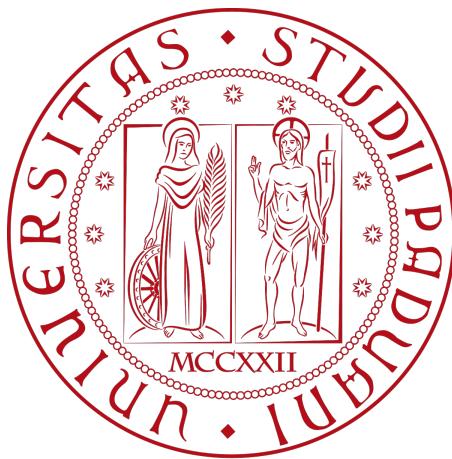


Università degli Studi di Padova

Dipartimento di Matematica «Tullio Levi-Civita»

Corso di Laurea in Informatica



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

Tesi di Laurea

Relatore

Prof. Tullio Vardanega

Laureando

Annalisa Egidi

Matricola: 1216745

Ringraziamenti

Sommario

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

Nel dominio applicativo di interesse dell'elaborato:

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

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

Struttura del testo

Il corpo principale della relazione è suddiviso in 4 capitoli:

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

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

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

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

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

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

Indice

1 Miriade: un ecosistema di innovazione tecnologica	1
1.1 L’azienda nel panorama informatico e sociale	1
1.2 Metodologie e tecnologie all’avanguardia	1
1.3 Architettura organizzativa	3
1.4 Investimento nel capitale umano e nella ricerca	5
2 Il progetto di migrazione <i>COBOL-Java</i>	8
2.1 Contesto di attualità	8
2.2 Obiettivi dello stage	10
2.2.1 Obiettivi principali	10
2.2.2 Obiettivi operativi	10
2.2.3 Metriche di successo	11
2.3 Vincoli	12
2.4 Pianificazione concordata	12
2.5 Valore strategico per l’azienda	13
2.6 Aspettative personali	14
3 Sviluppo del progetto: dal <i>parser</i> tradizionale all’ <i>AI</i>	16
3.1 <i>setup</i> iniziale e metodologia di lavoro	16
3.2 Primo periodo: immersione nel mondo COBOL	16
3.2.1 Studio del linguaggio e creazione progetti test	16
3.2.2 Mappatura dei pattern e analisi di traducibilità	19
3.2.3 Valutazione delle soluzioni esistenti	21
3.3 Secondo periodo: sviluppo del <i>parser</i> tradizionale	24
3.3.1 Implementazione del <i>parser</i> Java	24
3.3.2 Analisi critica e limiti dell’approccio	27
3.4 Terzo periodo: pivot verso l’intelligenza artificiale	28
3.4.1 Valutazione delle API di <i>AI</i> generativa	28
3.4.2 <i>Design</i> del sistema <i>AI-powered</i>	29
3.5 Quarto periodo: implementazione della soluzione <i>AI-driven</i>	31
3.5.1 Sviluppo del <i>parsing</i> engineering	32
3.5.2 Implementazione del translator completo	34
3.5.3 Generazione automatica di progetti <i>Maven</i>	36
3.6 Risultati raggiunti	38

3.6.1	Impatto dell' <i>AI</i> sui tempi di sviluppo	38
3.6.2	Analisi qualitativa dei risultati	39
3.6.3	Risultati quantitativi	41
4	Valutazioni retrospettive e prospettive future	43
4.1	Analisi retrospettiva del percorso	43
4.2	L'AI come <i>game changer</i> nella modernizzazione <i>software</i>	43
4.3	Crescita professionale e competenze acquisite	43
4.4	Valore della formazione universitaria nell'era dell'AI	43
4.5	<i>Roadmap</i> evolutiva e opportunità di sviluppo	43
5	Lista degli acronimi	44
6	Glossario	45
7	Sitografia	46

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
Figura 10	ProLeap COBOL parser	21
Figura 11	IBM WatsonX Code Assistant for Z	22
Figura 12	Chatbot di IBM WatsonX Code Assistant for Z	23
Figura 13	Pipeline del parser COBOL → Java	24
Figura 14	Rappresentazione schematica del sistema di migrazione AI-driven	31
Figura 15	Esecuzione del sistema di gestione conti correnti bancari convertito in Java ..	38
Figura 16	Output nel terminale del sistema di migrazione AI-driven	40
Figura 17	Progetto Maven generato automaticamente dal sistema di migrazione	42

