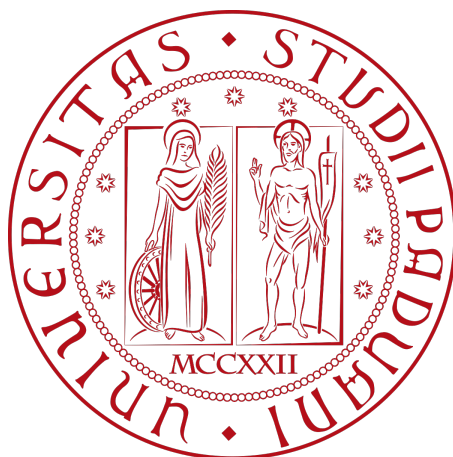


Università degli Studi di Padova

Dipartimento di Matematica «Tullio Levi-Civita»

Corso di Laurea in Informatica



**Archeologia Digitale e Rinascimento del Codice:
Modernizzazione dei Sistemi Legacy attraverso la
Migrazione Automatizzata COBOL - Java**

Tesi di laurea

Relatore

Prof. Tullio Vardanega

Laureando

Annalisa Egidi

Matricola: 1216745

Ringraziamenti

Sommario

L'elaborato descrive i processi, gli strumenti e le metodologie coinvolte nello sviluppo di un sistema di migrazione automatizzata per la modernizzazione di sistemi *legacy*, in particolare sulla conversione di applicazioni COBOL verso Java.

Nel dominio applicativo di interesse dell'elaborato:

- **Migrazione automatizzata:** è il processo di conversione di sistemi informatici da tecnologie obsolete a moderne architetture, preservando la logica di *business* originale;
- **Legacy Systems:** sistemi informatici datati ma ancora operativi, spesso critici per le organizzazioni, difficili da mantenere e integrare con tecnologie moderne (ingl. *legacy systems*).

Il progetto, sviluppato nel corso del tirocinio presso l'azienda Miriade Srl (d'ora in avanti **Miriade**), ha la peculiarità di aver esplorato inizialmente un approccio tradizionale basato su *parsing* deterministico per poi scegliere una soluzione innovativa basata su intelligenza artificiale generativa, dimostrando come l'AI possa cambiare drasticamente i tempi e la qualità dei risultati nel campo della modernizzazione *software*.

Struttura del testo

Il corpo principale della relazione è suddiviso in 4 capitoli:

Il **primo capitolo** descrive il contesto aziendale in cui sono state svolte le attività di tirocinio curricolare, presentando Miriade come ecosistema di innovazione tecnologica e analizzando le metodologie e tecnologie all'avanguardia adottate dall'azienda;

Il **secondo capitolo** approfondisce il progetto di migrazione COBOL - Java, delineando il contesto di attualità dei sistemi *legacy*, gli obiettivi del progetto e le sfide tecniche identificate nella modernizzazione di applicazioni COBOL verso architetture Java moderne;

Il **terzo capitolo** descrive lo sviluppo del progetto seguendo un approccio cronologico, dal *parser* tradizionale iniziale al *pivot* verso l'intelligenza artificiale, documentando le metodologie di lavoro, i risultati raggiunti e l'impatto trasformativo dell'AI sui tempi di sviluppo;

Il **quarto capitolo** sviluppa una retrospettiva sul progetto, analizzando le *lessons learned*, il valore dell'AI come *game changer* nella modernizzazione *software*, la crescita professionale acquisita e le prospettive future di evoluzione della soluzione sviluppata.

Le **appendici** completano l'elaborato con:

- **Acronimi:** elenco alfabetico degli acronimi utilizzati nel testo con le relative espansioni;
- **Glossario:** definizioni dei termini tecnici e specialistici impiegati nella trattazione;
- **Sitografia:** fonti consultate e riferimenti sitografici utilizzati per la redazione dell'elaborato.

Indice

Miriade: un ecosistema di innovazione tecnologica	1
L'azienda nel panorama informatico e sociale	1
Metodologie e tecnologie all'avanguardia	1
Architettura organizzativa	3
Investimento nel capitale umano e nella ricerca	5
Il progetto di migrazione COBOL-Java	8
Contesto di attualità	8
Obiettivi dello stage	10
Obiettivi principali	10
Obiettivi operativi	10
Metriche di successo	11
Vincoli	12
Pianificazione concordata	12
Valore strategico per l'azienda	13
Aspettative personali	14
Sviluppo del progetto: dall'analizzatore sintattico tradizionale all'AI (AI)	16
Configurazione iniziale e metodologia di lavoro	16
Primo periodo: immersione nel mondo COBOL	18
Studio del linguaggio e creazione progetti di prova	18
Mappatura degli schemi ricorrenti e analisi di traducibilità	21
Valutazione delle soluzioni esistenti	24
Secondo periodo: sviluppo dell'analizzatore sintattico tradizionale	26
Implementazione dell'analizzatore Java	27
Analisi critica e limiti dell'approccio	29
Terzo periodo: svolta verso l'intelligenza artificiale	32
Valutazione delle API di AI generativa	32
Progettazione del sistema basato su AI	34
Quarto periodo: implementazione della soluzione guidata da AI	36
Sviluppo dell'ingegneria delle istruzioni	36
Implementazione del traduttore completo	39
Generazione automatica di progetti Maven	42
Impatto dell'AI sui tempi di sviluppo	45

Analisi qualitativa dei risultati	46
Risultati quantitativi	48
Valutazioni retrospettive e prospettive future	51
Analisi retrospettiva del percorso	51
L'AI come <i>game changer</i> nella modernizzazione <i>software</i>	51
Crescita professionale e competenze acquisite	51
Valore della formazione universitaria nell'era dell'AI	51
<i>Roadmap</i> evolutiva e opportunità di sviluppo	51
Lista degli acronimi	52
Glossario	53
Sitografia	54

Elenco delle figure

Figura 1	Ecosistema Atlassian - dashboard Jira	2
Figura 2	Ecosistema Atlassian - dashboard Confluence	3
Figura 3	Struttura organizzativa delle divisioni Miriade	4
Figura 4	Funzioni nella sezione Analytics	5
Figura 5	Impegni etici e morali aziendali di Miriade	6
Figura 6	Interfaccia utente e codice COBOL tipici dei sistemi legacy	8
Figura 7	Confronto tra architettura monolitica dei mainframe e architettura moderna a microservizi	9
Figura 8	Diagramma di Gantt della pianificazione del progetto	13
Figura 9	Rappresentazione della metodologia Agile applicata al progetto	15

Miriade: un ecosistema di innovazione tecnologica

Miriade, come realtà nel panorama Information Technology (IT) italiano, si distingue per il suo approccio innovativo rispetto all'ecosistema completo del dato e alle soluzioni informatiche correlate. L'azienda, che ho avuto l'opportunità di conoscere durante il mio percorso di *stage*, si caratterizza per una filosofia aziendale orientata all'innovazione continua e all'investimento nel capitale umano, elementi che la rendono un ambiente particolarmente stimolante per la crescita professionale di figure *junior*.

L'azienda nel panorama informatico e sociale

Miriade si posiziona strategicamente nel settore dell'analisi dati e delle soluzioni informatiche, operando con quattro aree funzionali principali: *Analytics*, *Data*, *System Application* e *Operation*. L'azienda ha costruito nel tempo una solida reputazione nel mercato attraverso la capacità di fornire soluzioni innovative che rispondono non solo alle esigenze tecniche dei clienti, ma che prestano particolare attenzione alle relazioni umane e alle realtà del territorio.

Ciò che distingue *Miriade* nel contesto competitivo è la sua *vision* aziendale, che integra le competenze tecnologiche con una forte responsabilità sociale. L'azienda implementa attivamente azioni a supporto di società e cooperative del territorio, dimostrando come l'innovazione tecnologica possa essere un veicolo di sviluppo sociale ed economico locale. Questa attenzione alla dimensione sociale si riflette anche nell'approccio alle risorse umane, con una particolare propensione a individuare e coltivare giovani energie fin dalle scuole e università attraverso tirocini curriculari che permettono una crescita personale durante il percorso di studi.

La clientela di Miriade spazia tra i medi e grandi clienti, includendo sia realtà del settore privato che pubblico. Questa diversificazione del portfolio clienti permette all'azienda di confrontarsi con problematiche tecnologiche variegate, mantenendo una costante spinta all'innovazione e all'adattamento delle soluzioni proposte.

Metodologie e tecnologie all'avanguardia

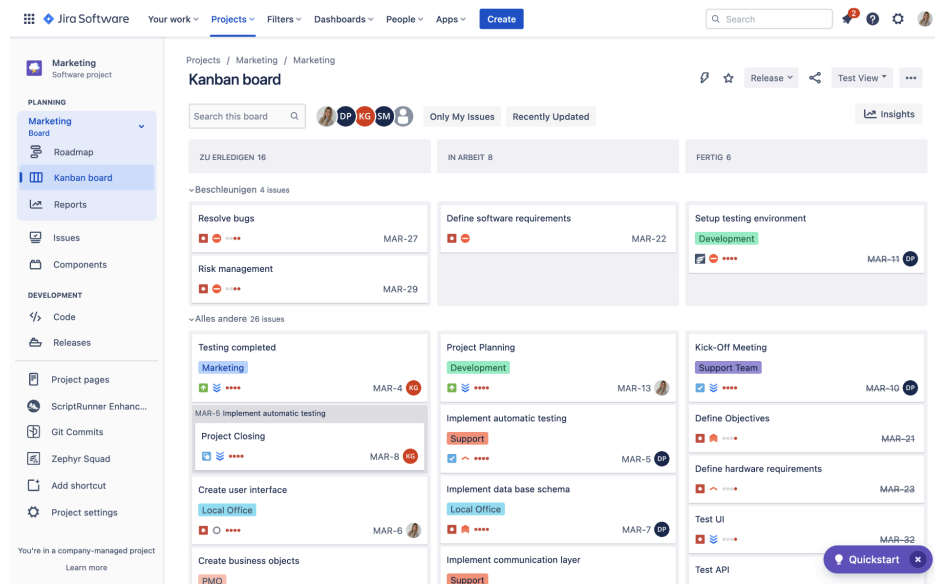
L'approccio metodologico di Miriade si fonda sull'adozione dell'*Agile* come filosofia operativa pervasiva, che permea tutti i processi aziendali e guida l'organizzazione del lavoro quotidiano. Durante il mio *stage*, ho potuto osservare direttamente come questa metodologia venga implementata attraverso *stand-up* giornalieri e *sprint* settimanali, creando un ambiente di lavoro dinamico e orientato agli obiettivi.

L'azienda utilizza sia *Kanban* che *Scrum*, adattando la metodologia alle specifiche esigenze progettuali e alle preferenze del cliente. Questa flessibilità metodologica dimostra la maturità organizzativa di Miriade e la sua capacità di adattare i processi alle diverse situazioni operative. Ho potuto constatare personalmente come gli *stand-up* mattutini fossero momenti fondamentali per l'allineamento del *team*, permettendo una comunicazione trasparente sullo stato di avanzamento delle attività e una rapida identificazione di eventuali impedimenti.

Lo *stack tecnologico* adottato riflette l'attenzione dell'azienda per gli strumenti di collaborazione e versionamento. L'*Atlassian Suite* costituisce la spina dorsale dell'infrastruttura collaborativa aziendale, utilizzata in modo strutturato e pervasivo per diverse finalità:

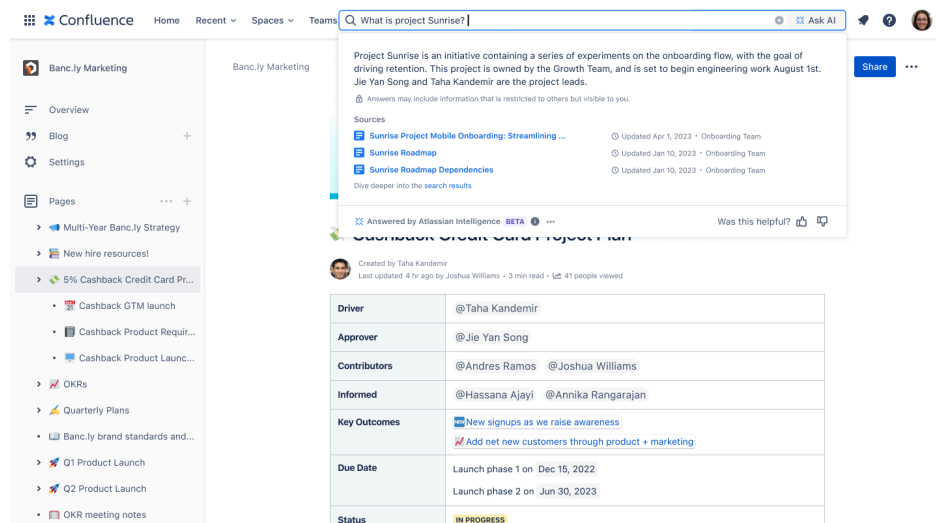
- **Confluence** per la gestione della *knowledge base* aziendale e la documentazione tecnica
- **Jira** per il *tracking* delle attività e la gestione dei progetti
- **Bitbucket** per il versionamento del codice e la collaborazione nello sviluppo

Durante il mio percorso, ho potuto apprezzare l'importanza che l'azienda attribuisce alla cultura del versionamento e della documentazione. Le Figura 1) e Figura 2 mostrano parte dell'ecosistema Atlassian integrato utilizzato quotidianamente in azienda, che ha rappresentato per me un elemento fondamentale nell'apprendimento delle pratiche professionali di sviluppo software.



Fonte: <https://www.peakforce.dev>

Figura 1: Ecosistema Atlassian - dashboard Jira



Fonte: <https://support.atlassian.com>

Figura 2: Ecosistema Atlassian - dashboard Confluence

La formazione continua sulle tecnologie emergenti è parte integrante della cultura aziendale. L'area *Analytics*, in particolare, mantiene un *focus* costante sull'esplorazione e implementazione di soluzioni basate su AI e Large Language Models (LLM), che rappresentano il naturale proseguimento di quello che precedentemente veniva incasellato come «*big data*» ed è parte integrante della strategia aziendale.

Architettura organizzativa

L'architettura organizzativa di Miriade si distingue per la sua struttura «piatta». L'azienda ha adottato un modello organizzativo che prevede solo due livelli gerarchici: l'amministratore delegato e i responsabili di area. Questa scelta strutturale facilita la comunicazione diretta e riduce le barriere comunicative, creando un ambiente di lavoro agile e responsabilizzante.

Le quattro aree funzionali principali - *Analytics*, *Data*, *System Application* e *Operation* - operano con un alto grado di autonomia, pur mantenendo una forte interconnessione attraverso aree trasversali. Queste aree trasversali, composte da persone provenienti dalle diverse divisioni, si occupano di attività di innovazione a vari livelli, come *DevOps*, *Account Management* e *Research & Development*. Questa struttura matriciale permette una *cross-fertilizzazione* delle competenze e favorisce l'innovazione continua. La rappresentazione visuale in Figura 3 illustra chiaramente questa struttura organizzativa interconnessa.



