguaggi di programmazione obsoleti a soluzioni moderne, riducendo al contempo i costi e il rischio di errori associati a una migrazione manuale. Il progetto si propone di sviluppare un sistema innovativo in grado di automatizzare il processo di migrazione del codice legacy COBOL, convertendolo in codice Java. In un contesto in cui molte organizzazioni si trovano a fronteggiare l'obsolescenza dei loro sistemi, questa soluzione rappresenta una risposta concreta a queste sfide, promuovendo l'adozione di tecnologie più avanzate e sostenibili.

Obiettivi Specifici

- Conversione automatizzata da COBOL a Java tramite AST
- Ottimizzazione del codice migrato per garantire efficienza e manutenibilità
- Comunicazione con repository di codice esterni
- Modernizzazione del codice legacy COBOL
- Realizzazione di una UI moderna e intuitiva

3. Criteri di Valutazione

Attinenza agli Obiettivi dell'Hackathon

Il sistema sviluppato risponde in modo diretto agli obiettivi dell'hackathon, mirando alla modernizzazione del codice legacy attraverso l'uso dell'intelligenza artificiale. Facilita la migrazione e l'ottimizzazione del codice COBOL verso Java, garantendo requisiti fondamentali come efficienza, sicurezza e manutenibilità.

Innovazione

L'aspetto innovativo di questo progetto risiede nell'impiego dell'intelligenza artificiale, in particolare un modello LLM tramite le API di OpenAI. Questo approccio consente la trasformazione dell'AST, generato attraverso l'uso della libreria ANTLR4, in codice Java. Inoltre, prevede la generazione automatica di documentazione, migliorando così la comprensibilità e la manutenibilità del codice risultante. Questa integrazione di tecnologie avanzate rappresenta un passo significativo verso la modernizzazione delle pratiche di sviluppo software.

4. Tecnologie e Strumenti Utilizzati

Framework e linguaggi

Python è stato il linguaggio scelto per il back-end, utilizzato per orchestrare il processo di trasformazione del codice COBOL in codice Java. Grazie alla sua semplicità e alla vasta disponibilità di librerie, Python ha reso possibile la gestione delle chiamate all'API di OpenAI e l'automazione del parsing con ANTLR. In aggiunta, l'uso del framework Flask ha facilitato la creazione di un'architettura web leggera e flessibile, consentendo l'integrazione delle diverse componenti del sistema. Python ha anche semplificato lo sviluppo di script per manipolare file, gestire i dati e condurre unit test, rendendolo uno strumento essenziale per l'implementazione e la gestione del progetto.

Angular è stato il framework scelto per il front-end del progetto, offrendo una struttura solida e scalabile per lo sviluppo dell'interfaccia utente. Grazie all'uso di HTML, CSS e TypeScript, è stato possibile creare componenti visivamente accattivanti e reattivi, migliorando notevolmente l'interazione con l'utente. HTML ha fornito la base per la struttura del contenuto, mentre CSS ha permesso di stilizzare l'interfaccia in modo coerente e professionale. TypeScript, con il suo supporto per i tipi statici e la programmazione orientata agli oggetti, ha facilitato lo sviluppo di codice più robusto e manutenibile. La vasta gamma di strumenti e librerie di Angular ha ulteriormente semplificato l'integrazione con il backend in Python, permettendo di effettuare chiamate API in modo fluido e sicuro. Inoltre, Angular ha reso possibile la gestione dello stato dell'applicazione e la navigazione tra le diverse sezioni dell'interfaccia, contribuendo a migliorare l'esperienza utente complessiva.

Generazione AST

ANTLR[2] (ANother Tool for Language Recognition) è stato utilizzato nel progetto per il parsing del codice COBOL e la generazione dell'AST (Abstract Syntax Tree). Attraverso la definizione di una grammatica specifica per COBOL, ANTLR ha permesso di analizzare il codice sorgente e generare un AST che è stato poi utilizzato per la generazione del codice Java tramite l'API di OpenAI, automatizzando così la migrazione del codice COBOL in modo efficace e riducendo i tempi e i costi di sviluppo.