Elenco delle tabelle

Tabella 1	Mappatura dei tipi di dati primitivi COBOL-Java	19
Tabella 2	Mappatura delle strutture di controllo e operazioni	19
Tabella 3	Mappatura delle strutture dati gerarchiche	20
Tabella 4	Pattern di trasformazione per costrutti complessi	20
Tabella 5	Soluzioni architetturali per costrutti senza equivalenti diretti	20
Tabella 6	Mappatura delle PICTURE clauses COBOL verso tipi Java	27

Elenco dei listati

Listato 1	Esempio di IDENTIFICATION DIVISION con metadati	17
Listato 2	Esempio di ENVIRONMENT DIVISION con specifiche hardware	17
Listato 3	Esempio di FILE-CONTROL con gestione esplicita dei file	17
Listato 4	Esempio di gestione degli SQLCODE in COBOL	19
Listato 5	Trasformazione della IDENTIFICATION DIVISION in JavaDoc	25
Listato 6	Esempio di dichiarazione file COBOL	26
Listato 7	Struttura dati COBOL per la gestione dei movimenti bancari	27
Listato 8	Esempio di prompt per la traduzione COBOL → Java	33
Listato 9	Confronto tra sintassi COBOL e Java per operazioni aritmetiche	35
Listato 10	Trasformazione di SECTION e PARAGRAPH COBOL in metodi Java	35
Listato 11	Esempio di pom.xml generato automaticamente	37

1 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*.

1.1 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 curricolari che permettono una crescita personale durante il percorso di studi.

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

1.2 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*.

The screenshot shows the Jira Kanban board for a 'Marketing' project. The left sidebar contains navigation links for Planning, Marketing, Roadmap, Kanban board, Reports, Issues, Components, Development, Code, Releases, Project pages, ScriptRunner Enhancements, Git Commits, Zephyr Squad, Add shortcut, and Project settings. The main area displays a Kanban board with columns: 'ZU ERLEDIGEN 16', 'IN ARBEIT 8', and 'FERTIG 6'. Tasks include 'Resolve bugs' (due MAR-27), 'Define software requirements' (due MAR-22), and 'Setup testing environment' (due MAR-11). Below these are sections for 'Beschleunigen 4 issues' and 'Alle anderen 26 issues'. A legend indicates issue types: Bug (red), Story (blue), Task (green), Epic (orange), and User Story (purple). Dates are listed next to each task, such as MAR-4, MAR-8, MAR-13, etc. A 'Search' bar and various filter options are at the top right.

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 Artificial Intelligence (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.

1.3 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.

1.4 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.

				
PLAY	PASSIONE	UNCONVENTIONAL	INTEGRITÀ	DISPONIBILITÀ
Risolviamo problemi. Inventiamo giochi nuovi e cambiamo le regole di quelli vecchi.	Arriviamo sempre alla soluzione perché siamo innamorati di quello che facciamo.	Il nostro approccio è informale, ma le nostre soluzioni strizzano l'occhio all'eleganza.	Per noi l'onestà sta davanti a ogni interesse. Lavoriamo con serietà e integrità.	Siamo aperti al cambiamento e a chi ci chiede di camminare insieme.

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.

2 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*.

2.1 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 ===> CSR
=COLS> -----+-----+-----+-----+-----+-----+-----+-----+
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:

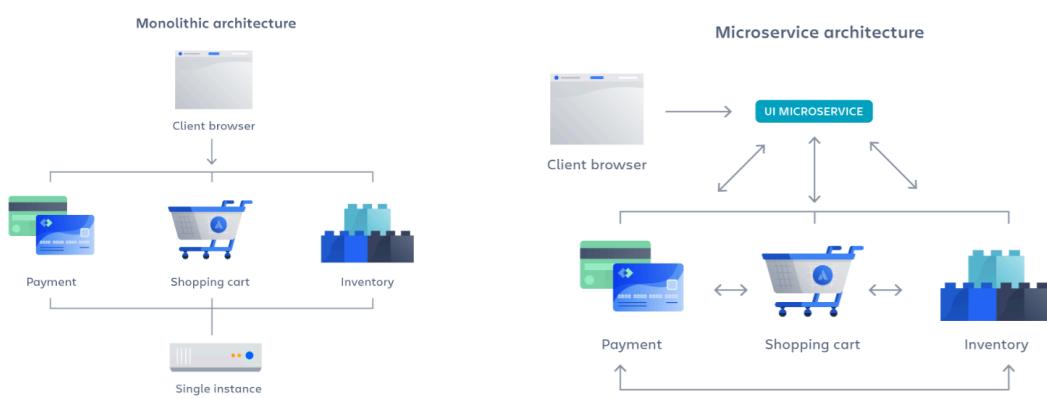
- La crescente difficoltà nel reperire sviluppatori COBOL qualificati [1]
- L'integrazione sempre più complessa con tecnologie moderne [2]
- I rischi operativi derivanti dall'utilizzo di piattaforme *hardware e software* che i *vendor* non supportano più attivamente [3]

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 architettoniche 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.

