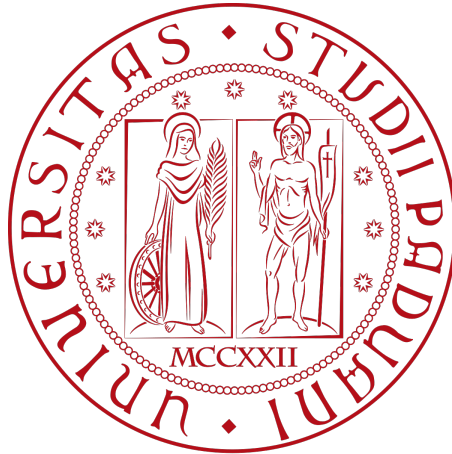


Università degli Studi di Padova

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

Anno Accademico 2024/2025

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: dal <i>parser</i> tradizionale all'AI (AI)	16
<i>Setup</i> iniziale e metodologia di lavoro	16
Primo periodo: immersione nel mondo COBOL	18
Studio del linguaggio e creazione progetti test	18
Mappatura dei <i>pattern</i> e analisi di traducibilità	21
Valutazione delle soluzioni esistenti	24
Secondo periodo: sviluppo del <i>parser</i> tradizionale	26
Implementazione del <i>parser</i> Java	26
Analisi critica e limiti dell'approccio	29
Terzo periodo: pivot verso l'intelligenza artificiale	31
Valutazione delle API di AI generativa	31
Design del sistema AI-powered	33
Quarto periodo: implementazione della soluzione AI-driven	35
Sviluppo del <i>prompt engineering</i>	36
Implementazione del <i>translator</i> completo	38
Generazione automatica di progetti Maven	41
Impatto dell'AI sui tempi di sviluppo	43

Analisi qualitativa dei risultati	45
Risultati quantitativi	47
Valutazioni retrospettive e prospettive future	49
Analisi retrospettiva del percorso	49
L'AI come <i>game changer</i> nella modernizzazione <i>software</i>	49
Crescita professionale e competenze acquisite	49
Valore della formazione universitaria nell'era dell'AI	49
<i>Roadmap</i> evolutiva e opportunità di sviluppo	49
Lista degli acronimi	50
Glossario	51
Sitografia	52

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 variegata, mantenendo una costante spinta all'innovazione e all'adattamento delle soluzioni proposte.

Metodologie e tecnologie all'avanguardia

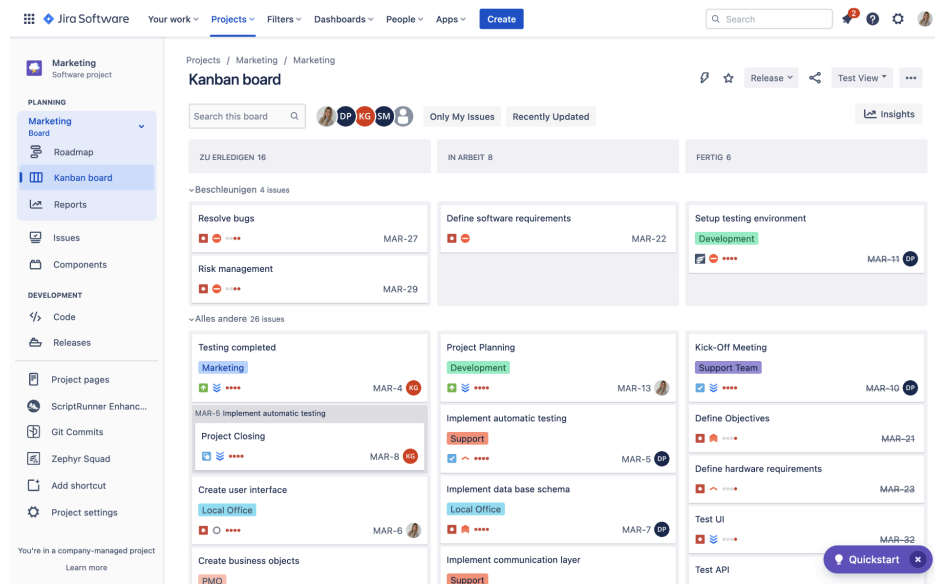
L'approccio metodologico di Miriade si fonda sull'adozione dell'*Agile* come filosofia operativa pervasiva, che permea tutti i processi aziendali e guida l'organizzazione del lavoro quotidiano. Durante il mio *stage*, ho potuto osservare direttamente come questa metodologia venga implementata attraverso *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:

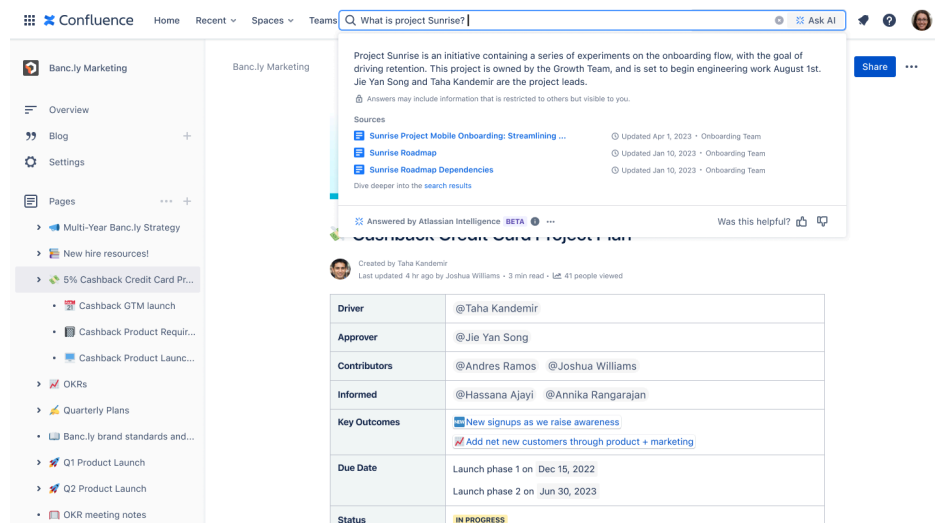
- **Confluence** per la gestione della *knowledge base* aziendale e la documentazione tecnica
- **Jira** per il *tracking* delle attività e la gestione dei progetti
- **Bitbucket** per il versionamento del codice e la collaborazione nello sviluppo

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.