API OpenAI

L'API di OpenAI è stata un componente cruciale del progetto, utilizzata per generare codice Java a partire dall'AST prodotto da ANTLR. Questa API ha consentito di sfruttare le potenzialità del modello gpt-4o per trasformare la rappresentazione strutturata del codice COBOL in un codice sorgente Java funzionale. Inoltre, l'API di OpenAI è stata utilizzata per generare un'auto documentazione del codice, facilitando la comprensione e l'uso del software sviluppato. Grazie alla sua flessibilità e capacità di elaborazione del linguaggio naturale, l'API ha reso possibile un processo di migrazione più efficiente e automatizzato.

[2] https://github.com/antlr/antlr4

5. Architettura della Soluzione

Architettura Generale

Lo schema riportato rappresenta un'architettura software che combina un'interfaccia front-end, microservizi back-end e un modello di linguaggio su larga scala (LLM), rappresentato dalle API di OpenAI. Questa architettura sfrutta richieste HTTP per la comunicazione tra le varie componenti, garantendo un'interazione fluida e modulare.

Di seguito, una descrizione dettagliata di ogni parte del sistema.

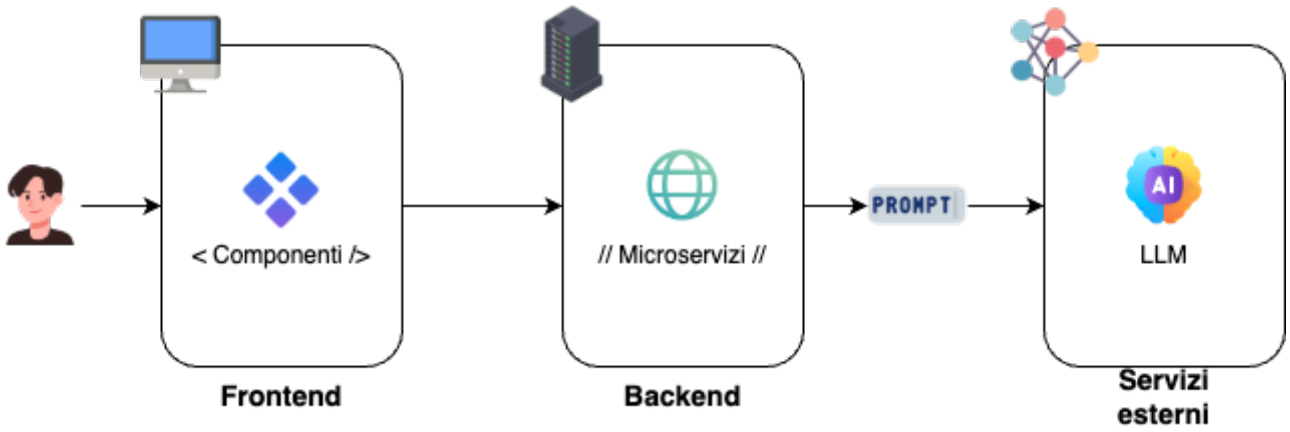

Figura 2 - Schema architetturale della soluzione Refactacy.

Utente e Interfaccia (Front-End)

L'utente interagisce con un'interfaccia web sviluppata utilizzando Angular, un framework JavaScript per la creazione di applicazioni front-end dinamiche. Angular gestisce la visualizzazione e l'interazione tramite componenti come form, bottoni e campi di input. Offre inoltre un robusto sistema di Routing che permette di navigare tra diverse pagine senza dover ricaricare l'intera applicazione.

Quando l'utente invia dati o interagisce con l'interfaccia, Angular genera delle richieste HTTP (ad esempio POST, GET, PUT, DELETE) verso il back-end, comunicando con i microservizi per elaborare i dati o eseguire operazioni specifiche. Le risposte vengono poi visualizzate e aggiornate in tempo reale grazie al two-way data binding di Angular, sincronizzando i dati tra la vista e il modello.