2.2 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* idiomatico e manutenibile.

2.2.1 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.

2.2.2 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

2.2.3 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

2.3 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

2.4 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 *databases* 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

2.5 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.

2.6 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

3 Sviluppo del progetto: dal *parser* tradizionale all'AI

Il percorso di sviluppo del progetto di migrazione COBOL-Java ha attraversato diverse fasi evolutive, caratterizzate da sfide tecnologiche e cambiamenti strategici. Questo capitolo analizza cronologicamente le fasi del progetto, dall'immersione nel linguaggio COBOL fino all'implementazione di una soluzione basata sull'intelligenza artificiale generativa.

3.1 *setup* iniziale e metodologia di lavoro

Il progetto ha adottato la metodologia *Agile* già consolidata in Miriade, descritta nella [Sezione 1.2](#), strutturando il lavoro in *sprint* settimanali con *stand-up* giornalieri alle 9:05. Come esposto nel capitolo precedente, l'azienda utilizza Jira per la gestione delle attività e Confluence per la documentazione condivisa, strumenti fondamentali per tracciare l'evoluzione del progetto di migrazione.

L'ambiente di sviluppo è stato configurato seguendo gli standard aziendali: Git con BitBucket per il versionamento del codice, con *branch* dedicati per ogni fase sperimentale del progetto. La documentazione progressiva su Confluence ha permesso di mantenere traccia delle decisioni architetturali e delle problematiche incontrate, facilitando le sessioni di *review* settimanali con la tutor aziendale Arianna Bellino.

3.2 Primo periodo: immersione nel mondo COBOL

Le prime due settimane del progetto sono state dedicate ad un'immersione completa nel linguaggio COBOL, dai paradigmi di programmazione moderni a cui ero abituata - con la loro enfasi su astrazione, modularità e riusabilità - a un approccio procedurale strutturato degli anni "60. 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.

3.2.1 Studio del linguaggio e creazione progetti test

Lo studio ha combinato manuali IBM degli anni "80 con tutorial moderni. La collaborazione con l'analista COBOL Luisa Biagi ha fornito contesto pratico sull'utilizzo in ambienti di produzione.

- La IDENTIFICATION DIVISION testimonia l'enfasi sulla documentazione incorporata nel codice, con campi dedicati per autore, data di installazione e osservazioni, riflettendo

un'epoca in cui il codice sorgente costituiva spesso l'unica documentazione disponibile. Un esempio della divisione IDENTIFICATION DIVISION è mostrato in [Listato 1](#).

```
IDENTIFICATION DIVISION.  
PROGRAM-ID. GESTIONE-CONTI.  
AUTHOR. ANNALISA EGIDI.  
INSTALLATION. MIRIADE SRL.  
DATE-WRITTEN. 2025-05-15.  
DATE-COMPILED. 2025-05-20.  
REMARKS. Sistema bancario per gestione conti correnti.
```

Listato 1: Esempio di IDENTIFICATION DIVISION con metadati

- La ENVIRONMENT DIVISION introduce esplicitamente considerazioni hardware e di sistema operativo nel codice sorgente. La necessità di specificare SOURCE-COMPUTER e OBJECT-COMPUTER evidenzia le sfide dell'era dei mainframe, dove la portabilità del software non poteva essere data per scontata. Un esempio della divisione ENVIRONMENT DIVISION è mostrato in [Listato 2](#).

```
ENVIRONMENT DIVISION.  
CONFIGURATION SECTION.  
SOURCE-COMPUTER. IBM-3090.  
OBJECT-COMPUTER. IBM-3090.  
SPECIAL-NAMES.  
DECIMAL-POINT IS COMMA.
```

Listato 2: Esempio di ENVIRONMENT DIVISION con specifiche hardware

- La sezione FILE-CONTROL, con la sua gestione esplicita dell'associazione tra file logici e fisici, richiede un cambio di mentalità significativo rispetto all'astrazione automatica fornita dai moderni sistemi operativi e strutture software. Un esempio della sezione FILE-CONTROL è mostrato in [Listato 3](#).

```
INPUT-OUTPUT SECTION.  
FILE-CONTROL.  
    SELECT CONTI-FILE  
        ASSIGN TO "CONTI.DAT"  
        ORGANIZATION IS INDEXED  
        ACCESS MODE IS RANDOM  
        RECORD KEY IS CONTO-ID.
```