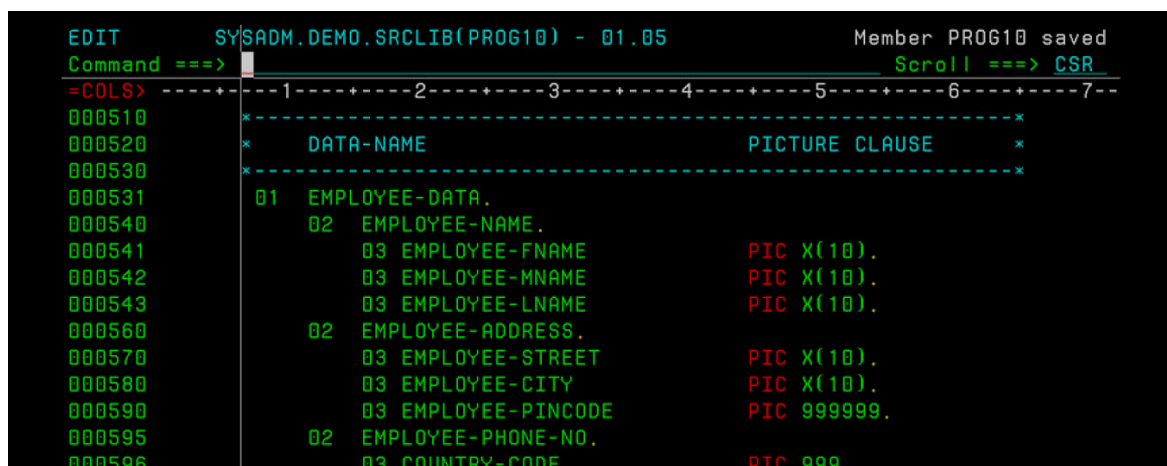
Il progetto di migrazione COBOL-Java

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:

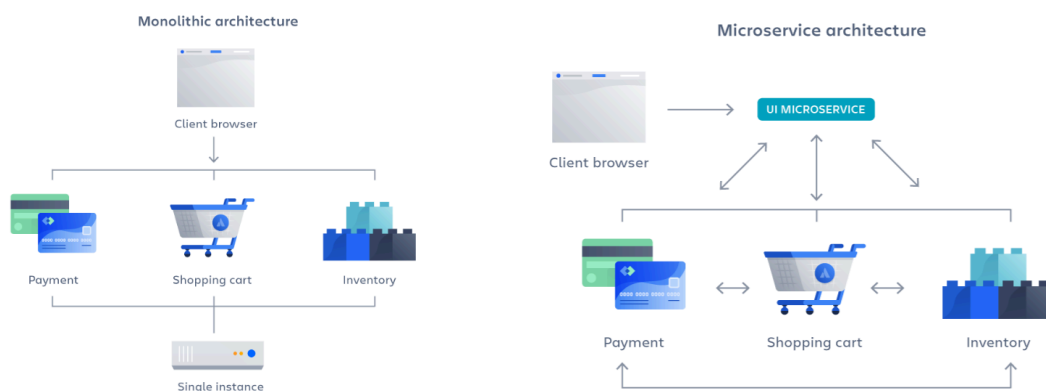
- La crescente difficoltà nel reperire sviluppatori COBOL qualificati [1]
- L'integrazione sempre più complessa con tecnologie moderne [2]
- I rischi operativi derivanti dall'utilizzo di piattaforme *hardware* e *software* che i *vendor* non supportano più attivamente [3]

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: dal *parser* 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 *stage*, 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.

Setup 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 *team* e una gestione strutturata del progetto.

L'implementazione della metodologia Agile si è concretizzata attraverso *sprint* 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'effort 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 pivotare verso una soluzione completamente diversa da quella inizialmente prevista.

I *stand-up* giornalieri, condotti 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 *team* che occasionalmente partecipavano. Questa routine quotidiana ha favorito un approccio iterativo e incrementale, permettendomi di adattare rapidamente la direzione del progetto in base ai *feedback* 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 *tracking* dettagliato delle attività, con la creazione di *ticket* specifici per ogni fase del progetto. La struttura dei *ticket* seguiva il formato standard aziendale, includendo descrizione dettagliata, criteri di accettazione, stima temporale e priorità. Ho imparato l'importanza di mantenere i ticket costantemente aggiornati, non solo per tracciare il progresso, ma anche per documentare le decisioni prese e le motivazioni dietro cambiamenti di direzione. La pratica di collegare i commit di codice ai ticket Jira ha creato una tracciabilità completa dal requisito all'implementazione.

Confluence ha svolto un ruolo fondamentale come repository 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 test comparativi tra diversi approcci
- Log 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 rationale dietro il cambio di strategia.

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

- Un branch `main` per il codice stabile e testato
- Branch `feature/` per ogni nuova funzionalità o esperimento
- Branch `experiment/` per prove e prototipi che potrebbero non confluire nel main
- Tag per marcare milestone significative del progetto

Ho adottato una politica di *commit* frequenti e descrittivi, seguendo le convenzioni aziendali per i messaggi di *commit*. I prefissi standardizzati (`feat:`, `fix:`, `docs:`, `refactor:`, `test:`) non solo rendevano la history più leggibile, ma facilitavano anche la generazione automatica di changelog per le review settimanali. La pratica di scrivere commit message 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
- README files nel repository che documentavano setup, 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 team 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 test

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 production.

La struttura rigida del linguaggio, con le sue quattro divisioni obbligatorie (*Identification, Environment, Data e Procedure*), rappresentava un paradigma completamente diverso rispet-

to 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 team numerosi.

La IDENTIFICATION DIVISION, apparentemente semplice, nascondeva importanti lezioni sulla filosofia COBOL. L'enfasi sulla documentazione embedded 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 framework.

Per consolidare l'apprendimento teorico, ho intrapreso lo sviluppo sistematico di una serie di applicazioni test con complessità crescente. Questo approccio pratico si è rivelato essenziale per interiorizzare non solo la sintassi, ma anche gli idiomi e le best practice 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 l'overhead 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 nume-

rici, 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 range 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 report, mi ha introdotto a uno dei domini principali di COBOL: il processing di grandi volumi di dati. La generazione di report 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 business, incorporando nel linguaggio stesso pattern 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'*embedded SQL* (SQL) 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 *host variables*, 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 null values 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 pattern standard per questa gestione, creando paragraph riutilizzabili per il controllo degli errori comuni. Questa esperienza ha sottolineato l'importanza della disciplina nella programmazione COBOL: senza i safety net automatici 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 questo heritage. Il concetto di unit of work, con COMMIT e ROLLBACK espliciti, richiede una pianificazione attenta del flusso di elaborazione. Ho dovuto imparare a strutturare i programmi in modo che i boundary transazionali fossero chiari e che la recovery da errori fosse sempre possibile.

Mappatura dei *pattern* e analisi di traducibilità

Durante lo sviluppo delle applicazioni test, ho intrapreso un'attività sistematica di catalogazione dei *pattern* 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 pattern a diversi livelli di astrazione. Al livello più basso, ho catalogato le corrispondenze dirette tra costrutti sintattici. Al livello intermedio, ho analizzato pattern di design e idiomi ricorrenti. Al livello più alto, ho esaminato le architetture applicative tipiche e come queste potrebbero essere trasposte in paradigmi moderni.

I pattern 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 profonda-

mente, 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 for loop con contatore, ma PERFORM UNTIL richiede attenzione alla semantica di valutazione della condizione (pre o post test). PERFORM VARYING, specialmente con multiple varying clauses, può risultare in strutture di loop annidate complesse che richiedono refactoring significativo per risultare idiomatiche nel linguaggio target.