Figura 3: Struttura organizzativa delle divisioni Miriade

La divisione *Analytics*, nella quale ho avuto il piacere di lavorare, guidata da Arianna Bellino, conta attualmente 17 persone ed è in veloce crescita. Rappresenta il motore di innovazione dell'azienda, specializzandosi nella gestione del dato, dal dato grezzo all'analisi avanzata, tramite approcci e tecnologie AI e LLM *based*, con *focus* sull'automazione dei processi e alla riduzione delle attività routinarie. I membri del *team* non hanno ruoli rigidamente definiti, ma piuttosto funzioni che possono evolversi in base alle esigenze progettuali e alle competenze individuali. Ho osservato dipendenti che svolgevano funzioni diverse quali:

- Pianificazione e gestione progetti
- Attività di prevendita e consulenza
- Ricerca e sviluppo di nuove soluzioni
- Sviluppo *software* e *data analysis*

Questa fluidità organizzativa crea un ambiente stimolante dove ogni persona può contribuire in modi diversi, favorendo la crescita professionale multidisciplinare. Durante lo stage, ho potuto interagire con colleghi che ricoprivano diverse funzioni, beneficiando della loro esperienza e prospettive diverse. La varietà di funzioni all'interno della divisione *Analytics* è rappresentata in Figura 4, che evidenzia la natura dinamica e multifunzionale del team.



Figura 4: Funzioni nella sezione Analytics

Il ruolo dello stagista in questo ecosistema aziendale è particolarmente valorizzato. Non viene visto come una risorsa marginale, ma come parte integrante del team, con la possibilità di contribuire attivamente ai progetti e di proporre soluzioni innovative. Il sistema di tutoraggio è strutturato con l'assegnazione di un tutor dell'area specifica e di un mentor che può provenire anche da altre aree. Il tutor segue il percorso tecnico dello stagista, mentre il mentor fornisce supporto a livello emotivo e di inserimento aziendale.

Particolarmente apprezzabili sono gli incontri settimanali chiamati «tiramisù», dedicati ai nuovi entrati in azienda. Durante questi momenti, vengono analizzate le possibili difficoltà relazionali o comunicative riscontrate durante la settimana, con il supporto di una figura dedicata. Questo approccio dimostra l'attenzione dell'azienda non solo alla crescita tecnica, ma anche al benessere e all'integrazione dei propri collaboratori.

Investimento nel capitale umano e nella ricerca

L'investimento nel capitale umano rappresenta uno dei pilastri fondamentali della strategia aziendale di Miriade. Durante il mio stage, ho potuto constatare come l'azienda non si limiti a dichiarare l'importanza delle risorse umane, ma implementi concretamente politiche e programmi volti alla valorizzazione e crescita delle persone, come ad esempio incontri, riflessioni e azioni sulla Parità di Genere, sulla quale sono certificati come azienda.



Fonte: <https://www.miriade.it>

Figura 5: Impegni etici e morali aziendali di Miriade

Come si può osservare in Figura 5, l'azienda si esprime esplicitamente riguardo i propri valori.

Il processo di selezione riflette questa filosofia: l'azienda ricerca persone sensibili, elastiche, proattive e autonome, ponendo l'enfasi sulle caratteristiche personali piuttosto che esclusivamente sulle competenze tecniche pregresse, un approccio che permette di costruire *team* coesi e motivati, capaci di affrontare sfide tecnologiche in continua evoluzione.

I programmi di formazione continua sono strutturati e costanti. L'azienda investe significativamente nella crescita professionale dei propri dipendenti attraverso:

- Corsi di formazione tecnica su nuove tecnologie
- Certificazioni professionali
- Partecipazione a conferenze e *workshop*
- Sessioni di *knowledge sharing* interno
- Progetti di ricerca e sviluppo che permettono sperimentazione

Il rapporto consolidato con le università rappresenta un altro aspetto distintivo dell'approccio di Miriade al capitale umano. Gli *stage* non sono visti come semplici adempimenti formativi, ma come veri e propri laboratori di sperimentazione tecnologica. Nel mio caso specifico, il progetto di migrazione COBOL-Java è stato scelto appositamente per valutare le capacità di *problem solving* e apprendimento, con maggiore attenzione al processo seguito piuttosto che al solo risultato finale.

L'equilibrio tra formazione e produttività negli *stage* è gestito con attenzione. Inizialmente, lo *stage* è orientato totalmente sulla formazione, per poi evolvere gradualmente verso un bilanciamento equilibrato tra formazione e contributo produttivo quando lo stagista diventa sufficientemente autonomo. Nel mio caso però, trattandosi di *stage* curricolare per tesi,

l'intero percorso è stato focalizzato sulla formazione, permettendomi di esplorare in profondità tecnologie e metodologie senza la pressione di *deadline* produttive immediate.

L'investimento in risorse *junior* è visivamente significativo, questo approccio permette all'azienda di formare professionisti allineati con la propria cultura e metodologie.

In conclusione, Miriade si presenta come un ecosistema aziendale dove l'innovazione tecnologica e la valorizzazione del capitale umano si integrano sinergicamente. L'esperienza di stage in questo contesto ha rappresentato un'opportunità unica di crescita professionale, permettendomi di osservare e partecipare a dinamiche aziendali mature e orientate al futuro. La combinazione di una struttura organizzativa agile, metodologie all'avanguardia, forte investimento nelle persone e attenzione alla responsabilità sociale crea un ambiente ideale per affrontare le sfide tecnologiche contemporanee.

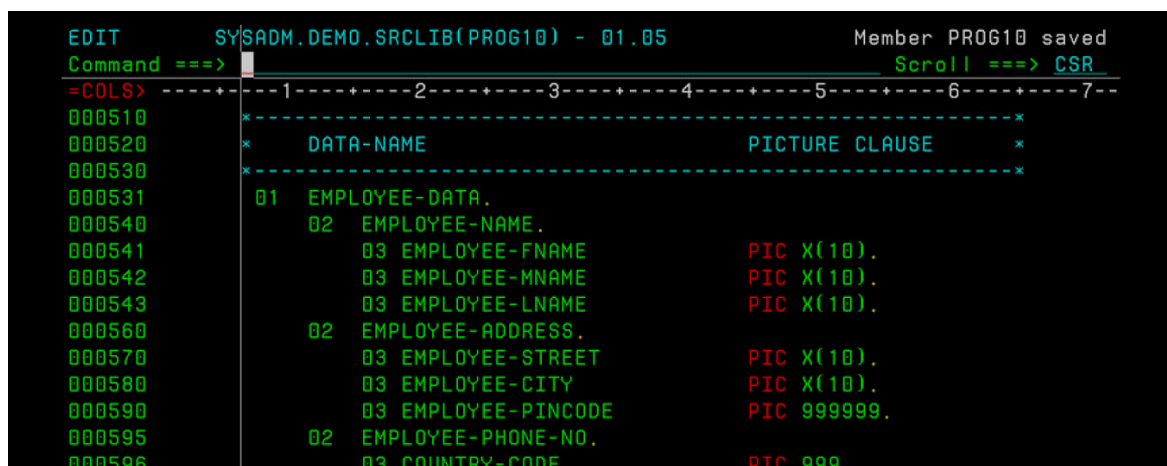
Il progetto di migrazione COBOL-Java

Il progetto di *stage* proposto da Miriade si inserisce in un contesto tecnologico di particolare rilevanza per il settore IT contemporaneo: la modernizzazione dei sistemi *legacy*. Durante il mio percorso, ho avuto l'opportunità di confrontarmi con una problematica comune a molte organizzazioni, in particolare nel settore bancario e assicurativo, dove i sistemi COBOL continuano a costituire l'impalcatura portante di infrastrutture critiche per il *business*.

Contesto di attualità

I sistemi legacy basati su Common Business-Oriented Language (COBOL) rappresentano ancora oggi una parte significativa dell'infrastruttura informatica di molte organizzazioni, specialmente nel settore bancario, finanziario e assicurativo. Nonostante COBOL sia stato sviluppato negli anni '60, ha una presenza significativa nelle moderne architetture.

Figura 6 mostra un esempio tipico di interfaccia utente e codice COBOL, che evidenzia il contrasto netto con le moderne interfacce grafiche e paradigmi di programmazione attuali. Questa differenza visuale è solo la punta dell'iceberg delle sfide che comporta il mantenimento di questi sistemi in un ecosistema tecnologico in rapida evoluzione.



```
EDIT      SYSADM.DEMO.SRCLIB(PROG10) - 01.05      Member PROG10 saved
Command ==>                                     Scroll ==> CSR
=COLS>  ---+---1---+---2---+---3---+---4---+---5---+---6---+---7---
000510  *-----*
000520  *   DATA-NAME                               PICTURE CLAUSE   *
000530  *-----*
000531  01  EMPLOYEE-DATA.
000540      02  EMPLOYEE-NAME.
000541          03  EMPLOYEE-FNAME                      PIC X(10).
000542          03  EMPLOYEE-MNAME                      PIC X(10).
000543          03  EMPLOYEE-LNAME                      PIC X(10).
000560      02  EMPLOYEE-ADDRESS.
000570          03  EMPLOYEE-STREET                    PIC X(10).
000580          03  EMPLOYEE-CITY                      PIC X(10).
000590          03  EMPLOYEE-PINCODE                    PIC 999999.
000595      02  EMPLOYEE-PHONE-NO.
000596          03  COUNTRY-CODE                      PIC 999.
```

Fonte: <https://overcast.blog>

Figura 6: Interfaccia utente e codice COBOL tipici dei sistemi legacy

La problematica della *legacy modernization* va ben oltre la semplice obsolescenza tecnologica. Durante il mio *stage*, attraverso l'analisi della letteratura e il confronto con i professionisti del settore, in particolare ho avuto modo di confrontarmi con la sig.ra Luisa Biagi, analista COBOL, ho potuto identificare come i costi nascosti del mantenimento di questi sistemi includano:

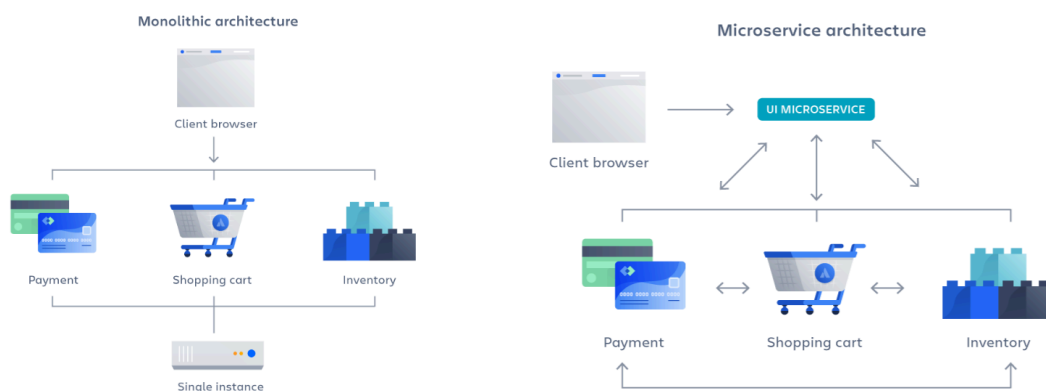
- La crescente difficoltà nel reperire sviluppatori COBOL qualificati [1]
- L'integrazione sempre più complessa con tecnologie moderne [2]
- I rischi operativi derivanti dall'utilizzo di piattaforme *hardware* e *software* che i *vendor* non supportano più attivamente [3]

Questi fattori si traducono in costi di manutenzione esponenzialmente crescenti e in una ridotta agilità nel rispondere alle esigenze di *business* in continua evoluzione.

I rischi associati al mantenimento di sistemi COBOL *legacy* nelle infrastrutture IT moderne sono molteplici e interconnessi:

- **Carenza di competenze:** La carenza di competenze specializzate crea una forte dipendenza da un *pool* sempre più ristretto di esperti, spesso prossimi al pensionamento [1].
- **Documentazione inadeguata:** La documentazione inadeguata o assente di molti di questi sistemi, sviluppati decenni fa, rende ogni intervento di manutenzione un'operazione ad alto rischio [4].
- **Incompatibilità tecnologica:** L'incompatibilità con le moderne pratiche di sviluppo come *DevOps*, *continuous integration* e *microservizi* limita in modo significativo la capacità delle organizzazioni di innovare e competere efficacemente nel mercato digitale [5].

Come illustrato in Figura 7, il contrasto tra l'architettura monolitica tipica dei sistemi *mainframe* e l'architettura moderna a microservizi evidenzia le sfide architetturali della migrazione. Questa differenza strutturale comporta non solo una riprogettazione tecnica, ma anche un ripensamento completo dei processi operativi e delle modalità di sviluppo.



Fonte: <https://www.atlassian.com>

Figura 7: Confronto tra architettura monolitica dei mainframe e architettura moderna a microservizi

La migrazione di questi sistemi verso tecnologie più moderne come Java rappresenta una sfida tecnica e una necessità strategica per garantire la continuità operativa e la competitività delle organizzazioni. Java, con il suo ecosistema maturo, la vasta *community* di sviluppatori e il supporto per paradigmi di programmazione moderni, si presenta come una delle destinazioni privilegiate per questi progetti di modernizzazione [6].

Obiettivi dello stage

Il macro-obiettivo era sviluppare un sistema prototipale di migrazione automatica da COBOL a Java che potesse dimostrare la fattibilità di automatizzare il processo di conversione, preservando la *business logic* originale e producendo codice Java idiomatrico e manutenibile.

Obiettivi principali

- **Esplorazione tecnologica:** Investigare e valutare diverse strategie di migrazione, dalla conversione sintattica diretta basata su regole deterministiche fino all'utilizzo di tecnologie di intelligenza artificiale generativa, identificando vantaggi e limitazioni di ciascun approccio.
- **Automazione del processo:** Sviluppare strumenti e metodologie che potessero automatizzare il più possibile il processo di conversione, riducendo l'intervento manuale e i conseguenti rischi di errore umano nella traduzione.
- **Qualità del risultato:** Garantire che il codice Java prodotto rispettasse standard di qualità professionale, con particolare attenzione alla leggibilità, manutenibilità e conformità alle convenzioni Java moderne.
- **Accessibilità della soluzione:** Fornire un'interfaccia utente (grafica o da linea di comando) che rendesse il sistema utilizzabile anche da personale non specializzato nella migrazione di codice.

Obiettivi operativi

Per rendere concreti e misurabili gli obiettivi principali, sono stati definiti obiettivi operativi specifici, classificati secondo tre livelli di priorità:

Obbligatori

- **OO01:** Sviluppare competenza nel linguaggio COBOL attraverso la produzione di almeno un progetto completo che includesse le quattro divisioni fondamentali (*Identification, Environment, Data e Procedure*)
- **OO02:** Esplorazione approfondita di diverse strategie di migrazione, dalla conversione sintattica diretta all'utilizzo di tecnologie di intelligenza artificiale generativa

- **OO03:** Implementare un sistema di conversione automatica che raggiungesse almeno il 75% di copertura delle divisioni
- **OO04:** Esplorare e documentare approcci distinti alla migrazione
- **OO05:** Completare la migrazione funzionante di almeno uno dei progetti COBOL sviluppati, validando l'equivalenza funzionale tra codice sorgente e risultato
- **OO06:** Produrre codice Java che rispettasse le convenzioni del linguaggio, includendo struttura dei *package*, nomenclatura standard e documentazione *JavaDoc*
- **OO07:** Fornire un'interfaccia utilizzabile (grafica o Command Line Interface (CLI)) per l'esecuzione del sistema di conversione
- **OO08:** Creare documentazione utente completa, includendo un *README* dettagliato con istruzioni di installazione, configurazione e utilizzo

Desiderabili

- **OD01:** Raggiungimento di una copertura del 100% nella conversione automatica del codice prodotto autonomamente
- **OD02:** Gestione efficace di costrutti COBOL complessi o non direttamente traducibili
- **OD03:** Implementazione di meccanismi di ottimizzazione del codice Java generato

Facoltativi

- **OF01:** Integrazione con sistemi di analisi statica per la verifica della qualità del codice generato
- **OF02:** Implementare un sistema di *reporting* dettagliato che producesse metriche sulla conversione, incluse statistiche di copertura, costrutti non convertiti e interventi manuali necessari
- **OF03:** Implementazione di funzionalità avanzate di *refactoring* del codice Java prodotto

Metriche di successo

Per valutare oggettivamente il raggiungimento degli obiettivi, erano state definite le seguenti metriche:

- **Copertura di conversione:** Percentuale di linee di codice COBOL convertite automaticamente senza intervento manuale
- **Equivalenza funzionale:** Corrispondenza interfaccia utente COBOL originale e Java convertito
- **Qualità del codice:** Conformità agli standard Java verificata tramite strumenti di analisi statica
- **Tempo di conversione:** Riduzione del tempo necessario per la migrazione rispetto a un approccio completamente manuale

- **Usabilità:** Capacità di utilizzo del sistema da parte di utenti con conoscenze base di programmazione

Vincoli

Il progetto si focalizzava sullo sviluppo di un sistema di migrazione automatica e questo aspetto caratterizzava le condizioni imposte per lo svolgimento del lavoro.