Listato 3: Esempio di FILE-CONTROL con gestione esplicita dei file

Ho sviluppato, progressivamente, tre codici applicativi di complessità crescente:

- Il primo progetto, un sistema di gestione conti correnti bancari, implementa le operazioni fondamentali del banking: apertura conti, depositi, prelievi ed estratti conto. Questo sistema sfrutta la forza di COBOL nell’aritmetica decimale precisa per gestire saldi, fidi e transazioni finanziarie, interfacciandosi con un database PostgreSQL tramite SQL embedded per garantire l’integrità transazionale e la persistenza dei dati.
- Il secondo progetto, un sistema di gestione paghe e stipendi, aumenta significativamente la complessità introducendo calcoli multi-livello per IRPEF con scaglioni progressivi, trattenute previdenziali, detrazioni e addizionali. Il sistema gestisce presenze, straordinari e genera cedolini dettagliati, richiedendo la coordinazione tra molteplici tabelle correlate e l’implementazione di logiche di business complesse per il calcolo delle retribuzioni secondo la normativa fiscale italiana.
- Il terzo progetto, un sistema di gestione magazzino e inventario, rappresenta il culmine della complessità con funzionalità avanzate come la valorizzazione FIFO/LIFO/costo medio ponderato, gestione lotti, analisi ABC degli articoli, controllo scorte con punti di riordino automatici e gestione completa del ciclo ordini fornitori. Il sistema utilizza cursori SQL multipli, transazioni annidate e genera report sofisticati per l’inventario fisico e l’analisi del valore di magazzino, dimostrando la capacità di COBOL di gestire processi aziendali articolati con elevati volumi di dati.

L’interfacciamento con database relazionali ha rappresentato una sfida particolare. Lavorando con PostgreSQL e DB2, ho approfondito le peculiarità dell’SQL embedded in COBOL. L’approccio differisce radicalmente dalle moderne Application Programming Interface (API) Java Database Connectivity (JDBC): il preprocessore COBOL-SQL analizza il codice sorgente, estrae le istruzioni SQL delimitate da EXEC SQL … END-EXEC, e genera il codice COBOL appropriato per l’interazione con il database. La gestione delle variabili host e controllo esplicito degli errori attraverso SQLCODE e SQLSTATE implementa la richiesta del controllo esplicito del codice di ritorno dopo ogni operazione SQL, come illustrato nel [Listato 4](#). Inoltre, per ogni colonna del database è richiesta la corrispettiva variabile locale che funziona da ponte tra il programma e il database, queste variabili sono gestite attraverso la dichiarazione delle stesse nella WORKING-STORAGE SECTION e devono essere necessariamente compatibili per tipo e dimensione con la colonna del database corrispondente.

```

EXEC SQL
    SELECT SALDO INTO :WS-SALDO
    FROM CONTI
    WHERE ID_CLIENTE = :WS-ID-CLIENTE
END-EXEC.

EVALUATE SQLCODE
    WHEN 0      CONTINUE
    WHEN 100    DISPLAY "Cliente non trovato"
    WHEN OTHER  PERFORM ERRORE-DATABASE
END-EVALUATE.

```

Listato 4: Esempio di gestione degli SQLCODE in COBOL

3.2.2 Mappatura dei pattern e analisi di traducibilità

Ho classificato tre categorie di pattern: con equivalenza diretta in Java, che richiedono trasformazioni, e costrutti problematici senza equivalenti.

3.2.2.1 Pattern di equivalenza diretta e costrutti base

Tipo COBOL	Tipo Java
PIC 9(n)	int/long
PIC X(n)	String
PIC 9(n)V9(n)	BigDecimal
PIC S9(n) COMP-3	BigDecimal

Tabella 1: Mappatura dei tipi di dati primitivi COBOL-Java

Costrutto COBOL	Equivalenti Java
IF-THEN-ELSE	if-else
EVALUATE	switch
ADD/SUBTRACT	Operatori aritmetici (+, -)
MULTIPLY/DIVIDE	Operatori aritmetici (*, /)
PERFORM UNTIL	while loop
PERFORM VARYING	for loop
PERFORM THRU	Metodi con sequenza di chiamate

Tabella 2: Mappatura delle strutture di controllo e operazioni

Struttura COBOL	Trasformazione Java
01 level declaration	Classe Java principale
02-49 level numbers	Campi della classe o inner classes
Record gerarchici	Classi Java annidate
WORKING-STORAGE items	Variabili di istanza private

Tabella 3: Mappatura delle strutture dati gerarchiche

Le tabelle [Tabella 1](#), [Tabella 2](#) e [Tabella 3](#) evidenziano pattern di equivalenza diretta che coprono la maggior parte dei costrutti COBOL di base.