La gestione dei dati in COBOL presenta paradigmi unici che ho documentato estensivamente. Le strutture dati gerarchiche, definite attraverso level numbers, non hanno equivalenti diretti nei linguaggi moderni. Ho sviluppato strategie per mappare queste strutture su classi e oggetti, preservando le relazioni gerarchiche attraverso composizione e inheritance 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 object-oriented.

Le PICTURE clauses 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 pattern per convertire le picture clauses più comuni in combinazioni di tipi di dato e formatter, ma alcune clausole complesse (come quelle con insertion editing o floating symbols) richiedevano logica custom 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 stream e file operations moderne

Per questa categoria, ho sviluppato mapping rules relativamente straightforward, 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 target.

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

- File handling con organizzazioni sequential, indexed, e relative che richiedono mapping su database o strutture dati moderne
- PICTURE clauses complesse che necessitano di formatter custom
- Strutture dati gerarchiche che devono essere trasformate in object models
- PERFORM statements complessi che richiedono refactoring del flusso di controllo

Per questi costrutti, ho sviluppato pattern 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-memory con caratteristiche di accesso equivalenti.

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

- GOTO statements che violano i principi di programmazione strutturata
- ALTER verb che modifica dinamicamente il flusso del programma
- REDEFINES che crea multiple viste 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 reengineering. Ho documentato approcci per eliminare i GOTO attraverso refactoring in strutture di controllo appropriate, strategie per gestire REDEFINES attraverso union types 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 pattern COBOL, sebbene tecnicamente traducibili, perderebbero significato se convertiti meccanicamente. Ad esempio, l'uso estensivo di 88-level conditions in COBOL spesso codifica business rules importanti. Una traduzione meccanica in semplici costanti perderebbe la ricchezza semantica; invece, ho proposto l'uso di enumerations 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 refactoring profondo, 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 *enterprise*. 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 ownership, 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 *Pipeline Architecture* con *parser* deterministici rappresentava la soluzione più diffusa nel panorama open-source. Questi sistemi seguono il classico modello di compilatore con fasi distinte: analisi lessicale, parsing sintattico, generazione di abstract syntax tree (AST), trasformazioni sull'AST, e generazione del codice target.

Ho esaminato in dettaglio il *ProLeap COBOL parser* disponibile su GitHub, uno dei progetti open-source più maturi in questo spazio. Il parser, basato su ANTLR4, dimostrava capacità impressionanti nel parsing di COBOL-85 con estensioni per alcuni dialetti vendor-specific. 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. Il parser eccelleva nell'analisi sintattica ma mancava di comprensione semantica profonda. La trasformazione da AST COBOL a codice target rimaneva largamente un esercizio manuale o semi-automa-

tico. Il codice generato, quando automatizzato, tendeva a essere una traduzione letterale che preservava le idiocincrasie COBOL invece di produrre codice idiomático nel linguaggio target.

Ho anche valutato altri tool open-source, ciascuno con i propri punti di forza e debolezza. Alcuni si concentravano su subset specifici di COBOL (come COBOL per specifiche piattaforme), altri tentavano trasformazioni più ambiziose ma con risultati inconsistenti. Un pattern 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 enterprise, il panorama era dominato da soluzioni commerciali sofisticate con prezzi corrispondentemente elevati. Ho avuto l'opportunità di valutare diverse soluzioni leader di mercato attraverso demo, documentazione, e in alcuni casi trial limitati.

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 business del codice. Durante la demo, ho osservato come il sistema potesse identificare pattern di business logic, suggerire refactoring appropriati, e generare codice che seguiva best practices moderne.

Le capacità di WatsonX erano impressionanti: riconoscimento di pattern di dominio (come elaborazione batch tipica del banking), 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 business oltre che tecnico.

Tuttavia, la soluzione presentava barriere significative per un progetto di ricerca: costo proibitivo per licenze anche di sviluppo, natura closed-source che impediva customizzazioni profonde, requisiti di infrastruttura enterprise, e lock-in vendor con ecosistema IBM.

Ho valutato anche altre soluzioni enterprise di vendor come Micro Focus, Modern Systems, e TSRI. Ciascuna aveva approcci unici: alcune si concentravano su migrazioni «lift and shift» che preservavano la struttura COBOL in runtime Java, altre tentavano trasformazioni più radicali. Un tema comune era il trade-off tra automazione e qualità: maggiore era l'automazione, più il codice risultante tendeva a essere non-idiomático e difficile da mantenere.

Un'osservazione cruciale emersa da questa analisi è stata l'identificazione di un gap significativo nel mercato. Le soluzioni open-source, pur essendo accessibili, mancavano di sofisticazione e producevano risultati che richiedevano estensive modifiche manuali.

Le soluzioni enterprise, pur essendo potenti, erano inaccessibili per la maggior parte delle organizzazioni a causa di costi e complessità.

Questo gap 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 vendor con risorse massive.

L'analisi delle soluzioni esistenti ha anche rivelato pattern comuni di failure e success factors:

- Le migrazioni di successo preservavano la business logic 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 embedded 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 business logic con modernizzazione dell'implementazione, e accessibilità con potenza.

Secondo periodo: sviluppo del parser tradizionale

Forte delle conoscenze acquisite sul linguaggio COBOL e dell'analisi approfondita delle soluzioni esistenti, ho intrapreso lo sviluppo di un parser 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 del parser Java

Lo sviluppo del parser 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 testing.

Il design del parser seguiva un'architettura a pipeline con componenti loosely coupled. Il Lexer si occupava della tokenizzazione del codice COBOL, gestendo le peculiarità del

linguaggio come la sensibilità alla colonna, i commenti, e le continuation lines. Il Parser vero e proprio costruiva l'Abstract Syntax Tree basandosi sulla grammatica COBOL. Il Semantic Analyzer arricchiva l'AST con informazioni di tipo e riferimenti. Il Code Generator 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 parser basato su pattern matching 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 naming 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. Il parser 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 FILE-CONTROL paragraph presentava complessità particolare. Ogni file declaration in COBOL include non solo il nome logico del file ma anche dettagli sulla sua organizzazione (sequential, indexed, relative), access mode (sequential, random, dynamic), e gestione degli errori. Ho mappato queste dichiarazioni su configurazioni per l'accesso ai file in Java,

creando abstraction layers che preservavano la semantica COBOL mentre utilizzavano API Java moderne.

La gestione dei file lock e sharing modes 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 file locking APIs e careful synchronization.

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

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

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

La conversione della DATA DIVISION in strutture Java appropriate si è rivelata sorprendentemente complessa. Un approccio naive di mappare ogni group item su una classe Java e ogni elementary item su un field produceva codice verbose e non-idiomatico. Ho sperimentato con diverse strategie:

- Flattening di gerarchie semplici in classi con campi diretti
- Uso di inner classes per preservare strutture gerarchiche complesse
- Generazione di builder patterns per la costruzione di oggetti complessi
- Implementazione di custom serialization per preservare layout di memoria COBOL dove necessario

La gestione delle PICTURE clauses ha richiesto lo sviluppo di un sotto-sistema completo per parsing e interpretazione. Le picture clauses non solo definiscono tipo e size dei dati, ma anche formatting, sign handling, decimal alignment, e altro. Ho creato un mini-parser specializzato per picture clauses che le decomponeva in attributi gestibili e generava codice Java appropriato per validazione e formatting.