Vincoli temporali

- Durata complessiva dello *stage*: 320 ore
- Periodo: dal 05 maggio al 27 giugno 2025
- Modalità di lavoro ibrida: 2 giorni a settimana in sede, 3 giorni in modalità telematica
- Orario lavorativo: 9:00 - 18:00

Vincoli tecnologici

- Il sistema doveva essere sviluppato utilizzando tecnologie moderne e supportate
- Necessità di preservare integralmente la *business logic* contenuta nei programmi COBOL originali
- La soluzione doveva essere scalabile, capace di gestire progetti di diverse dimensioni
- Utilizzo strumenti di versionamento (*Git*) e di documentazione continua.

Vincoli metodologici

- Adozione dei principi *Agile* con *sprint* settimanali e *stand-up* giornalieri per allineamento costante
- Revisioni settimanali degli obiettivi con adattamento del piano di lavoro

Pianificazione concordata

La pianificazione del progetto seguiva un approccio flessibile, con revisioni settimanali che permettevano di adattare il percorso in base ai progressi ottenuti. La distribuzione delle attività era inizialmente stata organizzata come segue:

Prima fase - analisi e apprendimento COBOL (2 settimane - 80 ore)

- Studio approfondito del linguaggio COBOL e delle sue peculiarità
- Analisi di sistemi COBOL
- Creazione di programmi COBOL di test con complessità crescente
- Implementazione dell'interfacciamento con *database* relazionali

Seconda fase - sviluppo del sistema di migrazione (4 settimane - 160 ore)

- Analisi dei *pattern* di traduzione COBOL-Java del codice prodotto in fase precedente
- Sviluppo di uno *script* o utilizzo di *tool* esistenti per automatizzare la traduzione del codice COBOL in Java equivalente

- Gestione della traduzione dei costrutti sintattici, logica di controllo e interazioni con il *database*
- Definizione della percentuale di automazione raggiungibile e la gestione di costrutti COBOL complessi o non direttamente traducibili

Terza fase - *testing* e validazione (1 settimana - 40 ore)

- *Test* funzionali sul codice Java generato
- Confronto comportamentale con le applicazioni COBOL originali

Quarta fase - documentazione e consegna (1 settimana - 40 ore)

- Documentazione completa del sistema sviluppato
- Preparazione del materiale di consegna
- Presentazione finale dei risultati

La rappresentazione temporale dettagliata della pianificazione è visualizzata in Figura 8, che mostra la distribuzione delle attività lungo l'arco temporale dello stage.



Figura 8: Diagramma di Gantt della pianificazione del progetto

Valore strategico per l'azienda

In base a quanto ho potuto osservare e comprendere durante il periodo di *stage*, la strategia di gestione del progetto di migrazione COBOL-Java dell'azienda ospitante persegue i seguenti obiettivi:

- **Innovazione tecnologica:** l'interesse dell'azienda non era limitato allo sviluppo di una soluzione tecnica specifica, ma si estendeva all'osservazione dell'approccio metodologico e del metodo di studio che una risorsa *junior* con formazione universitaria avrebbe applicato a un problema complesso di modernizzazione IT.

- **Creazione di competenze interne:** Il progetto permetteva di sviluppare *know-how* interno su una problematica di crescente rilevanza, preparando l'azienda a potenziali progetti futuri.
- **Esplorazione di tecnologie emergenti:** Il progetto era stato concepito per esplorare la possibile applicazione dell'intelligenza artificiale generativa a problemi di modernizzazione del *software*. Questo ambito, all'intersezione tra AI e *software engineering*, può rappresentare una frontiera tecnologica di forte attualità e di interesse per un'azienda che opera già attivamente nel campo dell'AI e dei *Large Language Models*.
- **Sviluppo di asset riutilizzabili:** Sebbene il progetto fosse autoconclusivo, permetteva di ottenere risultati tangibili nel breve termine dello *stage*, ma con il potenziale di evolversi in soluzioni più ampie e commercializzabili.

Aspettative personali

La scelta di intraprendere questo *stage* presso Miriade è stata guidata da una combinazione di motivazioni tecniche e personali che si allineavano con il mio percorso formativo universitario. Tra le diverse opportunità di *stage* che avevo valutato, questo progetto si distingueva per due elementi fondamentali:

- **Libertà tecnologica:** La libertà concessami nell'esplorazione delle tecnologie da utilizzare rappresentava un'opportunità unica di sperimentazione e apprendimento.
- **Interesse per COBOL:** Il mio forte interesse nel scoprire di più sul linguaggio COBOL, un affascinante paradosso tecnologico che, nonostante la sua longeva età, continua a essere cruciale nello scenario bancario e assicurativo internazionale.

Il mio percorso di *stage* mirava principalmente all'acquisizione di competenze pratiche nel campo della modernizzazione di sistemi *legacy* e gestione progetti:

Obiettivi tecnici

- Comprendere la struttura e la logica dei programmi COBOL attraverso lo sviluppo di applicazioni di test
- Esplorare approcci concreti alla migrazione del codice, sia deterministici che basati su AI
- Produrre un prototipo funzionante di sistema di conversione, anche se limitato

Competenze da sviluppare

- Familiarità di base con il linguaggio COBOL e le sue peculiarità sintattiche
- Comprensione pratica delle sfide nella traduzione tra paradigmi di programmazione diversi
- Esperienza nell'utilizzo di tecnologie emergenti come l'AI generativa applicata al codice

Crescita professionale attesa

- Sviluppare autonomia nella gestione di un progetto aziendale, dalla pianificazione all'implementazione
 - Acquisire capacità di *problem solving* in contesti reali, con vincoli temporali e tecnologici definiti
 - Migliorare le competenze comunicative attraverso l'interazione con il *team* e la presentazione dei progressi
 - Apprendere metodologie di lavoro *Agile* applicate a progetti di ricerca e sviluppo.
- Figura 9 rappresenta visivamente l'approccio metodologico Agile che ho appreso e applicato durante lo stage, evidenziando il ciclo iterativo di pianificazione, sviluppo, testing e revisione che ha caratterizzato il mio percorso formativo.
- Sviluppare pensiero critico nella valutazione di soluzioni tecnologiche alternative



Fonte: <https://indevlab.com>

Figura 9: Rappresentazione della metodologia Agile applicata al progetto

Sviluppo del progetto: dall'analizzatore sintattico tradizionale all'AI

Il percorso di sviluppo del progetto di migrazione COBOL-Java ha rappresentato un viaggio tecnologico affascinante, caratterizzato da sfide inaspettate e svolte strategiche che hanno trasformato radicalmente l'approccio iniziale. Durante le otto settimane di tirocinio, ho potuto sperimentare in prima persona come l'innovazione tecnologica possa emergere dall'adattamento e dalla capacità di ridefinire gli obiettivi in base alle scoperte progressive. Questo capitolo ripercorre cronologicamente le fasi del progetto, dall'immersione iniziale nel mondo COBOL fino all'implementazione di una soluzione basata sull'intelligenza artificiale generativa, evidenziando come la flessibilità metodologica e l'apertura all'innovazione siano state determinanti per il successo dell'iniziativa.

Configurazione iniziale e metodologia di lavoro

L'avvio del progetto è stato caratterizzato da un'attenta fase di configurazione dell'ambiente di lavoro e dall'adozione rigorosa della metodologia *Agile*, elemento distintivo della cultura aziendale di Miriade. La prima settimana è stata dedicata alla familiarizzazione con gli strumenti e i processi aziendali, fondamentali per garantire un'integrazione efficace nel gruppo di lavoro e una gestione strutturata del progetto.

L'implementazione della metodologia Agile si è concretizzata attraverso iterazioni settimanali, con obiettivi chiari e misurabili definiti ogni lunedì mattina durante le sessioni di pianificazione. Questi momenti di allineamento collettivo permettevano di definire le priorità della settimana, stimare l'impegno necessario per ciascuna attività e identificare potenziali rischi o dipendenze. La flessibilità intrinseca dell'approccio Agile si è rivelata particolarmente preziosa quando, a metà progetto, è stato necessario cambiare direzione verso una soluzione completamente diversa da quella inizialmente prevista.

Le riunioni quotidiane di allineamento, condotte puntualmente alle 9:30, hanno rappresentato momenti cruciali per l'allineamento con il tutor aziendale e per la condivisione dei progressi e delle eventuali difficoltà incontrate. Durante questi brevi incontri, strutturati secondo il formato classico delle tre domande (cosa ho fatto ieri, cosa farò oggi, quali ostacoli sto incontrando), ho potuto beneficiare non solo del supporto tecnico del tutor, ma anche delle prospettive e suggerimenti di altri membri del gruppo che occasionalmente partecipavano. Questa routine quotidiana ha favorito un approccio iterativo e incrementale, permettendomi

di adattare rapidamente la direzione del progetto in base ai riscontri ricevuti e alle scoperte tecniche progressive.

L'ambiente tecnologico predisposto per lo sviluppo includeva una postazione di lavoro con sistema operativo Windows 11, configurata con l'*Atlassian Suite* completa per la gestione del progetto. La configurazione iniziale ha richiesto particolare attenzione per garantire l'accesso a tutti gli strumenti necessari e l'integrazione con i sistemi aziendali esistenti.

Jira è stato utilizzato per il tracciamento dettagliato delle attività, con la creazione di segnalazioni specifiche per ogni fase del progetto. La struttura delle segnalazioni seguiva il formato standard aziendale, includendo descrizione dettagliata, criteri di accettazione, stima temporale e priorità. Ho imparato l'importanza di mantenere le attività costantemente aggiornate, non solo per tracciare il progresso, ma anche per documentare le decisioni prese e le motivazioni dietro cambiamenti di direzione. La pratica di collegare le conferme delle modifiche nel codice alle segnalazioni in *Jira* ha creato una tracciabilità completa dal requisito all'implementazione.

Confluence ha svolto un ruolo fondamentale come archivio della documentazione progressiva. Ho creato uno spazio dedicato al progetto di migrazione COBOL-Java, organizzato in sezioni logiche che riflettevano le diverse fasi del progetto. La documentazione includeva:

- Analisi tecniche dettagliate delle sfide incontrate
- Decisioni architetturali con pro e contro di ciascuna opzione valutata
- Documentazione del processo di apprendimento COBOL
- Risultati dei collaudi comparativi tra diversi approcci
- Registri settimanali dei progressi e delle riflessioni

La pratica di documentare non solo il «cosa» ma anche il «perché» delle decisioni si è rivelata preziosa, specialmente quando è stato necessario rivedere approcci precedenti o spiegare il razionale dietro il cambio di strategia.

Il versionamento del codice è stato gestito attraverso *Git*, con un archivio dedicato su *BitBucket*. La strategia di ramificazione adottata seguiva il Git Flow aziendale, con alcune personalizzazioni per adattarsi alla natura sperimentale del progetto. Ho mantenuto:

- Un ramo `main` per il codice stabile e testato
- Rami `feature/` per ogni nuova funzionalità o esperimento
- Rami `experiment/` per prove e prototipi che potrebbero non confluire nel principale
- Etichette per marcare tappe significative del progetto

Ho adottato una politica di conferme frequenti e descrittive, seguendo le convenzioni aziendali per i messaggi di conferma. I prefissi standardizzati (`feat:`, `fix:`, `docs:`,

`refactor:`, `test:`) non solo rendevano la cronologia più leggibile, ma facilitavano anche la generazione automatica di registri delle modifiche per le revisioni settimanali. La pratica di scrivere messaggi di conferma dettagliati, spiegando non solo cosa cambiava ma perché, si è rivelata un investimento prezioso quando è stato necessario ripercorrere l'evoluzione di determinate scelte implementative.

La documentazione progressiva del progetto è stata mantenuta con particolare attenzione attraverso diversi canali:

- Un diario tecnico settimanale in Confluence che registrava decisioni prese, problemi incontrati e soluzioni adottate
- File README nell'archivio che documentavano configurazione, utilizzo e architettura di ciascun componente
- Commenti dettagliati nel codice per spiegare logiche complesse o non ovvie
- Diagrammi architetturali aggiornati per visualizzare l'evoluzione del sistema

Questo approccio multi-livello alla documentazione si è rivelato prezioso nelle fasi successive del progetto, quando è stato necessario ripercorrere le motivazioni di determinate scelte tecniche o spiegare ad altri membri del gruppo l'evoluzione del progetto.

Primo periodo: immersione nel mondo COBOL

Le prime due settimane del progetto sono state dedicate a un'immersione completa nel linguaggio COBOL, un'esperienza che ha richiesto un notevole sforzo di adattamento mentale. Il passaggio dai paradigmi di programmazione moderni a cui ero abituato - con la loro enfasi su astrazione, modularità e riusabilità - a un approccio procedurale strutturato degli anni "60 ha rappresentato una sfida cognitiva significativa. Questa fase di apprendimento intensivo si è rivelata fondamentale non solo per acquisire competenze tecniche, ma anche per comprendere la filosofia e il contesto storico che hanno plasmato COBOL e, di conseguenza, i sistemi legacy che ancora oggi sostengono infrastrutture critiche.

Studio del linguaggio e creazione progetti di prova

L'apprendimento di COBOL è iniziato con lo studio sistematico della sintassi e semantica del linguaggio, un processo che ha richiesto un approccio metodico e paziente. Le risorse utilizzate spaziavano dai manuali tecnici IBM degli anni "80, sorprendentemente ancora rilevanti, a tutorial online più moderni che tentavano di rendere COBOL accessibile a programmatori contemporanei. Particolarmente preziosa si è rivelata la collaborazione con la sig.ra Luisa Biagi, analista COBOL con oltre vent'anni di esperienza nel settore bancario, che ha potuto

fornire non solo conoscenze tecniche ma anche contesto pratico su come COBOL viene effettivamente utilizzato in ambienti di produzione.

La struttura rigida del linguaggio, con le sue quattro divisioni obbligatorie (*Identification, Environment, Data e Procedure*), rappresentava un paradigma completamente diverso rispetto ai linguaggi orientati agli oggetti a cui ero abituato. Questa rigidità strutturale, inizialmente percepita come limitante, si è rivelata essere una caratteristica progettuale deliberata, pensata per imporre ordine e standardizzazione in progetti di grandi dimensioni sviluppati da gruppi numerosi.