3.2.2.2 Trasformazioni complesse e pattern di adattamento

Costrutto COBOL	Strategia di conversione Java
GOTO	Ristrutturazione del flusso con pattern State/Strategy
COPY statements	Classi Java dedicate con import/package
WORKING-STORAGE SECTION	Classi di storage con variabili statiche o di istanza
SECTION/PARAGRAPH	Metodi Java organizzati gerarchicamente
SQL embedded (EXEC SQL)	JDBC con PreparedStatement e gestione transazioni
REDEFINES	Union types tramite ereditarietà o interfacce
OCCURS DEPENDING ON	Collections dinamiche (ArrayList, HashMap)

Tabella 4: Pattern di trasformazione per costrutti complessi

La [Tabella 4](#) mostra pattern che richiedono analisi contestuale per determinare la strategia ottimale.

3.2.2.3 Costrutti problematici e soluzioni architetturali

Costrutto COBOL	Soluzione architetturale Java
ALTER statement	Pattern Strategy con selezione dinamica del comportamento
UNSTRING	String.split() con logica custom per overflow e contatori
EXAMINE	Pattern Matcher con espressioni regolari
NEXT SENTENCE	Controllo di flusso ristrutturato con flag booleani
SORT/MERGE files	Collections.sort() o Stream API con Comparator
Report Writer	Template engine (Jasper, Velocity) o generazione PDF
Screen Section	Framework GUI (Swing/JavaFX) o web UI

Tabella 5: Soluzioni architetturali per costrutti senza equivalenti diretti

La Tabella 5 evidenzia costrutti che richiedono reinterpretazione semantica completa.

3.2.3 Valutazione delle soluzioni esistenti

Soluzioni open-source

L'analisi del ProLeap COBOL *parser*, in Figura 10, uno dei progetti open-source più maturi nello spazio di GitHub, ha rivelato un'architettura solida basata su ANother Tool for Language Recognition (ANTLR)4 con capacità complete di analisi sintattica. Tuttavia, il sistema si limitava alla generazione dell'Abstract Syntax Tree (AST), richiedendo l'implementazione separata della trasformazione AST COBOL → AST Java e della successiva generazione del codice. Ad ogni modo il ProLeap *parser* presentava una architettura modulare che permetteva l'estensione per nuovi costrutti.