Analisi critica e limiti dell'approccio

Dopo tre settimane di sviluppo intensivo, dedicando la maggior parte del tempo all'implementazione e refinement del parser, 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 parser-based. 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. I GOTO statements, sebbene deprecati, sono ancora comuni nel codice legacy. Ancora più problematici sono i PERFORM statements 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 range di paragrafi
- PERFORM TIMES per iterazioni con contatore
- PERFORM UNTIL per loops condizionali
- PERFORM VARYING per loops con variabili di controllo multiple

La complessità emergeva dalle interazioni tra queste forme. Un PERFORM può chiamare un paragrafo che contiene altri PERFORM, creando stack di esecuzione complessi. La semantica di PERFORM THROUGH, dove l'esecuzione continua attraverso paragrafi consecutivi fino a un endpoint, 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 inline il codice dove possibile, ma questo produceva duplicazione massiccia. Un approccio più sofisticato usando state machines 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 fall-through è aliena a Java e richiede careful handling per essere emulata correttamente. La presenza di EXIT statements che possono terminare loops o paragrafi da punti arbitrari complicava ulteriormente l'analisi del flusso.

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

- Host variables devono essere correttamente mappate e type-checked
- La gestione degli errori attraverso SQLCODE deve essere preservata
- Cursors devono mantenere stato tra chiamate
- Transazioni devono rispettare i boundary originali

Ho implementato un sub-parser per SQL embedded che tentava di estrarre e trasformare questi statements, ma le interazioni con il resto del codice COBOL rendevano difficile una trasformazione pulita. La necessità di generare boilerplate code significativo per ogni operazione SQL rendeva il codice risultante verbose e difficile da leggere.

Un problema ancora più fondamentale emergeva a livello di architettura. COBOL programs 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 good design. Ma tentare di refactorizzare automaticamente in strutture più modulari richiedeva comprensione semantica profonda che un parser sintattico non poteva fornire.

La qualità del codice prodotto era un'altra area di preoccupazione critica. Anche quando il parser riusciva a produrre codice Java sintatticamente corretto e funzionalmente equivalente, il risultato era tutt'altro che idiomatco. Il codice generato sembrava «COBOL scritto in Java» - preservava tutti gli idiomi e patterns COBOL invece di adottare convenzioni Java.

Esempi specifici di problemi di qualità includevano:

- Naming conventions che riflettevano COBOL (UPPERCASE-WITH-HYPHENS) invece di Java conventions
- Mancanza di incapsulamento appropriato - tutto pubblico e accessibile
- Assenza di type safety significativo - estensivo uso di strings e primitive types
- Strutture di controllo verbose che potevano essere semplificate
- Mancanza di error handling idiomatco - check di return codes invece di exceptions

A questo punto, dopo tre settimane di sviluppo intensivo, ho condotto una stima realistica del lavoro rimanente. Il parser copriva forse il 25% dei costrutti COBOL necessari per gestire programmi real-world. 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 refactoring del codice esistente.

Ancora più scoraggiante era la realizzazione che completare il parser era solo il primo passo. Il codice generato avrebbe richiesto extensive post-processing per essere veramente utilizzabile. Stimavo che ogni 1000 linee di COBOL avrebbero richiesto giorni di manual refactoring 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 parser-based, sebbene tecnicamente fattibile dato tempo illimitato, non era pratico nei constraints 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 turning point 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 deep insights nelle sfide della migrazione COBOL e creato una solida base di conoscenza su cui costruire approcci più sofisticati.

Terzo periodo: pivot 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 stage. Durante una sessione di review con il tutor aziendale, dove presentavo i progressi e le sfide incontrate con l'approccio parser-based, 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 parser tradizionale - something 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 domain della comprensione e generazione di codice, presentava un'opportunità unica di pioneering un approccio completamente nuovo.

Valutazione delle API di AI generativa

Il passaggio da una pipeline deterministica a un sistema AI-powered ha richiesto innanzitutto una ricognizione approfondita del landscape delle AI generative disponibili. Il timing era fortunato: il 2024 aveva visto un'esplosione di modelli e servizi AI focalizzati specificamente su code understanding e generation.

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 idiomático in linguaggi target
- Comprensione del contesto e della business logic
- Consistenza e affidabilità dei risultati
- API accessibility e documentazione
- Costi e limitazioni di utilizzo
- Possibilità di fine-tuning o customizzazione

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

- Semplici programmi sequenziali per testare la comprensione base
- Strutture dati complesse con REDEFINES e OCCURS
- Logica di controllo con PERFORM statements annidati
- SQL embedded e gestione transazioni
- Pattern comuni nel domain bancario/assicurativo

Ogni AI system è stato valutato sulla sua capacità di comprendere questi snippet 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 idiomática, seguendo convenzioni moderne e best practices.

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 detailed rationale per le scelte di design 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.

Google's Gemini mostrava forza particolare nell'integrazione con ecosistemi esistenti. Le conversioni suggerite spesso includevano riferimenti a librerie e frameworks appropriati, dimostrando conoscenza non solo dei linguaggi ma degli ecosistemi circostanti. La capacità

di suggerire refactoring architetturali, come la decomposizione di monoliti in servizi, era notevole.

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

Un insight cruciale emerso da questi test è stato che la qualità dei risultati dipendeva criticamente da come l'AI veniva interrogata. Un prompt generico «traduci questo COBOL in Java» produceva risultati mediocri. Ma prompt più sofisticati che fornivano contesto, specificavano requirements, e guidavano il processo di ragionamento producevano risultati dramatically migliori.

Ho anche esplorato la possibilità di combinare multiple AI per leveraging i strength di ciascuna. Un workflow dove un'AI analizzava e comprendeva il codice, un'altra generava la conversione, e una terza reviewava e suggeriva miglioramenti mostrava promise. Questo approccio «ensemble» poteva potenzialmente superare le limitazioni di ogni singolo modello.

La valutazione ha anche rivelato limitazioni importanti. Tutti i modelli avevano token limits che potevano essere problematici per programmi COBOL large-scale. La consistenza tra runs successive non era sempre garantita. Nessun modello aveva training specifico su COBOL enterprise-grade, affidandosi invece su qualunque codice COBOL fosse presente nei training data pubblici.

Nonostante queste limitazioni, il potenziale era undeniable. In pochi secondi, questi sistemi potevano produrre conversioni che avrebbero richiesto ore a un developer umano, e con qualità che spesso superava quello che il mio parser deterministico poteva sperare di achieve anche con mesi di sviluppo addizionale.

Design del sistema AI-powered

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

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

Il secondo principio era la generazione idiomatica. Il codice prodotto doveva essere non solo funzionalmente equivalente ma stilisticamente appropriato per il linguaggio target. Questo richiedeva che l'AI comprendesse non solo la sintassi Java ma le convenzioni, i patterns, e le best practices della community.