La IDENTIFICATION DIVISION, apparentemente semplice, nascondeva importanti lezioni sulla filosofia COBOL. L'enfasi sulla documentazione incorporata nel codice, con campi dedicati per autore, data di installazione, data di compilazione e osservazioni, rifletteva un'epoca in cui il codice sorgente era spesso l'unica documentazione disponibile. Questa pratica, sebbene possa sembrare antiquata, offriva spunti interessanti sulla tracciabilità e la gestione del ciclo di vita del software.

La ENVIRONMENT DIVISION mi ha introdotto a un mondo dove l'hardware e il sistema operativo erano considerazioni esplicite nel codice sorgente. La necessità di specificare SOURCE-COMPUTER e OBJECT-COMPUTER, concetti alieni nella programmazione moderna dove la portabilità è data per scontata, evidenziava le sfide dell'era dei mainframe. La sezione FILE-CONTROL, con la sua gestione esplicita dell'associazione tra file logici e fisici, richiedeva un cambio di mentalità significativo rispetto all'astrazione automatica fornita dai moderni sistemi operativi e strutture software.

Per consolidare l'apprendimento teorico, ho intrapreso lo sviluppo sistematico di una serie di applicazioni di prova con complessità crescente. Questo approccio pratico si è rivelato essenziale per interiorizzare non solo la sintassi, ma anche gli idiomi e le migliori pratiche del linguaggio.

Il primo programma sviluppato, il classico «Hello World», ha immediatamente evidenziato la verbosità caratteristica di COBOL. Quella che in linguaggi moderni sarebbe una singola linea di codice richiedeva in COBOL una struttura completa con tutte le divisioni dichiarate, anche se molte rimanevano vuote. Questa esperienza iniziale ha sottolineato come COBOL fosse progettato per applicazioni complesse, dove il sovraccarico della struttura base diventava trascurabile rispetto alla dimensione totale del programma.

Progressivamente, ho sviluppato applicazioni più articolate che mi hanno permesso di esplorare diversi aspetti del linguaggio e le sue capacità. Il secondo progetto, un calcolatore

di interessi bancari, ha introdotto sfide legate alla gestione dei calcoli decimali con precisione finanziaria. COBOL eccelle in questo dominio grazie al suo supporto nativo per l'aritmetica decimale a precisione fissa, una caratteristica che molti linguaggi moderni gestiscono solo attraverso librerie specializzate. La clausola PICTURE per la definizione dei formati numerici, inizialmente criptica con la sua notazione basata su 9, V, S e altri simboli, si è rivelata straordinariamente potente per garantire la precisione richiesta nelle applicazioni finanziarie.

Il terzo progetto, un sistema di gestione inventario, ha rappresentato un salto significativo in complessità. Questo sistema implementava operazioni complete di creazione, lettura, aggiornamento e cancellazione (CRUD) su record strutturati. La gestione dei file in COBOL, con la distinzione tra file sequenziali, indicizzati e relativi, ha richiesto la comprensione di concetti di accesso ai dati molto più vicini all'hardware rispetto alle astrazioni moderne. La definizione di record con campi a lunghezza fissa, la gestione esplicita degli indici, e la necessità di considerare l'organizzazione fisica dei dati sul disco erano tutti aspetti che evidenziavano come COBOL fosse nato in un'era di risorse limitate dove l'efficienza era cruciale.

L'implementazione della logica di validazione dei dati ha rivelato un altro aspetto interessante di COBOL: la potenza delle sue strutture condizionali. L'istruzione EVALUATE, equivalente evoluto dello switch-case, permetteva di esprimere logiche complesse in modo sorprendentemente leggibile. Le condizioni 88-level, che permettono di associare nomi significativi a valori o intervalli di valori, dimostravano come COBOL, nonostante la sua età, incorporasse concetti di programmazione autodocumentante.

Il quarto progetto, un'applicazione di elaborazione batch per la generazione di rapporti, mi ha introdotto a uno dei domini principali di COBOL: l'elaborazione di grandi volumi di dati. La generazione di rapporti formattati, con totali di gruppo, subtotali e formattazione complessa, è supportata nativamente dal linguaggio attraverso caratteristiche come la REPORT SECTION. Questa esperienza ha evidenziato come COBOL fosse ottimizzato per scenari d'uso specifici del mondo aziendale, incorporando nel linguaggio stesso schemi comuni di elaborazione dati.

L'interfacciamento con database relazionali ha rappresentato una sfida particolare e illuminante. Ho lavorato sia con *PostgreSQL* che con *DB2*, scoprendo le peculiarità dell'SQL (SQL) incorporato in COBOL. L'approccio di COBOL all'integrazione SQL differisce radicalmente dai moderni ORM o anche dalle API JDBC. Il preprocessore COBOL-SQL analizza

il codice sorgente, estrae le istruzioni SQL delimitate da EXEC SQL...END-EXEC, e genera il codice COBOL appropriato per l'interazione con il database.

La gestione delle variabili ospite, variabili COBOL che fungono da ponte tra il programma e il database, richiede particolare attenzione. Ogni variabile utilizzata in una query SQL deve essere dichiarata nella WORKING-STORAGE con tipo e dimensione compatibili con la colonna del database corrispondente. La gestione dei valori nulli attraverso variabili indicatore separate aggiunge un ulteriore livello di complessità rispetto alle soluzioni moderne dove null è un valore gestito nativamente.

La gestione degli errori nelle operazioni database attraverso SQLCODE e SQLSTATE ha richiesto l'implementazione di routine di controllo sistematiche. A differenza delle moderne eccezioni, COBOL richiede il controllo esplicito del codice di ritorno dopo ogni operazione SQL. Ho sviluppato schemi standard per questa gestione, creando paragrafi riutilizzabili per il controllo degli errori comuni. Questa esperienza ha sottolineato l'importanza della disciplina nella programmazione COBOL: senza le reti di sicurezza automatiche dei linguaggi moderni, la robustezza deve essere costruita esplicitamente nel codice.

Un aspetto particolarmente interessante emerso durante questa fase è stata la gestione delle transazioni. COBOL, nato in un'era di elaborazione batch, ha un approccio alle transazioni che riflette questa eredità. Il concetto di unità di lavoro, con COMMIT e ROLLBACK espliciti, richiede una pianificazione attenta del flusso di elaborazione. Ho dovuto imparare a strutturare i programmi in modo che i confini transazionali fossero chiari e che il recupero da errori fosse sempre possibile.

Mappatura degli schemi ricorrenti e analisi di traducibilità

Durante lo sviluppo delle applicazioni di prova, ho intrapreso un'attività sistematica di catalogazione degli schemi ricorrenti nel codice COBOL, creando una mappatura dettagliata delle possibili corrispondenze con costrutti equivalenti nei linguaggi moderni. Questa analisi, condotta con rigore metodologico, ha rivelato un panorama complesso di similitudini incoraggianti e differenze strutturali significative che avrebbero influenzato profondamente l'approccio alla migrazione.

La metodologia adottata per questa mappatura prevedeva l'identificazione di schemi a diversi livelli di astrazione. Al livello più basso, ho catalogato le corrispondenze dirette tra costrutti sintattici. Al livello intermedio, ho analizzato modelli di progettazione e idiomi ricorrenti. Al livello più alto, ho esaminato le architetture applicative tipiche e come queste potrebbero essere trasposte in paradigmi moderni.

Gli schemi identificati spaziavano dalle strutture di controllo fondamentali alle operazioni più complesse di gestione dati. Per le strutture di controllo, ho documentato come le istruzioni IF-THEN-ELSE di COBOL mappassero naturalmente sui costrutti condizionali moderni, ma con alcune sottigliezze. La possibilità in COBOL di avere IF statements annidati profondamente, spesso senza ELSE espliciti, creava ambiguità che richiedevano analisi attenta del flusso di controllo per una conversione corretta.

Le strutture iterative presentavano maggiore complessità. PERFORM, il costrutto principale per l'iterazione in COBOL, ha numerose varianti che non sempre hanno corrispondenze dirette nei linguaggi moderni. PERFORM TIMES mappa naturalmente su un ciclo for con contatore, ma PERFORM UNTIL richiede attenzione alla semantica di valutazione della condizione (pre o post test). PERFORM VARYING, specialmente con clausole variabili multiple, può risultare in strutture di ciclo annidate complesse che richiedono ristrutturazione significativa per risultare idiomatiche nel linguaggio di destinazione.

La gestione dei dati in COBOL presenta paradigmi unici che ho documentato estensivamente. Le strutture dati gerarchiche, definite attraverso numeri di livello, non hanno equivalenti diretti nei linguaggi moderni. Ho sviluppato strategie per mappare queste strutture su classi e oggetti, preservando le relazioni gerarchiche attraverso composizione ed ereditarietà dove appropriato. La sfida principale risiedeva nel preservare la semantica di accesso ai dati: in COBOL, riferirsi a un gruppo implicitamente include tutti i suoi elementi subordinati, un comportamento che deve essere emulato esplicitamente nei linguaggi orientati agli oggetti.

Le clausole PICTURE rappresentavano una sfida particolare. Questi potenti descrittori di formato non solo definiscono il tipo e la dimensione dei dati, ma anche la loro rappresentazione. Ho catalogato schemi per convertire le clausole picture più comuni in combinazioni di tipi di dato e formattatori, ma alcune clausole complesse (come quelle con editing di inserimento o simboli flottanti) richiedevano logica personalizzata per essere replicate fedelmente.

L'analisi del grado di migrabilità ha prodotto una tassonomia dettagliata che ho organizzato in tre categorie principali, ciascuna con le proprie sfide e strategie di conversione.

La prima categoria, i costrutti direttamente traducibili, comprendeva circa il 60% del codice COBOL tipico. Questi includevano:

- Strutture di controllo base (IF, EVALUATE) che mappano su costrutti equivalenti
- Dichiarazioni di variabili semplici che possono essere convertite in tipi primitivi o semplici oggetti
- Operazioni aritmetiche standard, con attenzione alla precisione decimale
- Operazioni di I/O base che mappano su flussi e operazioni file moderne

Per questa categoria, ho sviluppato regole di mappatura relativamente dirette, anche se con attenzione ai dettagli semantici. Ad esempio, la divisione in COBOL può produrre sia quoziente che resto in una singola operazione, richiedendo decomposizione in operazioni separate nel linguaggio di destinazione.

La seconda categoria, costrutti traducibili con adattamento, rappresentava circa il 30% del codice e richiedeva trasformazioni più sofisticate:

- Gestione file con organizzazioni sequenziali, indicizzate e relative che richiedono mappatura su database o strutture dati moderne
- Clausole PICTURE complesse che necessitano di formattatori personalizzati
- Strutture dati gerarchiche che devono essere trasformate in modelli a oggetti
- Istruzioni PERFORM complesse che richiedono ristrutturazione del flusso di controllo

Per questi costrutti, ho sviluppato schemi di trasformazione che preservavano la semantica mentre modernizzavano l'implementazione. Ad esempio, un file indicizzato COBOL potrebbe essere mappato su una tabella database con indici appropriati, o su una struttura dati in memoria con caratteristiche di accesso equivalenti.

La terza categoria, i costrutti problematici, fortunatamente limitata al 10% circa del codice ma spesso critica, includeva:

- Istruzioni GOTO che violano i principi di programmazione strutturata
- Verbo ALTER che modifica dinamicamente il flusso del programma
- REDEFINES che crea viste multiple della stessa area di memoria
- Alcune forme esoteriche di PERFORM che creano flussi di controllo complessi

Per questi costrutti, la strategia non poteva essere una semplice traduzione ma richiedeva un vero e proprio reingegnerizzazione. Ho documentato approcci per eliminare i GOTO attraverso ristrutturazione in strutture di controllo appropriate, strategie per gestire REDEFINES attraverso tipi unione o getter/setter multipli, e tecniche per semplificare PERFORM complessi in strutture più manutenibili.

Un aspetto cruciale emerso da questa analisi è stata la realizzazione che la traducibilità non è solo una questione tecnica ma anche di preservazione dell'intento. Molti schemi COBOL, sebbene tecnicamente traducibili, perderebbero significato se convertiti meccanicamente. Ad esempio, l'uso estensivo di condizioni 88-level in COBOL spesso codifica regole aziendali importanti. Una traduzione meccanica in semplici costanti perderebbe la ricchezza semantica; invece, ho proposto l'uso di enumerazioni o classi di validazione che preservano e documentano queste regole.

Le strategie individuate per gestire le incompatibilità strutturali si basavano su principi di trasformazione che ho progressivamente raffinato:

Il principio di preservazione semantica stabiliva che ogni trasformazione doveva mantenere non solo il comportamento osservabile ma anche l'intento del codice originale. Questo significava che commenti, nomi significativi, e struttura logica dovevano essere preservati o migliorati, mai persi.

Il principio di modernizzazione incrementale suggeriva che non tutto il codice doveva essere modernizzato allo stesso livello. Parti critiche potevano beneficiare di ristrutturazione profonda, mentre sezioni stabili e ben funzionanti potevano subire trasformazioni più conservative.

Il principio di tracciabilità richiedeva che ogni trasformazione fosse documentata e reversibile concettualmente. Questo non significava poter tornare automaticamente al COBOL, ma che la logica di trasformazione doveva essere chiara e verificabile.

Valutazione delle soluzioni esistenti

Prima di procedere con lo sviluppo di una soluzione proprietaria, ho condotto un'analisi approfondita e sistematica delle soluzioni esistenti sul mercato, sia *open-source* che aziendali. Questa fase di ricerca e valutazione si è rivelata fondamentale non solo per comprendere lo stato dell'arte, ma anche per identificare opportunità di innovazione e differenziazione.

La mia analisi ha seguito un approccio strutturato, valutando ciascuna soluzione secondo criteri multipli: completezza della copertura COBOL, qualità del codice generato, facilità d'uso, costo totale di proprietà, supporto e manutenzione, estensibilità e personalizzazione. Ho creato una matrice di valutazione dettagliata che permettesse confronti oggettivi tra le diverse opzioni.

L'approccio tradizionale basato su architettura a catena di elaborazione con analizzatori sintattici deterministici rappresentava la soluzione più diffusa nel panorama open-source. Questi sistemi seguono il classico modello di compilatore con fasi distinte: analisi lessicale, analisi sintattica, generazione di albero sintattico astratto (AST), trasformazioni sull'AST, e generazione del codice di destinazione.

Ho esaminato in dettaglio il *ProLeap COBOL parser* disponibile su GitHub, uno dei progetti open-source più maturi in questo spazio. L'analizzatore, basato su ANTLR4, dimostrava capacità impressionanti nell'analisi di COBOL-85 con estensioni per alcuni dialetti specifici dei fornitori. L'architettura modulare permetteva l'estensione per nuovi costrutti, e la generazione dell'AST era completa e ben strutturata.

Tuttavia, l'analisi approfondita ha rivelato limitazioni significative. L'analizzatore eccelleva nell'analisi sintattica ma mancava di comprensione semantica profonda. La trasformazione da AST COBOL a codice di destinazione rimaneva largamente un esercizio manuale o semi-automatico. Il codice generato, quando automatizzato, tendeva a essere una traduzione letterale che preservava le idiosincrasie COBOL invece di produrre codice idiomatrico nel linguaggio di destinazione.