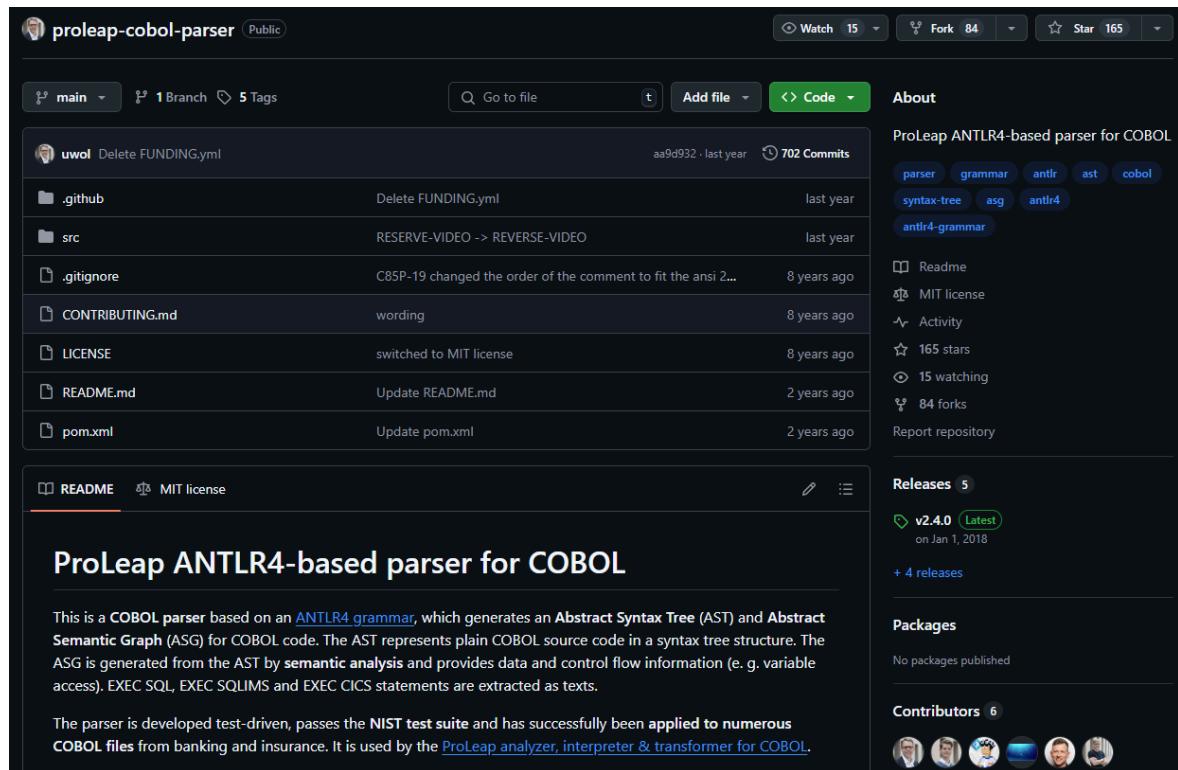


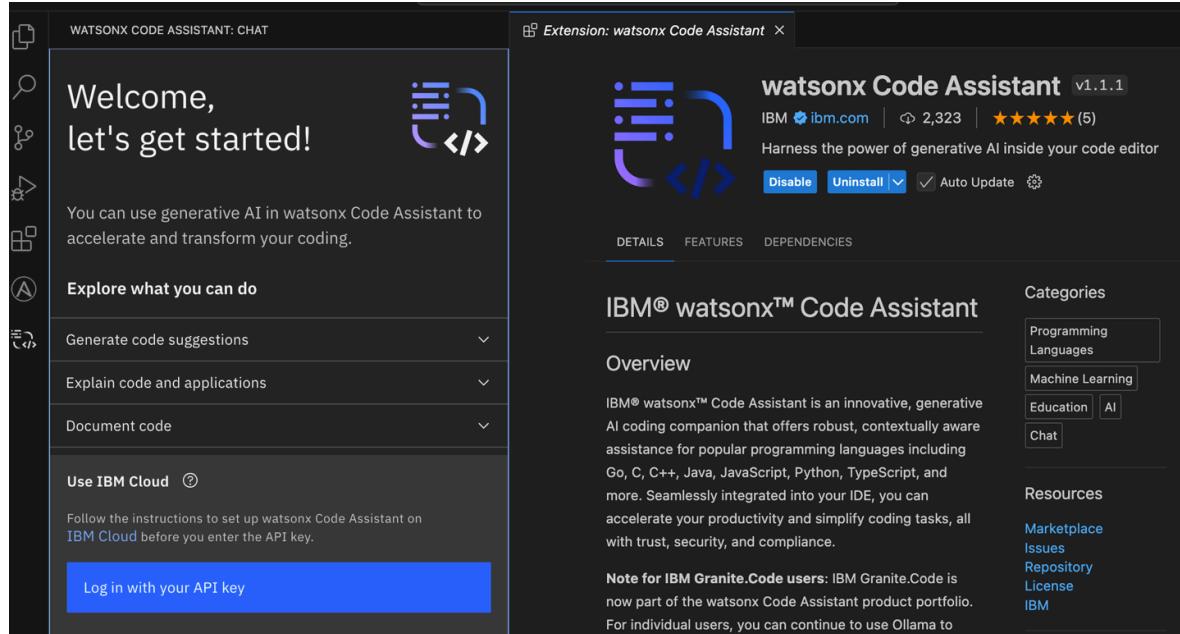
Figura 10: ProLeap COBOL parser

Soluzioni enterprise

Il panorama presentava soluzioni commerciali sofisticate con prezzi corrispondentemente elevati. Essendo soluzioni chiuse al pubblico ho avuto modo di testare tali soluzioni solo in modo limitato, quando prove gratuite o soluzioni demo lo permettevano.

In particolare, grazie alle soluzioni di prova per sviluppatori, ho potuto mettere mano a IBM WatsonX Code Assistant for Z, estensione mostrata in Figura 11, che rappresenta attual-

mente lo stato dell'arte nell'applicazione dell'intelligenza artificiale alla modernizzazione legacy. La soluzione IBM non si limita alla traduzione sintattica ma tenta di comprendere il contesto aziendale del codice tramite l'interazione e interconnessione con l'intelligenza artificiale.