Infine, Angular include strumenti avanzati per il testing e l'ottimizzazione delle performance, rendendolo ideale per lo sviluppo di applicazioni web di grandi dimensioni e ad alta interattività.

Microservizi (Back-End)

Al centro dell'architettura, troviamo una serie di microservizi sviluppati in Flask, un framework Python leggero ideale per la creazione di applicazioni modulari e API RESTful. Ogni microservizio ha una responsabilità specifica (come autenticazione, gestione dati o logica di business) ed è esposto tramite endpoint HTTP. Offre un controllo granulare su ogni richiesta HTTP e permette di lavorare con formati di dati come JSON in modo nativo.

Flask permette di definire facilmente le route (URL) della tua applicazione tramite decoratori Python. Puoi mappare URL specifici a funzioni che gestiscono richieste http. Offre anche la possibilità di creare rotte dinamiche, che possono accettare parametri dalle URL per adattare il comportamento delle pagine o delle API.

Quando Angular invia richieste HTTP al back-end, queste richieste vengono indirizzate al microservizio appropriato, che elabora i dati in base alla logica implementata. Flask processa le richieste, esegue l'elaborazione necessaria e, a sua volta, può comunicare con altri microservizi o con servizi esterni per completare il lavoro.

Inoltre, supporta operazioni asincrone tramite la gestione di async e await, migliorando l'efficienza nelle richieste concorrenti.

Flask offre un debugger integrato che aiuta a tracciare errori e problemi durante lo sviluppo, mostrando stack trace dettagliati e consentendo anche un'interfaccia interattiva per esaminare il contesto dell'errore. Supporta facilmente il testing unitario e funzionale, facilitando la creazione di test automatizzati per verificare che le API o le funzionalità dell'applicazione funzionino come previsto. L'ambiente di testing può essere facilmente configurato tramite librerie come unittest o pytest.

Prompt e LLM (Large Language Model - OpenAI API):

Uno dei principali compiti dei microservizi è generare un prompt a partire dai dati utente o dalle informazioni elaborate, che viene poi inviato al modello di linguaggio di OpenAI. Questa fase è cruciale, poiché un prompt ben formulato può influenzare significativamente la qualità e la rilevanza della risposta generata. La comunicazione con l'LLM avviene anch'essa tramite richieste HTTP, garantendo un'interazione standardizzata e scalabile tra i vari componenti del sistema.

Flask, tramite uno specifico microservizio, invia una richiesta HTTP alle API di OpenAI, utilizzando generalmente il metodo POST per inoltrare il prompt al modello di linguaggio (come GPT-3 o successivi). La richiesta HTTP contiene il prompt nel corpo della richiesta, in formato JSON o in altro formato adatto, e può includere parametri aggiuntivi come la temperatura, che determina il livello di creatività della risposta, e il numero massimo di token, che limita la lunghezza della risposta.

L'LLM elabora il prompt e, grazie alle sue capacità avanzate di machine learning, genera una risposta intelligente o un testo basato sui dati forniti, riflettendo le sue conoscenze su una vasta gamma di argomenti. Questo processo avviene in tempo reale, permettendo un'interazione dinamica con l'utente. Le API di OpenAI rispondono nuovamente al microservizio con un output sotto forma di risposta HTTP. La risposta può essere ulteriormente elaborata dal microservizio prima di essere inviata al front-end, permettendo personalizzazioni come la formattazione o l'adattamento del contenuto alle esigenze specifiche dell'utente.

Vantaggi

- **Comunicazione via HTTP**: L'intero sistema, dal front-end Angular, attraverso i microservizi Flask, fino alle API di OpenAI, comunica tramite richieste HTTP. Questo approccio standard rende il sistema interoperabile e facile da integrare con altre applicazioni o servizi esterni.
- **Potenza dell'Intelligenza Artificiale**: Integrando le API di OpenAI tramite HTTP, è possibile sfruttare le capacità avanzate di elaborazione del linguaggio naturale offerte dai modelli di linguaggio su larga scala (LLM), migliorando l'interazione e le risposte fornite agli utenti.
- **Facilità di Integrazione**: L'uso di HTTP come protocollo standard per tutte le comunicazioni rende l'architettura facile da integrare con altri sistemi o servizi esterni, permettendo future estensioni o integrazioni.
- **Modularità e Scalabilità**: L'uso dei microservizi permette di mantenere il sistema modulare e scalabile, dove ogni componente è indipendente e può essere aggiornato o scalato in base al carico.