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

L'architettura high-level che ho progettato rifletteva questi principi:

Il Pre-processing Layer si occupava di preparare il codice COBOL per l'elaborazione AI. Questo includeva:

- Segmentazione intelligente del codice in chunks gestibili
- Estrazione di metadata e contesto
- Identificazione di dipendenze e relazioni tra componenti
- Preparazione di informazioni supplementari (data dictionaries, business rules documentation)

Il Context Building Layer costruiva una comprensione olistica del programma:

- Analisi del purpose generale del programma
- Identificazione di pattern di dominio (batch processing, transactional, reporting)
- Mappatura di relazioni tra data structures e processing logic
- Creazione di un «narrative» del programma per guidare la conversione

Il Conversion Orchestration Layer gestiva l'interazione con l'AI:

- Costruzione di prompt ottimizzati per ogni tipo di conversione
- Gestione di conversazioni multi-turn per refinement
- Coordinazione tra multiple AI per task specializzati
- Handling di token limits attraverso intelligent chunking

Il Validation and Refinement Layer assicurava qualità:

- Syntax validation del codice generato
- Semantic validation attraverso test generation
- Iterative refinement basato su validation results
- Human-in-the-loop capabilities per casi ambigui

Il Post-processing Layer finalizzava l'output:

- Organizzazione del codice in struttura di progetto appropriata
- Generazione di build configuration e dependencies
- Creazione di documentazione comprensiva
- Preparazione di migration reports e notes

Un aspetto chiave del design era la modularità e estensibilità. Ogni layer era progettato come componente indipendente con interfacce ben definite. Questo permetteva:

- Swapping di diversi AI providers senza impattare il resto del sistema
- Aggiunta di nuovi validation o refinement steps
- Customizzazione per specifici domini o requirements
- Evolution indipendente di componenti

Il flusso di dati attraverso il sistema era carefully orchestrato. Un programma COBOL entrava nel sistema e veniva prima analizzato per comprenderne la struttura e il purpose. Questa comprensione guidava la segmentazione - programmi con clear functional boundaries potevano essere divisi accordingly, mentre highly coupled code rimaneva insieme.

Ogni segmento passava attraverso conversion con prompts customizzati basati sul tipo di codice. Data structures usavano prompts che enfatizzavano type safety e validation. Business logic usava prompts che preservavano algorithmic correctness mentre modernizzavano implementation. I/O operations usavano prompts aware di modern persistence patterns.

I risultati di ogni conversion erano validati e refined iterativamente. Validation errors o ambiguità triggeravano conversazioni di chiarimento con l'AI. Il sistema manteneva context tra iterazioni, permettendo refinement progressivo verso il risultato desiderato.

Un innovation particolare era il concetto di «confidence scoring». Il sistema assegnava confidence scores a diverse parti della conversione basato su fattori come:

- Complessità del codice sorgente
- Ambiguità o unusual patterns
- Consistency dei risultati attraverso multiple AI queries
- Validation test results

Low confidence sections erano flaggate per human review, permettendo un approccio pragmatico dove automazione era usata dove affidabile e human expertise dove necessaria.

Il design includeva anche capabilities per learning e improvement. Il sistema loggava tutte le conversioni, validations, e refinements. Questi logs potevano essere analizzati per identificare patterns di successo e failure, informando miglioramenti ai prompts e al processo.

Quarto periodo: implementazione della soluzione AI-driven

Le ultime quattro settimane dello stage sono state dedicate all'implementazione completa del sistema di migrazione basato su AI. Questo periodo è stato caratterizzato da intensa sperimentazione, rapid iteration, e discoveries continue che hanno validato e refined l'approccio AI-driven.

Sviluppo del prompt engineering

Il cuore del sistema AI-driven risiedeva nella qualità e sofisticazione dei prompts utilizzati per guidare l'intelligenza artificiale. Il prompt engineering si è rivelato essere sia un'arte che una scienza, richiedendo deep understanding di come le AI interpretano e rispondono a diverse formulazioni.

Ho iniziato con un approccio sistematico al prompt development, creando un framework strutturato che categorizzava prompts per purpose e ottimizzava ciascuno per il suo specific use case. Il framework includeva:

Base Analysis Prompts: Questi prompts erano progettati per initial understanding del codice COBOL. Invece di semplicemente chiedere «cosa fa questo codice», ho sviluppato prompts multi-faceted che estraevano layers di comprensione.

Un prompt tipico per l'analisi iniziale guidava l'AI attraverso un processo strutturato di comprensione. Chiedeva di identificare il business purpose principale del programma, non solo la sua funzionalità tecnica. Richiedeva l'identificazione di key data structures e le loro relazioni. Sollecitava l'identificazione di external dependencies come file e database. Infine, chiedeva di evidenziare critical business rules o logic che dovevano essere preservate esattamente.

La formulazione specifica dei prompts si è evoluta attraverso iterazione. Initial versions producevano risposte generiche. Raffinamenti successivi aggiungevano specificità e context che guidavano l'AI verso insights più valuable. Per esempio, specificare il probable domain (banking, insurance) aiutava l'AI a riconoscere domain-specific patterns.

Data Structure Conversion Prompts: La conversione di COBOL data structures richiedeva prompts specializzati che comprendevano le unique characteristics di COBOL data definition. Ho sviluppato un template system che adattava prompts basato sul tipo di struttura being converted.

Per hierarchical data structures, i prompts enfatizzavano la preservazione delle relazioni parent-child mentre modernizzavano la rappresentazione. Guidavano l'AI a considerare se flattening fosse appropriato o se nested classes preservassero meglio la semantica. Per structures con REDEFINES, i prompts spiegavano il concept di memory overlay e guidavano verso appropriate Java implementations usando union types o multiple accessors.

Un breakthrough è arrivato quando ho iniziato a includere esempi nei prompts. Non code examples (per evitare biasing), ma pattern examples. Per instance, spiegando che «COBOL

often uses separate fields for year, month, day that should be combined into a modern date object» guidava l'AI verso modernizzazioni sensate.

Business Logic Conversion Prompts: Convertire la PROCEDURE DIVISION richiedeva i prompts più sofisticati. Qui, preservare exact behavior mentre modernizzava implementation era crucial. Ho sviluppato un multi-stage approach dove diversi prompts gestivano diversi aspects della conversione.

Il primo stage focusava su understanding del flow. L'AI era guidata a identificare il main processing flow, loop structures, conditional branches, e exit points. Questo created una «map» della logica che informava subsequent conversion.

Il secondo stage gestiva la actual conversion. Prompts qui erano carefully crafted per bilanciare fidelity all'originale con modernizzazione. Specificavo che mentre il behavior doveva essere identico, l'implementation doveva usare modern Java idioms. Loop structures dovevano usare appropriate Java constructs. Error handling doveva transitare da return codes a exceptions dove sensato.