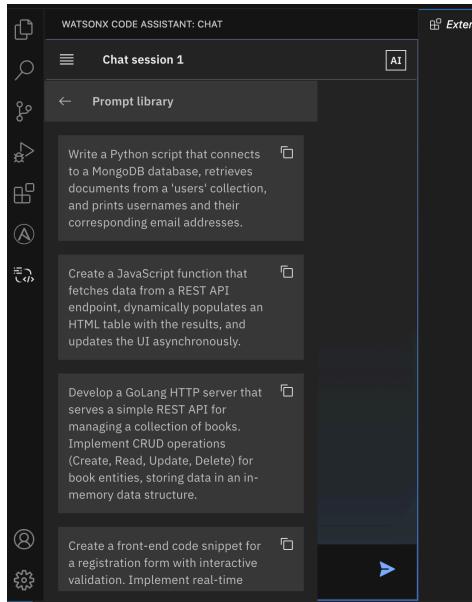
Fonte: <https://www.community.ibm.com/>

Figura 11: IBM WatsonX Code Assistant for Z

Durante la demo, la quale si presentava come chatbot integrato nell'IDE di riferimento (nel mio caso Visual Studio Code), una rappresentazione visiva in [Figura 12](#), ho osservato come il sistema potesse riconoscere schemi di dominio, desse suggerimenti di modernizzazione architettonale (conversione a microservizi dove appropriato), generazione di test automatici basati sulla comprensione del comportamento atteso, e documentazione automatica che catturava l'intento aziendale tramite domande di contesto oltre che puramente tecniche.

La soluzione presentava barriere significative per un progetto di ricerca:

- costo proibitivo per licenze, anche di sviluppo
- natura proprietaria che impediva personalizzazione profonde
- requisiti di infrastruttura aziendale
- vincolo con l'ecosistema IBM



Fonte: <https://community.ibm.com/>

Figura 12: Chatbot di IBM WatsonX Code Assistant for Z

Altre soluzioni enterprise che ho preso in analisi erano Micro Focus, Modern Systems e TSRI che risultavano però meno di valore per qualità-costo dei contenuti producibili, pertanto ho approfondito meno queste tecnologie.

Alla luce di tutte le ricerche, sia open-source che enterprise, ho potuto constatare un vuoto significativo sul tema nel mercato. Le soluzioni open-source, pur essendo accessibili, mancavano di sofisticazione e producevano risultati incompleti che richiedevano estese implementazioni e modifiche manuali. Le soluzioni aziendali, pur essendo potenti, erano inaccessibili per la maggior parte delle organizzazioni a causa dei costi¹.

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

Inoltre, l'analisi delle soluzioni esistenti mi ha fatto notare schemi comuni:

- le migrazioni che preservavano la logica di business erano le più richieste
- l'importanza della documentazione e della tracciabilità nel processo di migrazione
- la necessità di un approccio interattivo col personale umano per una validazione iterativa del codice prodotto
- il valore di preservare la conoscenza di dominio incorporata nel codice legacy

¹ I costi delle soluzioni enterprise variano da €100.000 a €1.000.000 per licenza annuale, a seconda della dimensione del deployment

Queste riflessioni hanno formato significativamente gli approcci che avrei adottato nelle fasi successive del progetto, suggerendo che una soluzione efficace avrebbe dovuto combinare «automatizzazione intelligente» con comprensione semantica profonda, preservazione della logica aziendale con modernizzazione dell’implementazione e accessibilità.

3.3 Secondo periodo: sviluppo del *parser tradizionale*

Ho intrapreso lo sviluppo di un analizzatore sintattico ibrido che combinasse ProLeap con l’implementazione autonoma di traduzione AST e generazione codice.

3.3.1 Implementazione del *parser Java*

L’obiettivo era implementare un sistema modulare ed estensibile che potesse crescere incrementalmente man mano che nuovi costrutti COBOL venivano supportati.

L’architettura seguiva il classico modello di compilatore a pipeline, la sua raffigurazione sequenziale è mostrata in [Figura 13](#):

1. Il **Lexer di ProLeap** gestiva la tokenizzazione del codice COBOL, affrontando le peculiarità del linguaggio come la sensibilità alla colonna (le colonne 1-6 riservate per numeri di linea, colonna 7 per indicatori speciali, colonne 8-72 per il codice effettivo), la gestione dei commenti e delle linee di continuazione
2. Il **Parser di ProLeap** costruiva l’albero sintattico astratto (AST) basandosi sulla grammatica COBOL definita in ANTLR4
3. Il **Traduttore custom** (da implementare) trasformava l’AST COBOL nell’AST Java corrispondente, gestendo le differenze semantiche tra i due linguaggi
4. Il **Generatore di codice** (da implementare) convertiva l’AST Java annotato in codice sorgente Java idiomatico