Ho anche valutato altri strumenti open-source, ciascuno con i propri punti di forza e debolezza. Alcuni si concentravano su sottoinsiemi specifici di COBOL (come COBOL per specifiche piattaforme), altri tentavano trasformazioni più ambiziose ma con risultati inconsistenti. Uno schema comune era la difficoltà nel gestire costrutti COBOL che non hanno equivalenti diretti nei linguaggi moderni, risultando in codice generato di difficile manutenzione.

Sul fronte aziendale, il panorama era dominato da soluzioni commerciali sofisticate con prezzi corrispondentemente elevati. Ho avuto l'opportunità di valutare diverse soluzioni leader di mercato attraverso dimostrazioni, documentazione, e in alcuni casi prove limitate.

IBM WatsonX Code Assistant for Z rappresentava lo stato dell'arte nell'applicazione dell'intelligenza artificiale alla modernizzazione legacy. La soluzione IBM non si limitava alla traduzione sintattica ma tentava di comprendere il contesto aziendale del codice. Durante la dimostrazione, ho osservato come il sistema potesse identificare schemi di logica aziendale, suggerire ristrutturazioni appropriate, e generare codice che seguiva le migliori pratiche moderne.

Le capacità di WatsonX erano impressionanti: riconoscimento di schemi di dominio (come elaborazione batch tipica del settore bancario), suggerimenti di modernizzazione architetturale (conversione a microservizi dove appropriato), generazione di test automatici basati sulla comprensione del comportamento atteso, e documentazione automatica che catturava l'intento aziendale oltre che tecnico.

Tuttavia, la soluzione presentava barriere significative per un progetto di ricerca: costo proibitivo per licenze anche di sviluppo, natura proprietaria che impediva personalizzazioni profonde, requisiti di infrastruttura aziendale, e vincolo con l'ecosistema IBM.

Ho valutato anche altre soluzioni aziendali di fornitori come Micro Focus, Modern Systems, e TSRI. Ciascuna aveva approcci unici: alcune si concentravano su migrazioni «solleva e sposta» che preservavano la struttura COBOL in runtime Java, altre tentavano trasformazioni più radicali. Un tema comune era il compromesso tra automazione e qualità:

maggiore era l'automazione, più il codice risultante tendeva a essere non-idiomatrico e difficile da mantenere.

Un'osservazione cruciale emersa da questa analisi è stata l'identificazione di un vuoto significativo nel mercato. Le soluzioni open-source, pur essendo accessibili, mancavano di sofisticazione e producevano risultati che richiedevano estese modifiche manuali. Le soluzioni aziendali, pur essendo potenti, erano inaccessibili per la maggior parte delle organizzazioni a causa di costi e complessità.

Questo vuoto suggeriva un'opportunità per una soluzione che combinasse l'accessibilità dell'open-source con capacità più sofisticate di comprensione e trasformazione. La recente democratizzazione dell'AI generativa attraverso API accessibili apriva possibilità precedentemente riservate solo a fornitori con risorse massive.

L'analisi delle soluzioni esistenti ha anche rivelato schemi comuni di fallimento e fattori di successo:

- Le migrazioni di successo preservavano la logica aziendale mentre modernizzavano l'implementazione
- L'importanza della documentazione e della tracciabilità nel processo di migrazione
- La necessità di un approccio iterativo con validazione continua
- Il valore di preservare la conoscenza di dominio incorporata nel codice legacy

Queste osservazioni hanno informato significativamente l'approccio che avrei adottato nelle fasi successive del progetto, suggerendo che una soluzione efficace dovrebbe combinare automazione intelligente con comprensione semantica profonda, preservazione della logica aziendale con modernizzazione dell'implementazione, e accessibilità con potenza.

Secondo periodo: sviluppo dell'analizzatore sintattico tradizionale

Forte delle conoscenze acquisite sul linguaggio COBOL e dell'analisi approfondita delle soluzioni esistenti, ho intrapreso lo sviluppo di un analizzatore sintattico proprietario basato su un approccio deterministico tradizionale. Questa decisione, presa in consultazione con il tutor aziendale, era motivata dal desiderio di comprendere profondamente le sfide tecniche della migrazione e di esplorare fino a che punto un approccio «classico» potesse essere spinto con tecniche moderne di sviluppo software.

Implementazione dell'analizzatore Java

Lo sviluppo dell'analizzatore è iniziato con una fase di progettazione architetturale dettagliata. L'obiettivo era creare un sistema modulare ed estensibile che potesse crescere incrementalmente man mano che nuovi costrutti COBOL venivano supportati. L'architettura progettata prevedeva una chiara separazione delle responsabilità tra componenti, facilitando sia lo sviluppo che il collaudo.

Il progetto dell'analizzatore seguiva un'architettura a catena di elaborazione con componenti debolmente accoppiati. Il Lexer si occupava della tokenizzazione del codice COBOL, gestendo le peculiarità del linguaggio come la sensibilità alla colonna, i commenti, e le linee di continuazione. L'analizzatore vero e proprio costruiva l'albero sintattico astratto basandosi sulla grammatica COBOL. L'analizzatore semantico arricchiva l'AST con informazioni di tipo e riferimenti. Il generatore di codice trasformava l'AST annotato in codice Java. Ogni componente comunicava attraverso interfacce ben definite, permettendo evoluzione indipendente.

La IDENTIFICATION DIVISION è stata il punto di partenza naturale, essendo la sezione più semplice e predicibile del programma COBOL. Questa divisione contiene principalmente metadati che non hanno impatto diretto sulla logica del programma ma forniscono contesto importante. Ho implementato un analizzatore basato su corrispondenza di schemi che estraeva informazioni come PROGRAM-ID, AUTHOR, INSTALLATION, DATE-WRITTEN, e REMARKS.

La strategia di conversione per questa divisione prevedeva la trasformazione di questi metadati in annotazioni Java e commenti strutturati. Il PROGRAM-ID diventava il nome della classe Java principale, seguendo convenzioni di denominazione Java (conversione da KEBAB-CASE a CamelCase). Le altre informazioni venivano preservate in un blocco di commenti JavaDoc in testa alla classe, mantenendo la tracciabilità con il programma originale.

Un aspetto interessante emerso durante l'implementazione è stata la gestione delle variazioni nella formattazione. COBOL, essendo un linguaggio basato su colonne, permette variazioni significative in come gli stessi costrutti possono essere formattati. L'analizzatore doveva essere abbastanza robusto da gestire queste variazioni senza perdere informazioni.

La ENVIRONMENT DIVISION ha presentato sfide più significative. Questa divisione specifica l'ambiente di esecuzione del programma COBOL, includendo informazioni su

hardware, sistema operativo, e mappatura dei file. In un contesto moderno, molte di queste informazioni sono obsolete o gestite diversamente.

Ho sviluppato una strategia di conversione che mappava le informazioni rilevanti su costrutti moderni. La CONFIGURATION SECTION, con SOURCE-COMPUTER e OBJECT-COMPUTER, veniva largamente ignorata o convertita in commenti documentativi. La più critica INPUT-OUTPUT SECTION, che definisce i file utilizzati dal programma, richiedeva trasformazione più sofisticata.

Il paragrafo FILE-CONTROL presentava complessità particolare. Ogni dichiarazione di file in COBOL include non solo il nome logico del file ma anche dettagli sulla sua organizzazione (sequenziale, indicizzata, relativa), modalità di accesso (sequenziale, casuale, dinamica), e gestione degli errori. Ho mappato queste dichiarazioni su configurazioni per l'accesso ai file in Java, creando livelli di astrazione che preservavano la semantica COBOL mentre utilizzavano API Java moderne.

La gestione dei blocchi dei file e delle modalità di condivisione ha richiesto particolare attenzione. COBOL permette controllo fine sulla condivisione dei file tra programmi concorrenti, un aspetto critico in ambienti mainframe multi-utente. Replicare questa semantica in Java ha richiesto l'uso di API di blocco file e sincronizzazione accurata.

La DATA DIVISION ha rappresentato il vero banco di prova per l'approccio basato su analizzatore sintattico. Questa divisione definisce tutte le strutture dati utilizzate dal programma, usando un sistema gerarchico basato su numeri di livello che non ha equivalenti diretti nei linguaggi moderni.

L'analisi della DATA DIVISION richiedeva la comprensione di concetti unici a COBOL. I numeri di livello (01-49, 66, 77, 88) definiscono una gerarchia di dati con semantiche specifiche. Le clausole PICTURE specificano il formato esatto dei dati con una notazione compatta ma espressiva. Le clausole REDEFINES permettono viste multiple della stessa area di memoria. Le clausole OCCURS definiscono array con possibili dimensioni variabili.

Ho implementato un analizzatore ricorsivo che costruiva una rappresentazione interna della gerarchia dei dati. Ogni elemento dati veniva rappresentato come un nodo in un albero, con attributi che catturavano tutte le clausole associate. La sfida principale era mantenere le relazioni tra elementi dati, specialmente per caratteristiche come REDEFINES che creano aliasing complesso.

La conversione della DATA DIVISION in strutture Java appropriate si è rivelata sorprendentemente complessa. Un approccio ingenuo di mappare ogni elemento di gruppo su una

classe Java e ogni elemento elementare su un campo produceva codice verboso e non-idiomatrico. Ho sperimentato con diverse strategie:

- Appiattimento di gerarchie semplici in classi con campi diretti
- Uso di classi interne per preservare strutture gerarchiche complesse
- Generazione di schemi costruttore per la costruzione di oggetti complessi
- Implementazione di serializzazione personalizzata per preservare layout di memoria COBOL dove necessario

La gestione delle clausole PICTURE ha richiesto lo sviluppo di un sotto-sistema completo per analisi e interpretazione. Le clausole picture non solo definiscono tipo e dimensione dei dati, ma anche formattazione, gestione del segno, allineamento decimale, e altro. Ho creato un mini-analizzatore specializzato per clausole picture che le decomponeva in attributi gestibili e generava codice Java appropriato per validazione e formattazione.

Analisi critica e limiti dell'approccio

Dopo tre settimane di sviluppo intensivo, dedicando la maggior parte del tempo all'implementazione e affinamento dell'analizzatore, i limiti intrinseci dell'approccio tradizionale sono diventati dolorosamente evidenti. Quello che era iniziato come un esercizio ambizioso ma fattibile si era trasformato in un progetto di complessità esponenzialmente crescente.

La PROCEDURE DIVISION, che contiene la logica applicativa vera e propria, ha rappresentato il punto di rottura dell'approccio basato su analizzatore sintattico. Mentre le altre divisioni definiscono struttura e metadati, la PROCEDURE DIVISION è dove il «vero lavoro» avviene, e la sua complessità eclissava tutto quello che avevo affrontato fino a quel momento.

Il primo ostacolo maggiore è stato la gestione del flusso di controllo. COBOL permette strutture di controllo che violano i principi della programmazione strutturata moderna. Le istruzioni GOTO, sebbene deprecate, sono ancora comuni nel codice legacy. Ancora più problematiche sono le istruzioni PERFORM con le loro molteplici varianti.

PERFORM in COBOL non è semplicemente una chiamata a subroutine. Può essere:

- Un semplice PERFORM di un paragrafo (simile a una chiamata a metodo)
- PERFORM THROUGH che esegue un intervallo di paragrafi
- PERFORM TIMES per iterazioni con contatore
- PERFORM UNTIL per cicli condizionali
- PERFORM VARYING per cicli con variabili di controllo multiple

La complessità emergeva dalle interazioni tra queste forme. Un PERFORM può chiamare un paragrafo che contiene altri PERFORM, creando pile di esecuzione complesse. La semantica di PERFORM THROUGH, dove l'esecuzione continua attraverso paragrafi consecutivi fino a un punto finale, non ha equivalenti diretti in Java.

Ho tentato diverse strategie per gestire questa complessità. L'approccio iniziale di mappare ogni paragrafo su un metodo Java falliva con PERFORM THROUGH. Ho poi tentato di incorporare il codice dove possibile, ma questo produceva duplicazione massiccia. Un approccio più sofisticato usando macchine a stati per tracciare il flusso di esecuzione diventava rapidamente ingestibile.

La gestione delle sezioni e dei paragrafi presentava ulteriori sfide. In COBOL, il flusso di esecuzione «cade attraverso» da un paragrafo al successivo a meno che non sia esplicitamente deviato. Questa semantica di caduta è aliena a Java e richiede gestione accurata per essere emulata correttamente. La presenza di istruzioni EXIT che possono terminare cicli o paragrafi da punti arbitrari complicava ulteriormente l'analisi del flusso.

L'SQL incorporato ha rivelato un altro livello di complessità. Non si trattava semplicemente di estrarre istruzioni SQL e convertirle in chiamate JDBC. Il preprocessore COBOL-SQL fa assunzioni sul contesto che devono essere preservate:

- Le variabili ospite devono essere correttamente mappate e controllate nel tipo
- La gestione degli errori attraverso SQLCODE deve essere preservata
- I cursori devono mantenere stato tra chiamate
- Le transazioni devono rispettare i confini originali

Ho implementato un sotto-analizzatore per SQL incorporato che tentava di estrarre e trasformare queste istruzioni, ma le interazioni con il resto del codice COBOL rendevano difficile una trasformazione pulita. La necessità di generare codice standard significativo per ogni operazione SQL rendeva il codice risultante verboso e difficile da leggere.

Un problema ancora più fondamentale emergeva a livello di architettura. I programmi COBOL sono tipicamente monolitici, con migliaia o decine di migliaia di linee in un singolo file sorgente. Tentare di convertire questa struttura direttamente in Java produceva classi massive che violavano ogni principio di buona progettazione. Ma tentare di riorganizzare automaticamente in strutture più modulari richiedeva comprensione semantica profonda che un analizzatore sintattico non poteva fornire.

La qualità del codice prodotto era un'altra area di preoccupazione critica. Anche quando l'analizzatore riusciva a produrre codice Java sintatticamente corretto e funzionalmente

equivalente, il risultato era tutt'altro che idiomático. Il codice generato sembrava «COBOL scritto in Java» - preservava tutti gli idiomi e schemi COBOL invece di adottare convenzioni Java.

Esempi specifici di problemi di qualità includevano:

- Convenzioni di denominazione che riflettevano COBOL (MAIUSCOLE-CON-TRATTINI) invece di convenzioni Java
- Mancanza di incapsulamento appropriato - tutto pubblico e accessibile
- Assenza di sicurezza dei tipi significativa - uso estensivo di stringhe e tipi primitivi
- Strutture di controllo verbose che potevano essere semplificate
- Mancanza di gestione degli errori idiomática - controllo di codici di ritorno invece di eccezioni

A questo punto, dopo tre settimane di sviluppo intensivo, ho condotto una stima realistica del lavoro rimanente. L'analizzatore copriva forse il 25% dei costrutti COBOL necessari per gestire programmi del mondo reale. Estrapolare linearmente (ottimisticamente) suggeriva almeno altri 2-3 mesi di sviluppo solo per completare la copertura sintattica. Ma la complessità non era lineare - ogni nuovo costrutto interagiva con quelli esistenti in modi che richiedevano ristrutturazione del codice esistente.

Ancora più scoraggiante era la realizzazione che completare l'analizzatore era solo il primo passo. Il codice generato avrebbe richiesto estesa post-elaborazione per essere veramente utilizzabile. Stimavo che ogni 1000 linee di COBOL avrebbero richiesto giorni di ristrutturazione manuale del Java generato per renderlo manutenibile.