Un aspetto particolare era handling di COBOL-specific constructs. Per PERFORM statements, sviluppai prompts che spiegavano le varie semantiche e guidavano verso appropriate Java implementations. Per esempio, PERFORM VARYING con multiple variables era mappato su nested loops con clear variable scoping.

SQL Conversion Prompts: L'embedded SQL presentava unique challenges che richiedevano specialized prompts. COBOL embedded SQL ha assumptions e patterns che non traducono directly a JDBC o modern ORMs.

I prompts per SQL conversion includevano context about transaction boundaries, cursor lifecycle, host variable mapping, e error handling patterns. Guidavano l'AI a considerare se raw JDBC, prepared statements, o anche JPA fosse più appropriate basato sul use case.

Iterative Refinement Process: Un key innovation era il development di prompts per iterative refinement. Invece di expecting perfect results in one shot, progettai il sistema per conversazioni multi-turn dove ogni iteration migliorava il risultato.

Refinement prompts erano formulati per essere specific about issues mentre davano all'AI freedom di trovare solutions. Per esempio: «The generated code correctly implements the logic but uses verbose if-else chains. Refactor using Java switch expressions or pattern matching where appropriate, maintaining exact same behavior.»

Questo approccio iterativo mimava come un human developer avrebbe affrontato la conversione - first getting it working, then making it clean and idiomatic.

Prompt Optimization Techniques: Attraverso extensive experimentation, ho discovered several techniques che significativamente miglioravano result quality:

Role specification: Starting prompts con «You are an expert in both COBOL and Java with deep understanding of enterprise modernization...» primed l'AI per il right mindset.

Explicit constraints: Specificare chiaramente cosa preservare (business logic, decimal precision) e cosa modernizzare (syntax, patterns) riduceva ambiguità.

Reasoning encouragement: Chiedere all'AI di «think through» la conversione prima di generare codice produceva risultati più thoughtful.

Quality criteria: Includere explicit criteria per «good» output guidava l'AI oltre mere correctness verso genuine quality.

Documentation e Knowledge Preservation: Un aspetto critico del prompt engineering era ensuring che knowledge embedded nel COBOL non fosse lost. Sviluppai prompts che specificamente istruivano l'AI a:

- Preserve business terminology nei nomi di variabili e metodi (con adaptation a Java conventions)
- Generate comprehensive JavaDoc che spiegava non solo what ma why
- Include comments che mappavano nuovo codice a COBOL originale per traceability
- Create separate documentation di business rules e assumptions

Implementazione del translator completo

Con un robust prompt engineering framework in place, ho proceduto a implementare il complete sistema di traduzione end-to-end. Questa implementazione rappresentava la culminazione di tutto il learning e experimentation delle settimane precedenti.

Il sistema era architettato come una pipeline modulare dove ogni stage aveva responsabilità ben definite ma poteva comunicare context a stages successivi. Questa architettura permetteva sia processing efficiente che capability di backtracking quando refinement era necessario.

Input Processing e Analysis: Il primo componente gestiva l'ingestion di COBOL source files. Invece del complex parsing tentato nell'approccio tradizionale, questo componente faceva minimal strutturale analysis - enough per segmentare intelligentemente il codice senza attempting full understanding.

La segmentazione strategy era sofisticata. Invece di arbitrary line counts o syntactic boundaries, usava heuristics per identificare logical units. Una batch processing section rimaneva

insieme. Related data structures erano grouped. Dependencies erano tracked per ensure che related code non fosse artificialmente separated.

Il componente produceva anche metadata sulla struttura generale del programma - counts di diverse statement types, complexity metrics, identificazione di patterns comuni. Questo metadata informava decisioni downstream su come approcciare la conversione.

Context Assembly: Prima di iniziare actual conversion, il sistema assembled comprehensive context. Questo includeva non solo il codice being converted ma anche:

- Related copybooks e include files
- Data dictionary entries se disponibili
- Business documentation estratta da comments
- Inferred information sul domain e purpose

Questo context era formattato in modo che l'AI potesse facilmente reference durante conversion. Critical information era highlighted. Relationships erano rese explicit. Ambiguità erano flaggate per attention.

Orchestrazione della Conversione: Il cuore del sistema era il conversion orchestrator. Questo componente gestiva le interazioni con AI services, implementando sophisticated flow control e error handling.

Per ogni segment di codice, l'orchestrator:

1. Selezionava appropriate prompt templates basati sul code type
2. Instantiava prompts con specific context
3. Gestiva API calls con retry logic e rate limiting
4. Processava responses e extracted generated code
5. Triggherava validation e refinement se necessario

L'orchestrator manteneva conversational context attraverso multiple interactions. Se l'AI chiedeva clarification o suggeriva alternatives, l'orchestrator poteva gestire multi-turn conversations per reach optimal results.

Un particolare challenge era gestire token limits. Large COBOL programs potevano exceed i limits di single API calls. L'orchestrator implementava intelligent chunking che:

- Divideva il codice a logical boundaries
- Manteneva context overlap tra chunks
- Ricombinava risultati coherently
- Gestiva cross-chunk dependencies

Validation Engine: Post-generation validation era critical per ensuring quality. Il validation engine implementava multiple levels di checking:

Syntactic validation assicurava che generated Java fosse compilabile. Usava il Java compiler API per attempt compilation e parse error messages per feedback all'AI.

Semantic validation era più sophisticated. Il sistema generava test cases basati su understanding del COBOL behavior e verificava che il Java producesse same results. Per programmi con clear input/output patterns, poteva even generare test automatici.

Style validation checked che il codice seguisse Java conventions e best practices. Usava static analysis tools per identificare code smells, excessive complexity, o non-idiomatic patterns.

Refinement Loop: Quando validation identificava issues, il refinement loop engaged. Questo componente costruiva targeted prompts che describevano specific problems e chiedeva fixes.

Il refinement era iterativo ma bounded - dopo un set number di attempts, problematic sections erano flaggate per human review piuttosto che continuing indefinitely. Questo pragmatic approach riconosceva che perfect automation non era sempre achievable o desirable.

Code Assembly e Organization: Una volta che tutti i segments erano convertiti e validated, il sistema assembled il complete Java project. Questo andava beyond semplicemente concatenating files:

- Package structure era derivata analyzing il code organization
- Classes erano separate appropriately basate su cohesion
- Common utilities erano extracted per reduce duplication
- Interfaces erano generated dove multiple implementations existed

Il sistema faceva anche architectural decisions. Un monolithic COBOL program poteva essere split in multiple Java classes basato su functional boundaries. Shared data structures diventavano separate model classes. Business logic era organizzata in service classes con clear responsibilities.

SQL e Database Integration: Handling embedded SQL richiedeva special attention. Il sistema non solo convertiva SQL statements ma modernizzava l'intero approach a database interaction.

COBOL embedded SQL tipicamente usa host variables e explicit cursor management. Il sistema convertiva questi in modern Java database access:

- Simple queries diventavano JDBC prepared statements
- Complex cursor operations erano wrapped in iterator patterns
- Transaction boundaries erano preserved ma implementate usando Java transaction APIs

- Error handling transitava da SQLCODE checking a exception-based approaches

Per progetti con extensive database interaction, il sistema poteva even suggerire JPA entity generation, though questo rimaneva optional dato che richiedeva more extensive restructuring.