Flusso del Processo

Questo capitolo descrive il processo di modernizzazione del codice COBOL, partendo dall'input del codice legacy fino alla sua conversione automatizzata in Java. Attraverso l'uso di tecnologie come ANTLR4 per la generazione dell'AST e i modelli di linguaggio AI per la traduzione e la documentazione, il sistema garantisce una transizione fluida ed efficiente verso un linguaggio moderno. Ogni fase del processo contribuisce a preservare la logica di business originale, rendendola più accessibile e compatibile con le infrastrutture attuali.

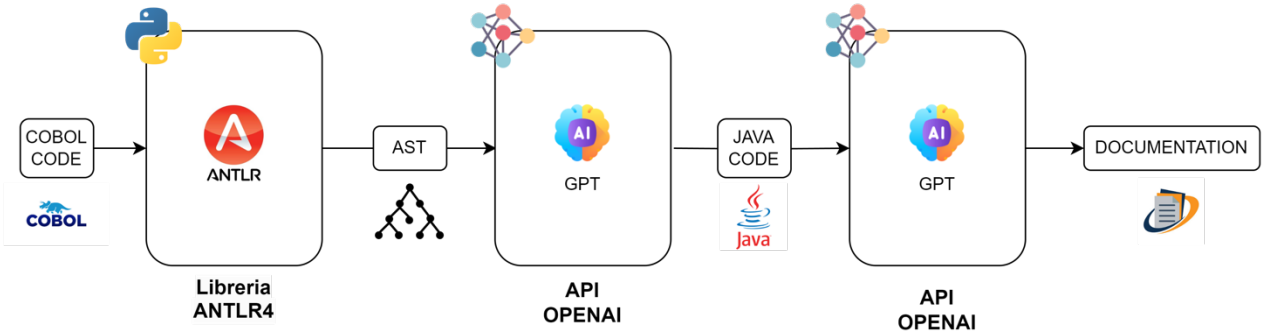

Figura 3 - Pipeline

Input del codice COBOL

L'utente inizia il processo fornendo il codice sorgente scritto in COBOL tramite collegamento a un repository GitHub. Questo codice rappresenta una logica di business legacy preesistente che deve essere modernizzata o integrata in un nuovo sistema. L'input viene acquisito tramite l'interfaccia dell'applicazione.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. HelloWorld.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.

PROCEDURE DIVISION.
DISPLAY "Hello World".
STOP RUN.
```

Generazione dell'AST tramite ANTLR4

Una volta ricevuto il codice COBOL, viene inviato al back-end, dove un microservizio Flask si occupa di elaborarlo. Utilizzando ANTLR4 (Another Tool for Language Recognition), una potente libreria per la creazione di parser, il microservizio genera un AST (Abstract Syntax Tree). Questo passo è cruciale perché permette di rappresentare il codice COBOL in una struttura intermedia rispetto alla traduzione finale e gerarchica, consentendo una facile analisi della sintassi e della logica del programma. L'AST scompone il codice sorgente in nodi che rappresentano le varie istruzioni e costrutti del linguaggio COBOL, rendendolo più facile da processare nel passaggio successivo.