In una sessione di retrospettiva con il tutor aziendale, abbiamo analizzato criticamente i risultati ottenuti e le prospettive future. Era chiaro che l'approccio basato su analizzatore sintattico, sebbene tecnicamente fattibile dato tempo illimitato, non era pratico nei vincoli del progetto. Più importante, anche se completato, avrebbe prodotto risultati sub-ottimali che richiedevano comunque intervento umano significativo.

Questa realizzazione, sebbene inizialmente frustrante, si è rivelata un punto di svolta cruciale nel progetto. Aveva dimostrato definitivamente i limiti dell'approccio deterministico e aperto la strada all'esplorazione di alternative più innovative. Il lavoro svolto non era stato vano - aveva fornito approfondimenti profondi nelle sfide della migrazione COBOL e creato una solida base di conoscenza su cui costruire approcci più sofisticati.

Terzo periodo: svolta verso l'intelligenza artificiale

Il momento di svolta del progetto è avvenuto durante la quarta settimana, in quella che si è rivelata essere una delle decisioni più significative dell'intero percorso di tirocinio. Durante una sessione di revisione con il tutor aziendale, dove presentavo i progressi e le sfide incontrate con l'approccio basato su analizzatore sintattico, è emersa chiaramente la necessità di riconsiderare radicalmente la strategia.

La discussione è stata illuminante. Il tutor, con la sua esperienza in progetti di innovazione, ha sottolineato come il valore del progetto non risiedesse necessariamente nel completare un analizzatore tradizionale - qualcosa già tentato molte volte nel settore - ma nell'esplorare approcci genuinamente innovativi alla sfida della migrazione. La recente esplosione di capacità nell'AI generativa, particolarmente nel dominio della comprensione e generazione di codice, presentava un'opportunità unica di essere pionieri in un approccio completamente nuovo.

Valutazione delle API di AI generativa

Il passaggio da una catena di elaborazione deterministica a un sistema basato su AI ha richiesto innanzitutto una ricognizione approfondita del panorama delle AI generative disponibili. Il tempismo era fortunato: il 2024 aveva visto un'esplosione di modelli e servizi AI focalizzati specificamente su comprensione e generazione di codice.

Ho strutturato la valutazione secondo criteri multipli che riflettevano le necessità specifiche del progetto di migrazione:

- Comprensione di linguaggi legacy (specificamente COBOL)
- Capacità di generare codice idiomatrico in linguaggi di destinazione
- Comprensione del contesto e della logica aziendale
- Consistenza e affidabilità dei risultati
- Accessibilità API e documentazione
- Costi e limitazioni di utilizzo
- Possibilità di ottimizzazione o personalizzazione

La prima fase di valutazione ha coinvolto test pratici con frammenti di codice COBOL di varia complessità. Ho preparato una suite di benchmark che includeva:

- Semplici programmi sequenziali per testare la comprensione base
- Strutture dati complesse con REDEFINES e OCCURS
- Logica di controllo con istruzioni PERFORM annidate
- SQL incorporato e gestione transazioni
- Schemi comuni nel dominio bancario/assicurativo

Ogni sistema AI è stato valutato sulla sua capacità di comprendere questi frammenti e produrre spiegazioni accurate o suggerire conversioni appropriate. I risultati sono stati rivelatori e hanno sfidato molte delle mie assunzioni iniziali.

OpenAI GPT-4 ha dimostrato capacità impressionanti nella comprensione del codice COBOL. Non solo riconosceva la sintassi, ma dimostrava comprensione del contesto e dello scopo del codice. Quando presentato con un programma di calcolo interessi, GPT-4 non solo spiegava cosa faceva ogni sezione, ma identificava che si trattava di un calcolo finanziario e suggeriva considerazioni sulla precisione decimale nella conversione. La generazione di codice Java era generalmente idiomantica, seguendo convenzioni moderne e migliori pratiche.

Anthropic Claude si è distinto per la sua capacità di ragionamento e spiegazione. Quando presentato con codice COBOL complesso, Claude non solo lo traduceva ma forniva rationale dettagliato per le scelte di progettazione nella conversione. Particolarmente impressionante era la capacità di identificare potenziali problemi o ambiguità nel codice originale e suggerire chiarimenti. La generazione di documentazione era eccezionale, producendo commenti e JavaDoc che catturavano non solo il «cosa» ma il «perché» del codice.

Gemini di Google mostrava forza particolare nell'integrazione con ecosistemi esistenti. Le conversioni suggerite spesso includevano riferimenti a librerie e strutture appropriate, dimostrando conoscenza non solo dei linguaggi ma degli ecosistemi circostanti. La capacità di suggerire ristrutturazioni architetturali, come la decomposizione di monoliti in servizi, era notevole.

Modelli più specializzati come Codex e suoi derivati mostravano competenze tecniche solide ma mancavano della comprensione contestuale più ampia dei modelli più generali. Potevano tradurre sintassi accuratamente ma spesso perdevano sfumature di logica aziendale o producevano codice che, seppur corretto, non era ottimale per il contesto.

Un'intuizione cruciale emersa da questi test è stata che la qualità dei risultati dipendeva criticamente da come l'AI veniva interrogata. Un'istruzione generica «traduci questo COBOL in Java» produceva risultati mediocri. Ma istruzioni più sofisticate che fornivano contesto, specificavano requisiti, e guidavano il processo di ragionamento producevano risultati drammaticamente migliori.

Ho anche esplorato la possibilità di combinare multiple AI per sfruttare i punti di forza di ciascuna. Un flusso di lavoro dove un'AI analizzava e comprendeva il codice, un'altra generava la conversione, e una terza revisionava e suggeriva miglioramenti mostrava promesse.

Questo approccio «ensemble» poteva potenzialmente superare le limitazioni di ogni singolo modello.

La valutazione ha anche rivelato limitazioni importanti. Tutti i modelli avevano limiti di token che potevano essere problematici per programmi COBOL su larga scala. La consistenza tra esecuzioni successive non era sempre garantita. Nessun modello aveva addestramento specifico su COBOL di livello aziendale, affidandosi invece su qualunque codice COBOL fosse presente nei dati di addestramento pubblici.

Nonostante queste limitazioni, il potenziale era innegabile. In pochi secondi, questi sistemi potevano produrre conversioni che avrebbero richiesto ore a uno sviluppatore umano, e con qualità che spesso superava quello che il mio analizzatore deterministico poteva sperare di raggiungere anche con mesi di sviluppo aggiuntivo.

Progettazione del sistema basato su AI

La progettazione del nuovo sistema basato su AI ha richiesto un ripensamento completo dell'architettura e del flusso di lavoro. Invece di tentare di codificare regole per ogni possibile costruito COBOL, il nuovo approccio si basava su principi fondamentalmente diversi che sfruttavano le capacità uniche dell'AI generativa.

Il primo principio era quello della comprensione contestuale. Invece di analisi meccanica, il sistema doveva comprendere lo scopo e il contesto del codice COBOL. Questo significava non solo analizzare sintassi ma identificare schemi di logica aziendale, comprendere il dominio applicativo, e preservare l'intento oltre che la funzionalità.

Il secondo principio era la generazione idiomantica. Il codice prodotto doveva essere non solo funzionalmente equivalente ma stilisticamente appropriato per il linguaggio di destinazione. Questo richiedeva che l'AI comprendesse non solo la sintassi Java ma le convenzioni, gli schemi, e le migliori pratiche della comunità.

Il terzo principio era l'iterazione intelligente. Riconoscendo che nessuna conversione sarebbe perfetta al primo tentativo, il sistema doveva supportare raffinamento iterativo basato su riscontro e validazione.

L'architettura di alto livello che ho progettato rifletteva questi principi:

Il livello di pre-elaborazione si occupava di preparare il codice COBOL per l'elaborazione AI. Questo includeva:

- Segmentazione intelligente del codice in blocchi gestibili
- Estrazione di metadati e contesto
- Identificazione di dipendenze e relazioni tra componenti

- Preparazione di informazioni supplementari (dizionari dati, documentazione regole aziendali)

Il livello di costruzione del contesto costruiva una comprensione olistica del programma:

- Analisi dello scopo generale del programma
- Identificazione di schemi di dominio (elaborazione batch, transazionale, reportistica)
- Mappatura di relazioni tra strutture dati e logica di elaborazione
- Creazione di una «narrativa» del programma per guidare la conversione

Il livello di orchestrazione della conversione gestiva l'interazione con l'AI:

- Costruzione di istruzioni ottimizzate per ogni tipo di conversione
- Gestione di conversazioni multi-turno per raffinamento
- Coordinamento tra multiple AI per compiti specializzati
- Gestione dei limiti di token attraverso segmentazione intelligente

Il livello di validazione e raffinamento assicurava qualità:

- Validazione sintattica del codice generato
- Validazione semantica attraverso generazione di test
- Raffinamento iterativo basato su risultati di validazione
- Capacità di intervento umano per casi ambigui

Il livello di post-elaborazione finalizzava l'output:

- Organizzazione del codice in struttura di progetto appropriata
- Generazione di configurazione di compilazione e dipendenze
- Creazione di documentazione comprensiva
- Preparazione di rapporti di migrazione e note

Un aspetto chiave della progettazione era la modularità e estensibilità. Ogni livello era progettato come componente indipendente con interfacce ben definite. Questo permetteva:

- Scambio di diversi fornitori AI senza impattare il resto del sistema
- Aggiunta di nuovi passi di validazione o raffinamento
- Personalizzazione per domini o requisiti specifici
- Evoluzione indipendente di componenti

Il flusso di dati attraverso il sistema era accuratamente orchestrato. Un programma COBOL entrava nel sistema e veniva prima analizzato per comprenderne la struttura e lo scopo. Questa comprensione guidava la segmentazione - programmi con chiari confini funzionali potevano essere divisi di conseguenza, mentre codice altamente accoppiato rimaneva insieme.

Ogni segmento passava attraverso conversione con istruzioni personalizzate basate sul tipo di codice. Le strutture dati usavano istruzioni che enfatizzavano sicurezza dei tipi e valida-

zione. La logica aziendale usava istruzioni che preservavano correttezza algoritmica mentre modernizzavano l'implementazione. Le operazioni I/O usavano istruzioni consapevoli di moderni schemi di persistenza.

I risultati di ogni conversione erano validati e raffinati iterativamente. Errori di validazione o ambiguità attivavano conversazioni di chiarimento con l'AI. Il sistema manteneva contesto tra iterazioni, permettendo raffinamento progressivo verso il risultato desiderato.

Un'innovazione particolare era il concetto di «punteggio di confidenza». Il sistema assegnava punteggi di confidenza a diverse parti della conversione basato su fattori come:

- Complessità del codice sorgente
- Ambiguità o schemi inusuali
- Consistenza dei risultati attraverso multiple interrogazioni AI
- Risultati dei test di validazione

Sezioni a bassa confidenza erano segnalate per revisione umana, permettendo un approccio pragmatico dove l'automazione era usata dove affidabile e l'esperienza umana dove necessaria.

La progettazione includeva anche capacità per apprendimento e miglioramento. Il sistema registrava tutte le conversioni, validazioni, e raffinamenti. Questi registri potevano essere analizzati per identificare schemi di successo e fallimento, informando miglioramenti alle istruzioni e al processo.

Quarto periodo: implementazione della soluzione guidata da AI

Le ultime quattro settimane del tirocinio sono state dedicate all'implementazione completa del sistema di migrazione basato su AI. Questo periodo è stato caratterizzato da intensa sperimentazione, iterazione rapida, e scoperte continue che hanno validato e raffinato l'approccio guidato da AI.

Sviluppo dell'ingegneria delle istruzioni

Il cuore del sistema guidato da AI risiedeva nella qualità e sofisticazione delle istruzioni utilizzate per guidare l'intelligenza artificiale. L'ingegneria delle istruzioni si è rivelata essere sia un'arte che una scienza, richiedendo profonda comprensione di come le AI interpretano e rispondono a diverse formulazioni.

Ho iniziato con un approccio sistematico allo sviluppo delle istruzioni, creando una struttura organizzata che categorizzava istruzioni per scopo e ottimizzava ciascuna per il suo caso d'uso specifico. La struttura includeva:

Istruzioni di analisi base: Queste istruzioni erano progettate per comprensione iniziale del codice COBOL. Invece di semplicemente chiedere «cosa fa questo codice», ho sviluppato istruzioni multi-sfaccettate che estraevano livelli di comprensione.

Un'istruzione tipica per l'analisi iniziale guidava l'AI attraverso un processo strutturato di comprensione. Chiedeva di identificare lo scopo aziendale principale del programma, non solo la sua funzionalità tecnica. Richiedeva l'identificazione di strutture dati chiave e le loro relazioni. Sollecitava l'identificazione di dipendenze esterne come file e database. Infine, chiedeva di evidenziare regole aziendali critiche o logica che dovevano essere preservate esattamente.

La formulazione specifica delle istruzioni si è evoluta attraverso iterazione. Versioni iniziali producevano risposte generiche. Raffinamenti successivi aggiungevano specificità e contesto che guidavano l'AI verso intuizioni più preziose. Per esempio, specificare il dominio probabile (bancario, assicurativo) aiutava l'AI a riconoscere schemi specifici del dominio.

Istruzioni di conversione strutture dati: La conversione di strutture dati COBOL richiedeva istruzioni specializzate che comprendevano le caratteristiche uniche della definizione dati COBOL. Ho sviluppato un sistema di modelli che adattava istruzioni basato sul tipo di struttura da convertire.

Per strutture dati gerarchiche, le istruzioni enfatizzavano la preservazione delle relazioni genitore-figlio mentre modernizzavano la rappresentazione. Guidavano l'AI a considerare se l'appiattimento fosse appropriato o se classi annidate preservassero meglio la semantica. Per strutture con REDEFINES, le istruzioni spiegavano il concetto di sovrapposizione di memoria e guidavano verso implementazioni Java appropriate usando tipi unione o accessori multipli.

Una svolta è arrivata quando ho iniziato a includere esempi nelle istruzioni. Non esempi di codice (per evitare condizionamenti), ma esempi di schemi. Per esempio, spiegando che «COBOL spesso usa campi separati per anno, mese, giorno che dovrebbero essere combinati in un moderno oggetto data» guidava l'AI verso modernizzazioni sensate.

Istruzioni di conversione logica aziendale: Convertire la PROCEDURE DIVISION richiedeva le istruzioni più sofisticate. Qui, preservare comportamento esatto mentre si modernizzava l'implementazione era cruciale. Ho sviluppato un approccio multi-stadio dove diverse istruzioni gestivano diversi aspetti della conversione.

Il primo stadio si focalizzava sulla comprensione del flusso. L'AI era guidata a identificare il flusso di elaborazione principale, strutture di ciclo, rami condizionali, e punti di uscita. Questo creava una «mappa» della logica che informava la conversione successiva.

Il secondo stadio gestiva la conversione effettiva. Le istruzioni qui erano accuratamente formulate per bilanciare fedeltà all'originale con modernizzazione. Specificavo che mentre il comportamento doveva essere identico, l'implementazione doveva usare idiomi Java moderni. Le strutture di ciclo dovevano usare costrutti Java appropriati. La gestione degli errori doveva transitare da codici di ritorno a eccezioni dove sensato.

Un aspetto particolare era la gestione di costrutti specifici COBOL. Per istruzioni PERFORM, sviluppai istruzioni che spiegavano le varie semantiche e guidavano verso implementazioni Java appropriate. Per esempio, PERFORM VARYING con variabili multiple era mappato su cicli annidati con chiaro ambito delle variabili.