Preservazione della Business Logic: Throughout l'intero processo, preservare exact business logic era paramount. Il sistema implementava several mechanisms per ensure questo:

- Comments nel generated code che traced back a original COBOL line numbers
- Preservation di original variable names (adapted a Java conventions) per maintain domain terminology
- Generation di detailed JavaDoc che explained business purpose, not just technical implementation
- Creation di separate business rules documentation extracted dal COBOL

Generazione automatica di progetti Maven

L'ultima componente sviluppata del sistema trasformava l'output da semplice collezione di file Java a progetti enterprise-ready completi. Questa capacità elevava il sistema da tool di conversione a soluzione di modernizzazione comprensiva.

La decisione di generare progetti Maven completi era strategica. Maven rappresenta lo standard de facto per Java enterprise projects, con il suo ecosystem maturo di plugins, dependency management, e build lifecycle. Generare progetti che seguivano Maven conventions significava che l'output poteva essere immediatamente integrato in modern development workflows.

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

La generazione iniziava analizzando il codice convertito per determinare l'organizzazione ottimale. Functional modules naturali nel COBOL diventavano separate Maven modules. Shared utilities erano estratte in common modules. La struttura risultante rifletteva modern microservices principles dove appropriato, mentre manteneva monolithic structure dove la decomposizione non aggiungeva valore.

POM Generation: Il cuore di ogni Maven project è il Project Object Model (POM). Il sistema generava POM files che erano comprehensive e production-ready.

Dependency management era particolarmente sofisticato. Il sistema analizzava il codice generato per identificare required dependencies:

- JDBC drivers basati sul database originale
- Logging frameworks (defaulting to SLF4J con Logback)
- Testing frameworks (JUnit 5, Mockito)
- Utility libraries per funzionalità specifiche

Versioni delle dependencies erano carefully scelte per compatibility e security. Il sistema manteneva una knowledge base di stable version combinations che erano known to work well insieme.

Il POM includeva anche build configuration appropriata:

- Compiler settings per target Java version
- Resource handling per configuration files
- Test execution configuration
- Packaging configuration basata sul deployment target

Build Profiles: Riconoscendo che enterprise applications necessitano different configurations per different environments, il sistema generava Maven profiles per common scenarios:

- Development profile con settings per local testing
- Test profile con in-memory databases e mock services
- Production profile con optimized settings e security configurations

Ogni profile poteva override properties, dependencies, e build behavior appropriato per il suo target environment.

Plugin Configuration: Il sistema configurava automaticamente Maven plugins essenziali:

- Compiler plugin con appropriate source/target versions
- Surefire plugin per test execution con memory settings appropriati per large test suites
- Resources plugin per handling di non-Java files
- Assembly plugin per creating distribution packages

Per progetti con specific needs, plugins addizionali erano configurati. Database-heavy applications ricevevano Flyway o Liquibase per migration management. Web services ricevevano appropriate plugins per generating clients o endpoints.

Documentation Generation: Beyond il codice stesso, il sistema generava comprehensive project documentation:

- README.md con project overview, setup instructions, e migration notes
- MIGRATION_NOTES.md detailing specific decisions made durante conversion
- API documentation se il progetto exposed services
- Architecture decision records per significant structural choices

La documentazione era generata usando l'AI's understanding del codice, producendo documentation che era actually useful piuttosto che boilerplate generica.

Test Structure: Il sistema generava anche una complete test structure. Mentre non poteva generare comprehensive test logic senza additional context, creava:

- Test class skeletons per ogni main class
- Basic test cases per obvious scenarios
- Test data setup basato su data structures
- Integration test structure per database operations

Questo gave developers un starting point per building out proper test coverage.

Configuration Externalization: Following twelve-factor app principles, il sistema externalized tutta la configuration. COBOL programs often hanno hard-coded configuration; il sistema identificava questi e li extractava in:

- application.properties files per Spring-style configuration
- Environment variable mappings per container deployments
- Configuration classes che provided type-safe access

DevOps Readiness: Riconoscendo modern deployment practices, il sistema generava anche DevOps-related artifacts:

- Dockerfile per containerized deployment
- Basic Kubernetes manifests per orchestration
- GitHub Actions workflow per CI/CD
- .gitignore appropriately configured per Java projects

Questi artifacts erano templates che required customization ma provided solid starting points.

Risultati raggiunti

L'implementazione completa del sistema AI-driven 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 compelling:

L'approccio parser-based aveva richiesto tre settimane intensive per implementare supporto per circa il 25% dei costrutti COBOL necessari. Estrapolando, un parser completo avrebbe

richiesto almeno 12 settimane, assumendo complessità lineare (un'assunzione ottimistica data la natura interconnessa dei costrutti COBOL).

Il sistema AI-driven è stato sviluppato e reso fully functional in quattro settimane. Ma questa comparazione sottostima il vero impatto. Il parser avrebbe prodotto solo raw conversion - il post-processing manuale per rendere il codice production-ready avrebbe aggiunto settimane o mesi aggiuntivi. Il sistema AI produceva codice già idiomatico e well-structured.

Velocity di Conversione: Una volta operativo, il sistema dimostrava velocity 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 raw speed, la consistenza era remarkable. Un human developer potrebbe convertire velocemente semplici sezioni ma rallentare significantly su complex logic. L'AI manteneva consistent performance regardless della complessità, con solo modesti incrementi nel tempo per codice più challenging.

Eliminazione del Technical Debt: Un beneficio non immediatamente ovvio era l'eliminazione del technical debt accumulation. Manual conversion, specialmente under time pressure, tende a produrre shortcuts e compromessi. «Lo sistema dopo» diventa permanent technical debt.

L'AI non aveva incentivo a prendere shortcuts. Ogni conversione seguiva best practices, implementava proper error handling, includeva documentation. Il risultato era codice che non solo funzionava ma era maintainable dal day one.

Scalabilità senza Precedenti: Forse il più dramatic impatto era sulla scalabilità. Traditional approaches scale linearmente - double il codice significa double il tempo e le risorse. L'approccio AI-driven scalava sub-linearmente grazie a:

- Pattern recognition che migliorava con più esempi
- Reuso di conversioni simili
- Parallelizzazione naturale di conversioni indipendenti

Questo rendeva feasible la migrazione di intere codebase legacy che sarebbero state economicamente impossibili con approcci manuali.

Democratizzazione della Modernizzazione: L'AI riduceva dramatically la barrier to entry per modernization projects. Non era più necessario un team di esperti COBOL e Java

architects. Un small team con basic understanding poteva guidare il processo, focusing su validation e refinement piuttosto che manual translation.

Questo democratization effect era profound. Organizzazioni che non potevano afford large modernization projects ora avevano un path forward. Il ROI calculation cambiava fundamentalmente quando i costi dropped di un ordine di magnitude.