```
def ASTGenerator(payload):
    input_stream = InputStream(payload)
    lexer = Cobol85Lexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = Cobol85Parser(stream)
    tree = parser.startRule()
    ast = tree.toStringTree(recog=parser)
    return ast
```

L'AST funge da base per tutte le elaborazioni successive, poiché consente al sistema di "comprendere" il flusso logico del codice COBOL in una forma strutturata. Grazie a questa rappresentazione, diventa possibile effettuare operazioni complesse come la traduzione del codice in altri linguaggi, analizzando le dipendenze tra variabili, funzioni, cicli e altre componenti del programma originale.

Conversione automatizzata dell'AST in codice Java tramite LLM

Una volta generato l'AST, il passo successivo consiste nel convertire automaticamente il codice COBOL in codice Java. Questo viene realizzato tramite un modello di linguaggio di grandi dimensioni (LLM) come quelli offerti dalle API di OpenAI. Il microservizio di Flask, dopo aver prodotto l'AST, invia un prompt ben formulato a un LLM tramite una richiesta HTTP, fornendo il contesto necessario per la conversione.

Il modello di linguaggio riceve come input l'AST e, grazie alla sua capacità di comprendere i linguaggi di programmazione, genera un codice equivalente funzionale in Java. La conversione automatica assicura che la logica di business originale venga mantenuta, ma tradotta in un linguaggio moderno e versatile come Java, che offre maggiore scalabilità e compatibilità con infrastrutture attuali. Il modello di linguaggio non si limita a una traduzione parola per parola, ma comprendere e replica le strutture logiche complesse del codice COBOL, convertendole nelle strutture equivalenti in Java e al suo paradigma di programmazione orientato agli oggetti.

Generazione di documentazione del codice Java tramite LLM

Durante il processo di conversione o immediatamente dopo, un altro importante compito del modello di linguaggio è generare documentazione automatizzata per il codice Java. La documentazione prodotta dal LLM descrive le principali funzioni e componenti del nuovo codice, facilitando la comprensione per gli sviluppatori che interagiranno con il codice generato.