Istruzioni di conversione SQL: L'SQL incorporato presentava sfide uniche che richiedevano istruzioni specializzate. L'SQL incorporato COBOL ha assunzioni e schemi che non traducono direttamente a JDBC o ORM moderni.

Le istruzioni per conversione SQL includevano contesto su confini transazionali, ciclo di vita dei cursori, mappatura delle variabili ospite, e schemi di gestione errori. Guidavano l'AI a considerare se JDBC grezzo, istruzioni preparate, o anche JPA fosse più appropriato basato sul caso d'uso.

Processo di raffinamento iterativo: Un'innovazione chiave era lo sviluppo di istruzioni per raffinamento iterativo. Invece di aspettarsi risultati perfetti in un solo colpo, progettai il sistema per conversazioni multi-turno dove ogni iterazione migliorava il risultato.

Le istruzioni di raffinamento erano formulate per essere specifiche sui problemi mentre davano all'AI libertà di trovare soluzioni. Per esempio: «Il codice generato implementa correttamente la logica ma usa catene if-else verbose. Ristruttura usando espressioni switch Java o corrispondenza di schemi dove appropriato, mantenendo esatto stesso comportamento.»

Questo approccio iterativo mimava come uno sviluppatore umano avrebbe affrontato la conversione - prima facendola funzionare, poi rendendola pulita e idiomatica.

Tecniche di ottimizzazione delle istruzioni: Attraverso estesa sperimentazione, ho scoperto diverse tecniche che miglioravano significativamente la qualità dei risultati:

Specificazione del ruolo: Iniziare istruzioni con «Sei un esperto sia in COBOL che in Java con profonda comprensione della modernizzazione aziendale...» preparava l'AI per la mentalità giusta.

Vincoli espliciti: Specificare chiaramente cosa preservare (logica aziendale, precisione decimale) e cosa modernizzare (sintassi, schemi) riduceva l'ambiguità.

Incoraggiamento del ragionamento: Chiedere all'AI di «pensare attraverso» la conversione prima di generare codice produceva risultati più riflessivi.

Criteri di qualità: Includere criteri espliciti per output «buono» guidava l'AI oltre la mera correttezza verso genuina qualità.

Documentazione e preservazione della conoscenza: Un aspetto critico dell'ingegneria delle istruzioni era assicurare che la conoscenza incorporata nel COBOL non fosse persa. Sviluppai istruzioni che specificamente istruivano l'AI a:

- Preservare terminologia aziendale nei nomi di variabili e metodi (con adattamento a convenzioni Java)
- Generare JavaDoc comprensiva che spiegava non solo cosa ma perché
- Includere commenti che mappavano nuovo codice a COBOL originale per tracciabilità
- Creare documentazione separata di regole aziendali e assunzioni

Implementazione del traduttore completo

Con una solida struttura di ingegneria delle istruzioni in posizione, ho proceduto a implementare il sistema completo di traduzione dall'inizio alla fine. Questa implementazione rappresentava la culminazione di tutto l'apprendimento e la sperimentazione delle settimane precedenti.

Il sistema era architettato come una catena di elaborazione modulare dove ogni stadio aveva responsabilità ben definite ma poteva comunicare contesto a stadi successivi. Questa architettura permetteva sia elaborazione efficiente che capacità di ritorno quando il raffinamento era necessario.

Elaborazione input e analisi: Il primo componente gestiva l'acquisizione di file sorgente COBOL. Invece dell'analisi complessa tentata nell'approccio tradizionale, questo componente faceva analisi strutturale minima - abbastanza per segmentare intelligentemente il codice senza tentare comprensione completa.

La strategia di segmentazione era sofisticata. Invece di conteggi di linee arbitrari o confini sintattici, usava euristiche per identificare unità logiche. Una sezione di elaborazione batch rimaneva insieme. Strutture dati correlate erano raggruppate. Le dipendenze erano tracciate per assicurare che codice correlato non fosse artificialmente separato.

Il componente produceva anche metadati sulla struttura generale del programma - conteggi di diversi tipi di istruzioni, metriche di complessità, identificazione di schemi comuni. Questi metadati informavano decisioni a valle su come approcciare la conversione.

Assemblaggio del contesto: Prima di iniziare la conversione effettiva, il sistema assemblava contesto comprensivo. Questo includeva non solo il codice da convertire ma anche:

- Copybook correlati e file di inclusione
- Voci del dizionario dati se disponibili
- Documentazione aziendale estratta da commenti
- Informazioni dedotte sul dominio e scopo

Questo contesto era formattato in modo che l'AI potesse facilmente fare riferimento durante la conversione. Informazioni critiche erano evidenziate. Le relazioni erano rese esplicite. Le ambiguità erano segnalate per attenzione.

Orchestrazione della conversione: Il cuore del sistema era l'orchestratore della conversione. Questo componente gestiva le interazioni con servizi AI, implementando controllo del flusso sofisticato e gestione degli errori.

Per ogni segmento di codice, l'orchestratore:

1. Selezionava modelli di istruzioni appropriati basati sul tipo di codice
2. Istanziava istruzioni con contesto specifico
3. Gestiva chiamate API con logica di ripetizione e limitazione della frequenza
4. Elaborava risposte ed estraeva codice generato
5. Attivava validazione e raffinamento se necessario

L'orchestratore manteneva contesto conversazionale attraverso interazioni multiple. Se l'AI chiedeva chiarimenti o suggeriva alternative, l'orchestratore poteva gestire conversazioni multi-turno per raggiungere risultati ottimali.

Una sfida particolare era gestire limiti di token. Grandi programmi COBOL potevano eccedere i limiti di singole chiamate API. L'orchestratore implementava segmentazione intelligente che:

- Divideva il codice a confini logici
- Manteneva sovrapposizione di contesto tra blocchi
- Ricombinava risultati coerentemente
- Gestiva dipendenze tra blocchi

Motore di validazione: La validazione post-generazione era critica per assicurare qualità. Il motore di validazione implementava livelli multipli di controllo:

La validazione sintattica assicurava che il Java generato fosse compilabile. Usava l'API del compilatore Java per tentare la compilazione e analizzare messaggi di errore per riscontro all'AI.

La validazione semantica era più sofisticata. Il sistema generava casi di test basati sulla comprensione del comportamento COBOL e verificava che il Java producesse stessi risultati. Per programmi con chiari schemi input/output, poteva anche generare test automatici.

La validazione dello stile controllava che il codice seguisse convenzioni Java e migliori pratiche. Usava strumenti di analisi statica per identificare cattivi odori del codice, complessità eccessiva, o schemi non-idiomatici.

Ciclo di raffinamento: Quando la validazione identificava problemi, il ciclo di raffinamento si attivava. Questo componente costruiva istruzioni mirate che descrivevano problemi specifici e chiedeva correzioni.

Il raffinamento era iterativo ma limitato - dopo un numero stabilito di tentativi, sezioni problematiche erano segnalate per revisione umana piuttosto che continuare indefinitamente. Questo approccio pragmatico riconosceva che l'automazione perfetta non era sempre raggiungibile o desiderabile.

Assemblaggio e organizzazione del codice: Una volta che tutti i segmenti erano convertiti e validati, il sistema assemblava il progetto Java completo. Questo andava oltre il semplice concatenare file:

- La struttura dei pacchetti era derivata analizzando l'organizzazione del codice
- Le classi erano separate appropriatamente basate sulla coesione
- Utilità comuni erano estratte per ridurre duplicazione
- Interfacce erano generate dove esistevano implementazioni multiple

Il sistema faceva anche decisioni architetturali. Un programma COBOL monolitico poteva essere diviso in classi Java multiple basate su confini funzionali. Strutture dati condivise diventavano classi modello separate. La logica aziendale era organizzata in classi di servizio con responsabilità chiare.

Integrazione SQL e database: La gestione dell'SQL incorporato richiedeva attenzione speciale. Il sistema non solo convertiva istruzioni SQL ma modernizzava l'intero approccio all'interazione con il database.

L'SQL incorporato COBOL tipicamente usa variabili ospite e gestione esplicita dei cursori. Il sistema convertiva questi in accesso database Java moderno:

- Query semplici diventavano istruzioni preparate JDBC

- Operazioni complesse con cursori erano avvolte in schemi iteratore
- I confini transazionali erano preservati ma implementati usando API di transazione Java
- La gestione degli errori transitava da controllo SQLCODE ad approcci basati su eccezioni

Per progetti con estesa interazione database, il sistema poteva anche suggerire generazione di entità JPA, sebbene questo rimanesse opzionale dato che richiedeva ristrutturazione più estesa.

Preservazione della logica aziendale: Durante l'intero processo, preservare l'esatta logica aziendale era fondamentale. Il sistema implementava diversi meccanismi per assicurare questo:

- Commenti nel codice generato che tracciavano indietro ai numeri di linea COBOL originali
- Preservazione dei nomi di variabili originali (adattati a convenzioni Java) per mantenere terminologia di dominio
- Generazione di JavaDoc dettagliato che spiegava scopo aziendale, non solo implementazione tecnica
- Creazione di documentazione separata delle regole aziendali estratta dal COBOL

Generazione automatica di progetti Maven

L'ultima componente sviluppata del sistema trasformava l'output da semplice collezione di file Java a progetti completi pronti per l'azienda. Questa capacità elevava il sistema da strumento di conversione a soluzione di modernizzazione comprensiva.

La decisione di generare progetti Maven completi era strategica. Maven rappresenta lo standard de facto per progetti Java aziendali, con il suo ecosistema maturo di plugin, gestione delle dipendenze, e ciclo di vita di compilazione. Generare progetti che seguivano convenzioni Maven significava che l'output poteva essere immediatamente integrato in flussi di lavoro di sviluppo moderni.

Struttura del progetto: Il sistema generava una struttura di progetto che seguiva rigorosamente il layout standard Maven. Questa standardizzazione facilitava l'adozione da parte di gruppi di sviluppo che potevano immediatamente riconoscere e navigare la struttura.

La generazione iniziava analizzando il codice convertito per determinare l'organizzazione ottimale. Moduli funzionali naturali nel COBOL diventavano moduli Maven separati. Utilità condivise erano estratte in moduli comuni. La struttura risultante rifletteva principi moderni

di microservizi dove appropriato, mentre manteneva struttura monolitica dove la decomposizione non aggiungeva valore.

Generazione POM: Il cuore di ogni progetto Maven è il Project Object Model (POM). Il sistema generava file POM che erano comprensivi e pronti per la produzione.

La gestione delle dipendenze era particolarmente sofisticata. Il sistema analizzava il codice generato per identificare dipendenze richieste:

- Driver JDBC basati sul database originale
- Strutture di registrazione (predefinito SLF4J con Logback)
- Strutture di test (JUnit 5, Mockito)
- Librerie di utilità per funzionalità specifiche

Le versioni delle dipendenze erano accuratamente scelte per compatibilità e sicurezza. Il sistema manteneva una base di conoscenza di combinazioni di versioni stabili che erano note per funzionare bene insieme.

Il POM includeva anche configurazione di compilazione appropriata:

- Impostazioni del compilatore per versione Java di destinazione
- Gestione delle risorse per file di configurazione
- Configurazione dell'esecuzione dei test
- Configurazione del packaging basata sul target di distribuzione

Profili di compilazione: Riconoscendo che le applicazioni aziendali necessitano configurazioni diverse per ambienti diversi, il sistema generava profili Maven per scenari comuni:

- Profilo di sviluppo con impostazioni per test locali
- Profilo di test con database in memoria e servizi simulati
- Profilo di produzione con impostazioni ottimizzate e configurazioni di sicurezza

Ogni profilo poteva sovrascrivere proprietà, dipendenze, e comportamento di compilazione appropriato per il suo ambiente di destinazione.

Configurazione plugin: Il sistema configurava automaticamente plugin Maven essenziali:

- Plugin del compilatore con versioni sorgente/destinazione appropriate
- Plugin Surefire per esecuzione test con impostazioni di memoria appropriate per grandi suite di test
- Plugin delle risorse per gestione di file non-Java
- Plugin di assemblaggio per creare pacchetti di distribuzione

Per progetti con esigenze specifiche, plugin aggiuntivi erano configurati. Applicazioni intensive di database ricevevano Flyway o Liquibase per gestione delle migrazioni. Servizi web ricevevano plugin appropriati per generare client o endpoint.

Generazione documentazione: Oltre al codice stesso, il sistema generava documentazione completa del progetto:

- README.md con panoramica del progetto, istruzioni di configurazione, e note di migrazione
- MIGRATION_NOTES.md dettagliando decisioni specifiche prese durante la conversione
- Documentazione API se il progetto esponeva servizi
- Record di decisioni architetturali per scelte strutturali significative

La documentazione era generata usando la comprensione dell'AI del codice, producendo documentazione che era effettivamente utile piuttosto che standard generica.

Struttura dei test: Il sistema generava anche una struttura completa di test. Mentre non poteva generare logica di test comprensiva senza contesto aggiuntivo, creava:

- Scheletri di classi di test per ogni classe principale
- Casi di test base per scenari ovvi
- Configurazione dati di test basata su strutture dati
- Struttura test di integrazione per operazioni database

Questo dava agli sviluppatori un punto di partenza per costruire copertura di test appropriata.

Esternalizzazione della configurazione: Seguendo i principi delle applicazioni a dodici fattori, il sistema esternalizzava tutta la configurazione. I programmi COBOL spesso hanno configurazione codificata; il sistema identificava questi e li estraeva in:

- File application.properties per configurazione stile Spring
- Mappature di variabili d'ambiente per distribuzioni in contenitori
- Classi di configurazione che fornivano accesso sicuro ai tipi

Prontezza DevOps: Riconoscendo le pratiche moderne di distribuzione, il sistema generava anche artefatti relativi a DevOps:

- Dockerfile per distribuzione in contenitori
- Manifesti Kubernetes base per orchestrazione
- Flusso di lavoro GitHub Actions per CI/CD
- .gitignore configurato appropriatamente per progetti Java

Questi artefatti erano modelli che richiedevano personalizzazione ma fornivano solidi punti di partenza.

Risultati raggiunti

L'implementazione completa del sistema guidato da AI ha portato a risultati che hanno superato significativamente le aspettative iniziali del progetto. Quello che era iniziato come un esperimento ambizioso si è trasformato in una soluzione pratica e potente che dimostrava il potenziale trasformativo dell'AI nella modernizzazione del software.

Impatto dell'AI sui tempi di sviluppo

La quantificazione dell'impatto dell'intelligenza artificiale sui tempi di sviluppo ha rivelato miglioramenti che andavano oltre la semplice accelerazione lineare. L'AI non solo velocizzava il processo ma lo trasformava qualitativamente, rendendo possibile quello che sarebbe stato impraticabile con approcci tradizionali.

Metriche comparative: Le metriche raccolte durante il progetto raccontavano una storia convincente:

L'approccio basato su analizzatore sintattico aveva richiesto tre settimane intensive per implementare supporto per circa il 25% dei costrutti COBOL necessari. Estrapolando, un analizzatore completo avrebbe richiesto almeno 12 settimane, assumendo complessità lineare (un'assunzione ottimistica data la natura interconnessa dei costrutti COBOL).

Il sistema guidato da AI è stato sviluppato e reso completamente funzionale in quattro settimane. Ma questa comparazione sottostima il vero impatto. L'analizzatore avrebbe prodotto solo conversione grezza - la post-elaborazione manuale per rendere il codice pronto per la produzione avrebbe aggiunto settimane o mesi aggiuntivi. Il sistema AI produceva codice già idiomatrico e ben strutturato.