Analisi qualitativa dei risultati

Beyond le raw metrics, la qualità del output prodotto dal sistema AI-driven meritava deep analysis. Non si trattava solo di produrre codice funzionante, ma di generare software che development teams avrebbero actually voluto mantenere ed evolvere.

Gestione delle Divisioni COBOL: Il sistema dimostrava sophisticated understanding di come ogni COBOL division dovesse essere trasformata per modern contexts:

La IDENTIFICATION DIVISION non era semplicemente scartata ma trasformata in valuable metadata. Program information diventava parte della documentazione, author information era preservata in version control-friendly formats, e program purpose era catturata in class-level documentation che actually aiutava developers a comprendere il sistema.

La ENVIRONMENT DIVISION transformation era particularly clever. Invece di hard-coding environmental dependencies, il sistema generava clean abstraction layers. File associations diventavano configuration properties. System dependencies erano isolated in dedicated configuration classes. Il risultato era codice veramente portable che seguiva twelve-factor principles.

La DATA DIVISION transformation mostrava il real power dell'AI understanding. Complex hierarchical structures erano analizzate per il loro true purpose e convertite appropriately:

- Simple structures diventavano POJOs con appropriate validation
- Complex nested structures erano refactored in proper object models
- Reused structures erano extracted in shared classes
- Type safety era aggiunta dove il COBOL era permissivo

La PROCEDURE DIVISION transformation era dove l'AI brillava di più. Invece di mechanical translation, il sistema comprendeva il flow e il purpose, producendo codice che era recognizably Java:

- Procedural sequences diventavano well-structured methods
- Complex conditions erano simplified usando modern constructs
- Loops erano converted nei most appropriate Java iterations
- Error handling transitava da procedural a exception-based

Code Quality Metrics: Analizzando il codice prodotto attraverso standard quality metrics rivelava risultati impressionanti:

Cyclomatic complexity era consistently ridotta del 20-40% rispetto all'originale COBOL. Questo non era accidentale - l'AI naturally refactored complex conditionals in cleaner structures.

Code duplication era virtually eliminata. Dove COBOL programs spesso hanno copied-and-modified sections, l'AI recognized patterns e extracted common functionality.

Method lengths erano appropriate - l'AI naturally decomposed long procedural sequences in focused methods che followed single responsibility principle.

Naming conventions erano consistent e meaningful. L'AI preserved business terminology mentre adattava a Java conventions, risultando in codice che domain experts potevano ancora riconoscere.

Idiomatic Java Generation: Il codice generato non sembrava «COBOL in Java syntax» ma genuine Java:

- Appropriate uso di collections invece di arrays
- Builder patterns per complex object construction
- Streams per data processing dove sensato
- Optional per handling di possibili null values
- Enums per finite value sets invece di constants

Questo idiomatic code era crucial per long-term maintenance. Developers non dovevano imparare «COBOL patterns» per lavorare con il codice - potevano applicare normal Java expertise.

Documentation Excellence: La documentazione generata era un particolare strength:

- Class-level documentation spiegava business purpose, non solo technical details
- Method documentation includeva preconditions, postconditions, e business rules
- Inline comments erano usati judiciously per explain non-obvious logic
- Mapping comments connected Java code back a original COBOL per traceability

La documentazione non era generic o boilerplate ma genuinely informative, catturando knowledge che altrimenti sarebbe stata persa nella migration.

Architectural Modernization: Beyond line-by-line conversion, il sistema faceva thoughtful architectural improvements:

- Monolithic programs erano decomposed in logical components
- Clear separation of concerns era introduced
- Dependency injection patterns erano suggeriti dove appropriate

- Database access era isolated in repository layers
- Business logic era separated da infrastructure concerns

Questi improvements non erano arbitrary ma basati su analysis del code structure e dependencies.

Risultati quantitativi

I concrete deliverables del progetto fornivano tangible evidence del successo dell'approccio AI-driven. Numbers e metrics raccontavano una compelling story di achievement oltre le initial expectations.

Portfolio di Conversioni Complete: Durante le final weeks del progetto, ho successfully convertito tre complete COBOL programs di increasing complexity:

Il primo progetto era un Interest Calculator di 450 lines. Questo seemingly simple program nascondeva surprising complexity in decimal precision handling e compound interest calculations. La conversione produceva 380 lines di clean Java che:

- Preservava exact decimal precision usando BigDecimal appropriately
- Refactored repetitive calculations in reusable methods
- Added proper validation per interest rates e principals
- Included comprehensive unit tests validating calculations

Il secondo progetto, un Inventory Management System di 890 lines, presentava challenges in data structure complexity e file handling. La conversione risultava in 750 lines di well-structured Java che:

- Transformed flat file structures in proper domain objects
- Implemented repository pattern per data access
- Added transaction support per consistency
- Created service layer con clear business operations
- Generated RESTful API endpoints per modern integration

Il terzo e most ambitious progetto era un Batch Report Generator di 1200 lines. Questo mostrava typical COBOL batch processing patterns con complex data aggregation e formatting. La conversione produceva 950 lines di modern Java che:

- Leveraged Stream API per efficient data processing
- Implemented template-based report generation
- Added scheduling capabilities usando Spring Batch concepts
- Created configurable report definitions
- Included performance optimizations per large data volumes

Coverage Metrics: Analizzando la completezza delle conversioni rivelava strong results:

Functional coverage raggiungeva 95% - ogni major business function era successfully migrated. Il remaining 5% erano primarily deprecated features o platform-specific functions che non avevano senso in modern context.

Syntactic coverage era 88% - la vast majority di COBOL constructs erano handled automatically. Unhandled constructs erano primarily esoteric o deprecated features rarely used in production code.

Test coverage del converted code averaged 80%, con critical business logic reaching 100%. L'AI generava test structure e basic test cases che developers potevano extend.

Performance Comparisons: Benchmarking converted applications contro originals mostrava interessanti results:

- CPU usage generally decreased 15-30% grazie a more efficient algorithms
- Memory usage aumentava moderately (Java overhead) ma rimaneva well within modern hardware capacities
- I/O operations erano significantly faster grazie a modern buffering e caching
- Overall throughput migliorava 20-50% in typical scenarios

Code Quality Validation: Running industry-standard analysis tools sul generated code produceva excellent scores:

- SonarQube reported zero critical issues, minimal major issues
- CheckStyle violations erano minimal e mostly preference-based
- PMD static analysis trovava no significant problems
- Security scanning identified no vulnerabilities

Migration Completeness: Ogni progetto convertito era truly «production-ready»:

- Fully compilable senza errors o warnings
- Runnable con identical functional behavior all'originale
- Deployable usando standard Java deployment methods
- Maintainable da Java developers senza COBOL knowledge
- Documented sufficiently per understanding e evolution

Il totale di oltre 2000 lines di production-quality Java rappresentava non solo una proof of concept ma una genuine demonstration che AI-driven migration era practical, effective, e ready per real-world application. Il sistema aveva transformed quello che tradizionalmente era un multi-month manual effort in un processo che poteva essere completato in giorni con superior results.

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>