```
def interagisci_con_gpt4(APIKEY, prompt):
    client = OpenAI(
        api_key=APIKEY,
    )
    stream = client.chat.completions.create(
        model="gpt-4o",
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        max_tokens=4096,
        temperature=0.2,
    )
    response_message = stream.choices[0].message.content
    return response_message
```

6. Unit Test

Gli unit test sono test automatici che verificano il corretto funzionamento delle singole unità di codice, come funzioni o classi, in isolamento dal resto del sistema. Nel progetto, gli unit test sono utilizzati per garantire che ogni singola funzione funzioni correttamente secondo i requisiti, riducendo i bug e migliorando la manutenibilità del codice.

Ogni file di test segue la convenzione t*est_<nome_modulo>.py* ed è stata utilizzata la libreria *unittest* integrata in Python per la creazione e gestione degli unit test.

Gli unit test coprono le principali funzioni utilizzate nel processo, tuttavia alcune funzionalità legate alla chiamata di API (OpenAI), in quanto le risposte delle chiamate possono cambiare a seconda del contesto, dell'input o anche a causa di aggiornamenti del modello. Questo rende difficile confrontare i risultati attesi in modo deterministico, mentre un test non deterministico è un test il cui risultato può variare tra esecuzioni identiche, senza che il codice sottoposto a test sia stato modificato. Questo comportamento è contrario al principio degli unit test, che dovrebbero essere deterministici e produrre sempre lo stesso risultato quando eseguiti nelle stesse condizioni.

Il progetto include test su:

- Estrazione di repository e file a partire da un link GitHub
- Creazione di un file unico che unisce più file .dat
- Creazione di un file unico che unisce tutti i file cobol, comprendendo i file .cpy e .dat
- Conversione del codice COBOL in AST tramite libreria ANTLR

File	Statements	Missed	Coverage
cobol_append.py	60	3	95%
cobol_to_ast.py	12	0	100%
test_cobol_append.py	13	1	92%
test_cobol_to_ast.py	11	1	91%
test_git_utils.py	16	1	94%
utils/__init__.py	0	0	100%
utils/gitUtils.py	48	14	71%
TOTAL	**160**	**20**	**88%**

È stato utilizzato il modulo unittest della libreria standard di Python per creare una suite di test automatizzata che copre le principali funzionalità del progetto. Inoltre, per valutare la coverage, è stato usato *coverage.py*, uno strumento che permette di misurare la percentuale di codice effettivamente eseguita durante l'esecuzione dei test. Attualmente, la coverage del progetto è 88%, ben oltre superiore all'obbiettivo minimo dell'80%.

In conclusione, gli unit test svolgono un ruolo fondamentale nel garantire la qualità e l'affidabilità del codice nel progetto. Sebbene alcune parti del sistema, come le chiamate API a OpenAI, presentino difficoltà nel testing deterministico, la copertura delle funzionalità critiche assicura che le principali operazioni siano validate e funzionino correttamente. Attraverso questi test, il progetto è in grado di identificare eventuali anomalie in fase di sviluppo e mantenere un'elevata manutenibilità del codice, assicurando una maggiore stabilità nel tempo.

7. Team di sviluppo

ALESSANDRO IANNONE

Laureato in ingegneria informatica e specializzato in Cybersecurity & cloud, con competenze riguardanti la programmazione backend, soprattutto in Java e Python, e lo sviluppo web. Da qualche mese ML/AI Engineer.

GIUSEPPE SIMONE

Studente di Informatica, appassionato di sviluppo backend, nello specifico Python e Java. Da diversi mesi Machine Learning Developer, spaziando in diversi campi dell'intelligenza artificiale come Gen AI e LLM e time-series forecasting.

DOMENICO de GIOIA

Studente del corso di laurea magistrale di Ingegneria Informatica, appassionato di backend (in particolare Python) e data science, con intento di specializzazione in ML/AI Engineer.

8. Conclusioni

L’uso combinato di strumenti statici e strumenti basati sull’intelligenza artificiale (AI) offre vantaggi complementari, specialmente in processi complessi come la migrazione del software. Gli strumenti statici (come i parser e gli AST generatori) garantiscono precisione e coerenza nell’analisi del codice, scomponendolo in strutture formali e riducendo il rischio di errori sintattici. Questi strumenti sono ideali per operazioni che richiedono una rigorosa aderenza a regole specifiche, come la validazione e la trasformazione del codice. Gli strumenti AI, invece, apportano flessibilità e adattabilità. I modelli di linguaggio come quelli di OpenAI comprendono non solo la sintassi, ma anche la logica e il contesto del codice. Ciò consente loro di gestire conversioni intelligenti tra linguaggi e ottimizzazioni, migliorando la leggibilità e manutenibilità del codice generato. Combinando entrambi gli approcci, si ottiene un’analisi affidabile e rigorosa grazie agli strumenti statici, arricchita dalla creatività e comprensione contestuale degli strumenti AI, per un risultato più completo ed efficace.

La documentazione generata dal LLM riduce significativamente il tempo necessario per comprendere il codice, offrendo descrizioni chiare e dettagliate delle funzioni, delle variabili e dei flussi logici. Questo supporto automatico permette una più rapida transizione verso il nuovo linguaggio, semplificando il lavoro degli sviluppatori e agevolando il passaggio di consegne tra team diversi. In questo modo, si minimizza il rischio di errori o fraintendimenti che potrebbero derivare da una cattiva interpretazione del codice, garantendo al contempo una maggiore coerenza nel progetto e una migliore manutenibilità a lungo termine.

Per quanto riguarda gli sviluppi futuri, il progetto potrebbe espandersi ulteriormente includendo una maggiore integrazione di tecniche di machine learning avanzate per migliorare la precisione nella conversione del codice e nelle ottimizzazioni successive. Si potrebbe lavorare sull'implementazione di algoritmi più complessi per gestire casi particolari nel codice COBOL, migliorando la flessibilità della soluzione. Un altro aspetto interessante sarebbe l'inclusione di un'interfaccia utente interattiva più intuitiva che permetta all'utente di visualizzare e modificare direttamente il codice convertito, apportando eventuali correzioni in modo collaborativo con l'AI. Inoltre, un'estensione dell'automazione potrebbe prevedere la gestione di altri linguaggi legacy, come Fortran, RPG o PL/I, ampliando così il campo di applicazione del progetto.

Link utili

Repository GitHub principale:

https://github.com/aleiann/Refactacy

Video di presentazione:

https://youtu.be/lBKG2NLCfx4