Velocità di conversione: Una volta operativo, il sistema dimostrava velocità di conversione straordinarie:

- Programmi semplici (< 500 LOC): minuti invece di giorni
- Programmi medi (500-2000 LOC): ore invece di settimane
- Programmi complessi (> 2000 LOC): giorni invece di mesi

Ma oltre la velocità grezza, la consistenza era notevole. Uno sviluppatore umano potrebbe convertire velocemente sezioni semplici ma rallentare significativamente su logica complessa. L'AI manteneva prestazioni consistenti indipendentemente dalla complessità, con solo modesti incrementi nel tempo per codice più impegnativo.

Eliminazione del debito tecnico: Un beneficio non immediatamente ovvio era l'eliminazione dell'accumulo di debito tecnico. La conversione manuale, specialmente sotto

pressione temporale, tende a produrre scorciatoie e compromessi. «Lo sistema dopo» diventa debito tecnico permanente.

L'AI non aveva incentivo a prendere scorciatoie. Ogni conversione seguiva migliori pratiche, implementava gestione degli errori appropriata, includeva documentazione. Il risultato era codice che non solo funzionava ma era manutenibile dal primo giorno.

Scalabilità senza precedenti: Forse l'impatto più drammatico era sulla scalabilità. Gli approcci tradizionali scalano linearmente - raddoppiare il codice significa raddoppiare tempo e risorse. L'approccio guidato da AI scalava sub-linearmente grazie a:

- Riconoscimento di schemi che migliorava con più esempi
- Riutilizzo di conversioni simili
- Parallelizzazione naturale di conversioni indipendenti

Questo rendeva fattibile la migrazione di intere basi di codice legacy che sarebbero state economicamente impossibili con approcci manuali.

Democratizzazione della modernizzazione: L'AI riduceva drammaticamente la barriera d'ingresso per progetti di modernizzazione. Non era più necessario un gruppo di esperti COBOL e architetti Java. Un piccolo gruppo con comprensione base poteva guidare il processo, concentrandosi su validazione e raffinamento piuttosto che traduzione manuale.

Questo effetto di democratizzazione era profondo. Organizzazioni che non potevano permettersi grandi progetti di modernizzazione ora avevano un percorso da seguire. Il calcolo del ritorno sull'investimento cambiava fundamentalmente quando i costi scendevano di un ordine di grandezza.

Analisi qualitativa dei risultati

Oltre alle metriche grezze, la qualità dell'output prodotto dal sistema guidato da AI meritava analisi approfondita. Non si trattava solo di produrre codice funzionante, ma di generare software che i gruppi di sviluppo avrebbero effettivamente voluto mantenere ed evolvere.

Gestione delle divisioni COBOL: Il sistema dimostrava comprensione sofisticata di come ogni divisione COBOL dovesse essere trasformata per contesti moderni:

La IDENTIFICATION DIVISION non era semplicemente scartata ma trasformata in metadati preziosi. Le informazioni del programma diventavano parte della documentazione, le informazioni dell'autore erano preservate in formati compatibili con il controllo versione, e lo scopo del programma era catturato in documentazione a livello di classe che effettivamente aiutava gli sviluppatori a comprendere il sistema.

La trasformazione della ENVIRONMENT DIVISION era particolarmente intelligente. Invece di codificare dipendenze ambientali, il sistema generava livelli di astrazione puliti. Le associazioni di file diventavano proprietà di configurazione. Le dipendenze di sistema erano isolate in classi di configurazione dedicate. Il risultato era codice veramente portabile che seguiva i principi a dodici fattori.

La trasformazione della DATA DIVISION mostrava il vero potere della comprensione AI. Strutture gerarchiche complesse erano analizzate per il loro vero scopo e convertite appropriatamente:

- Strutture semplici diventavano POJO con validazione appropriata
- Strutture annidate complesse erano ristrutturare in modelli a oggetti appropriati
- Strutture riutilizzate erano estratte in classi condivise
- La sicurezza dei tipi era aggiunta dove il COBOL era permissivo

La trasformazione della PROCEDURE DIVISION era dove l'AI brillava di più. Invece di traduzione meccanica, il sistema comprendeva il flusso e lo scopo, producendo codice che era riconoscibilmente Java:

- Sequenze procedurali diventavano metodi ben strutturati
- Condizioni complesse erano semplificate usando costrutti moderni
- I cicli erano convertiti nelle iterazioni Java più appropriate
- La gestione degli errori transitava da procedurale a basata su eccezioni

Metriche di qualità del codice: Analizzando il codice prodotto attraverso metriche di qualità standard rivelava risultati impressionanti:

La complessità ciclomatica era consistentemente ridotta del 20-40% rispetto al COBOL originale. Questo non era accidentale - l'AI naturalmente ristrutturava condizionali complessi in strutture più pulite.

La duplicazione del codice era virtualmente eliminata. Dove i programmi COBOL spesso hanno sezioni copiate e modificate, l'AI riconosceva schemi ed estraeva funzionalità comuni.

Le lunghezze dei metodi erano appropriate - l'AI naturalmente decomponendo lunghe sequenze procedurali in metodi focalizzati che seguivano il principio di responsabilità singola.

Le convenzioni di denominazione erano consistenti e significative. L'AI preservava terminologia aziendale mentre adattava a convenzioni Java, risultando in codice che gli esperti di dominio potevano ancora riconoscere.

Generazione di Java idiomatice: Il codice generato non sembrava «COBOL in sintassi Java» ma genuino Java:

- Uso appropriato di collezioni invece di array

- Schemi costruttore per costruzione di oggetti complessi
- Stream per elaborazione dati dove sensato
- Optional per gestione di possibili valori nulli
- Enum per insiemi di valori finiti invece di costanti

Questo codice idiomático era cruciale per manutenzione a lungo termine. Gli sviluppatori non dovevano imparare «schemi COBOL» per lavorare con il codice - potevano applicare normale esperienza Java.

Eccellenza nella documentazione: La documentazione generata era un particolare punto di forza:

- La documentazione a livello di classe spiegava scopo aziendale, non solo dettagli tecnici
- La documentazione dei metodi includeva precondizioni, postcondizioni, e regole aziendali
- I commenti in linea erano usati con giudizio per spiegare logica non ovvia
- I commenti di mappatura collegavano codice Java indietro a COBOL originale per tracciabilità

La documentazione non era generica o standard ma genuinamente informativa, catturando conoscenza che altrimenti sarebbe stata persa nella migrazione.

Modernizzazione architetturale: Oltre la conversione linea per linea, il sistema faceva miglioramenti architetturali riflessivi:

- Programmi monolitici erano decomposti in componenti logici
- Chiara separazione delle preoccupazioni era introdotta
- Schemi di iniezione delle dipendenze erano suggeriti dove appropriato
- L'accesso al database era isolato in livelli di repository
- La logica aziendale era separata dalle preoccupazioni infrastrutturali

Questi miglioramenti non erano arbitrari ma basati su analisi della struttura del codice e dipendenze.

Risultati quantitativi

I deliverable concreti del progetto fornivano evidenza tangibile del successo dell'approccio guidato da AI. Numeri e metriche raccontavano una storia convincente di risultati oltre le aspettative iniziali.

Portfolio di conversioni completate: Durante le settimane finali del progetto, ho convertito con successo tre programmi COBOL completi di complessità crescente:

Il primo progetto era un calcolatore di interessi di 450 linee. Questo programma apparentemente semplice nascondeva complessità sorprendente nella gestione della precisione

decimale e calcoli di interesse composto. La conversione produceva 380 linee di Java pulito che:

- Preservava esatta precisione decimale usando BigDecimal appropriatamente
- Ristrutturava calcoli ripetitivi in metodi riutilizzabili
- Aggiungeva validazione appropriata per tassi di interesse e principali
- Includeva test unitari comprensivi che validavano calcoli

Il secondo progetto, un sistema di gestione inventario di 890 linee, presentava sfide in complessità delle strutture dati e gestione file. La conversione risultava in 750 linee di Java ben strutturato che:

- Trasformava strutture file piatte in oggetti di dominio appropriati
- Implementava schema repository per accesso dati
- Aggiungeva supporto transazioni per consistenza
- Creava livello di servizio con operazioni aziendali chiare
- Generava endpoint API RESTful per integrazione moderna

Il terzo e più ambizioso progetto era un generatore di rapporti batch di 1200 linee. Questo mostrava schemi tipici di elaborazione batch COBOL con aggregazione dati complessa e formattazione. La conversione produceva 950 linee di Java moderno che:

- Sfruttava API Stream per elaborazione dati efficiente
- Implementava generazione rapporti basata su modelli
- Aggiungeva capacità di pianificazione usando concetti Spring Batch
- Creava definizioni rapporti configurabili
- Includeva ottimizzazioni prestazioni per grandi volumi di dati

Metriche di copertura: Analizzando la completezza delle conversioni rivelava risultati forti:

La copertura funzionale raggiungeva il 95% - ogni funzione aziendale maggiore era migrata con successo. Il rimanente 5% erano principalmente caratteristiche deprecate o funzioni specifiche della piattaforma che non avevano senso nel contesto moderno.

La copertura sintattica era 88% - la vasta maggioranza dei costrutti COBOL erano gestiti automaticamente. I costrutti non gestiti erano principalmente caratteristiche esoteriche o deprecate raramente usate nel codice di produzione.

La copertura dei test del codice convertito era in media 80%, con logica aziendale critica che raggiungeva 100%. L'AI generava struttura di test e casi di test base che gli sviluppatori potevano estendere.

Confronti prestazionali: Il benchmarking delle applicazioni convertite contro gli originali mostrava risultati interessanti:

- L'uso della CPU generalmente diminuiva del 15-30% grazie ad algoritmi più efficienti
- L'uso della memoria aumentava moderatamente (overhead Java) ma rimaneva ben entro le capacità hardware moderne
- Le operazioni I/O erano significativamente più veloci grazie a buffering e caching moderni
- Il throughput complessivo migliorava del 20-50% in scenari tipici

Validazione qualità del codice: L'esecuzione di strumenti di analisi standard del settore sul codice generato produceva punteggi eccellenti:

- SonarQube riportava zero problemi critici, problemi maggiori minimi
- Le violazioni CheckStyle erano minime e principalmente basate su preferenze
- L'analisi statica PMD non trovava problemi significativi
- La scansione di sicurezza non identificava vulnerabilità

Completezza della migrazione: Ogni progetto convertito era veramente «pronto per la produzione»:

- Completamente compilabile senza errori o avvisi
- Eseguitibile con comportamento funzionale identico all'originale
- Distribuibile usando metodi di distribuzione Java standard
- Manutenibile da sviluppatori Java senza conoscenza COBOL
- Documentato sufficientemente per comprensione ed evoluzione

Il totale di oltre 2000 linee di Java di qualità produttiva rappresentava non solo una prova di concetto ma una genuina dimostrazione che la migrazione guidata da AI era pratica, efficace, e pronta per applicazione nel mondo reale. Il sistema aveva trasformato quello che tradizionalmente era uno sforzo manuale di diversi mesi in un processo che poteva essere completato in giorni con risultati superiori.

Valutazioni retrospettive e prospettive future

Qui introdurrò brevemente il contenuto delle sezioni sottostanti.

Analisi retrospettiva del percorso

In questa sezione analizzerò il soddisfacimento degli obiettivi al capitolo 2 grazie all'approccio AI, confronterò i risultati ottenuti con le stime iniziali basate sullo sviluppo tradizionale e identificherò le *lessons learned* e *best practices* emerse dal progetto.

L'AI come *game changer* nella modernizzazione *software*

In questa sezione descriverò come l'intelligenza artificiale abbia trasformato il progetto da «prototipo dimostrativo» a «soluzione potenzialmente completa», confronterò l'approccio sviluppato con soluzioni *enterprise* come IBM *WatsonX*, analizzerò il ruolo cruciale del *prompt engineering* e valuterò limiti e potenzialità dell'approccio AI-driven.

Crescita professionale e competenze acquisite

In questa sezione descriverò le *hard skills* acquisite in migrazione *legacy*, AI *engineering* e *prompt design*, analizzerò le *soft skills* sviluppate come *problem solving* e adattabilità, illustrerò la visione sistemica della modernizzazione IT maturata e la capacità di valutare e integrare pragmaticamente tecnologie emergenti.

Valore della formazione universitaria nell'era dell'AI

In questa sezione analizzerò come il percorso universitario mi abbia fornito le solide basi metodologiche essenziali per affrontare questa sfida tecnologica, valorizzando in particolare l'approccio al *problem solving* e il metodo di studio critico acquisiti. Descriverò come la formazione teorica ricevuta si sia rivelata fondamentale per comprendere e padroneggiare tecnologie emergenti come l'AI, evidenziando l'importanza dell'approccio universitario che insegna ad «imparare ad imparare».

Roadmap evolutiva e opportunità di sviluppo

In questa sezione descriverò le possibili evoluzioni della soluzione verso il supporto *multi-linguaggio* per altri sistemi *legacy*, analizzerò il potenziale di commercializzazione della soluzione e esplorerò l'uso di *multi-agent systems* per conversioni complesse.

Lista degli acronimi

AI: Artificial Intelligence

CLI: Command Line Interface

COBOL: Common Business-Oriented Language

IT: Information Technology

LLM: Large Language Models

Glossario

Agile: Metodologia di sviluppo software iterativa e incrementale

DevOps: Pratiche che combinano sviluppo software e operazioni IT

Kanban: Sistema di gestione del workflow visuale

Scrum: Framework Agile per la gestione di progetti complessi

legacy: Sistemi informatici datati ma ancora in uso

mainframe: Computer di grandi dimensioni per elaborazioni complesse

microservizi: Architettura software basata su servizi indipendenti

sprint: Periodo di tempo definito per completare un set di attività

stack tecnologico: Insieme di tecnologie software utilizzate per sviluppare un'applicazione

stand-up: Breve riunione giornaliera del team Agile

Sitografia

- [1] CBT Nuggets, «What is COBOL and Who Still Uses It?». Consultato: giugno 2025. [Online]. Disponibile su: <https://www.cbtnuggets.com/blog/technology/programming/what-is-cobol-and-who-still-uses-it>
- [2] Version 1, «Legacy System Modernization: Challenges and Solutions». Consultato: maggio 2025. [Online]. Disponibile su: <https://www.version1.com/insights/legacy-system-modernization/>
- [3] DXC Luxoft, «How come COBOL-driven mainframes are still the banking system of choice?». Consultato: giugno 2025. [Online]. Disponibile su: <https://www.luxoft.com/blog/why-banks-still-rely-on-cobol-driven-mainframe-systems>
- [4] How-To Geek, «What Is COBOL, and Why Do So Many Institutions Rely on It?». Consultato: maggio 2025. [Online]. Disponibile su: <https://www.howtogeek.com/667596/what-is-cobol-and-why-do-so-many-institutions-rely-on-it/>
- [5] CAST Software, «Why COBOL Still Dominates Banking—and How to Modernize». Consultato: giugno 2025. [Online]. Disponibile su: <https://www.castsoftware.com/pulse/why-cobol-still-dominates-banking-and-how-to-modernize>
- [6] New Relic, «2024 State of the Java Ecosystem». Consultato: 12 maggio 2025. [Online]. Disponibile su: <https://newrelic.com/resources/report/2024-state-of-java-ecosystem>