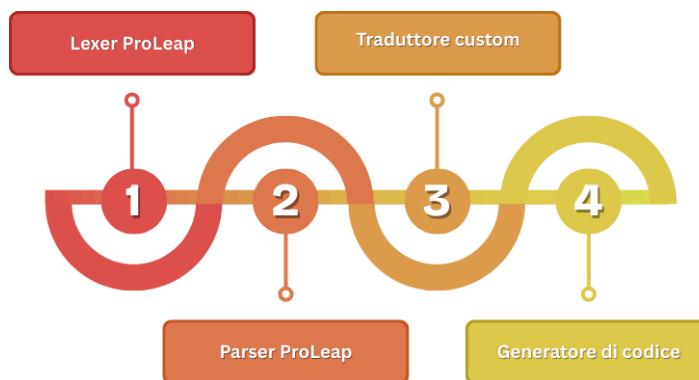


Figura 13: Pipeline del parser COBOL → Java

L’implementazione è proceduta per divisioni:

- **IDENTIFICATION DIVISION**, punto di partenza naturale per la sua semplicità strutturale e prevedibilità. Questa divisione, contenendo principalmente metadati senza impatto diretto sulla logica del programma, forniva contesto essenziale per la comprensione del sistema. Ho sviluppato un analizzatore basato su pattern matching che estraeva sistematicamente le informazioni standard: PROGRAM-ID, AUTHOR, INSTALLATION, DATE-WRITTEN e REMARKS. La strategia di conversione per questa divisione prevedeva una trasformazione semanticamente ricca dei metadati:
 - Il PROGRAM-ID veniva convertito nel nome della classe Java principale, applicando le convenzioni di denominazione Java (trasformazione da KEBAB-CASE o UNDERSCORE_CASE a CamelCase)
 - Le informazioni di contesto (AUTHOR, INSTALLATION, DATE-WRITTEN) venivano preservate in un blocco JavaDoc strutturato in testa alla classe, mantenendo la completa tracciabilità con il programma originale e rispettando gli standard di documentazione Java

```

IDENTIFICATION DIVISION.
PROGRAM-ID. GESTIONE-CONTI.
AUTHOR. ANNALISA EGIDI.
DATE-WRITTEN. 2025-05-15.
```

```

/**
 * GESTIONE-CONTI
 * Author: ANNALISA EGIDI
 * Date Written: 2025-05-15
 *
 * Converted from COBOL on: 2025-06-20
 */
public class GestioneConti {
    // ...
}
```

Listato 5: Trasformazione della IDENTIFICATION DIVISION in JavaDoc

- La **ENVIRONMENT DIVISION** ha presentato le prime sfide sostanziali. Questa divisione, che specifica l'ambiente di esecuzione del programma COBOL includendo informazioni su hardware, sistema operativo e mappatura dei file, riflette un'epoca in cui tali dettagli erano critici per l'esecuzione. Nel contesto moderno, molte di queste informazioni risultano obsolete o vengono gestite attraverso meccanismi completamente

diversi. Ho sviluppato una strategia di conversione che preservava le informazioni semanticamente rilevanti mentre scartava quelle puramente storiche:

- La CONFIGURATION SECTION, contenente SOURCE-COMPUTER e OBJECT-COMPUTER, veniva convertita in commenti documentativi strutturati, preservando l'informazione per riferimento storico senza impatto sul codice generato
- Nella INPUT-OUTPUT SECTION, particolarmente la FILE-CONTROL, ogni dichiarazione di file in COBOL include non solo il nome logico del file ma anche dettagli critici sulla sua organizzazione (sequenziale, indicizzata, relativa), modalità di accesso (sequenziale, random, dinamica), e strategie di gestione degli errori.

Ho implementato un sistema di mapping che traduceva le dichiarazioni COBOL in configurazioni Java moderne. Ad esempio, una dichiarazione COBOL come in [Listato 6](#).

```
SELECT CUSTOMER-FILE ASSIGN TO CUSTMAST.DAT  
      ORGANIZATION IS INDEXED  
      ACCESS MODE IS RANDOM  
      RECORD KEY IS CUSTOMER-ID
```

Listato 6: Esempio di dichiarazione file COBOL

veniva trasformata in una configurazione che utilizzava classi di accesso ai file Java con appropriati livelli di astrazione, preservando la semantica COBOL (accesso indicizzato, chiave di record) mentre si integrava con le API Java moderne per I/O.

- La DATA DIVISION ha rappresentato la sfida tecnica più significativa di questa fase. La complessità derivava dal sistema gerarchico di definizione dei dati in COBOL, che utilizza numeri di livello (01-49, 66, 77, 88) per definire strutture dati annidate con semantiche specifiche, un esempio è rappresentato in [Listato 7](#). Ho implementato un analizzatore ricorsivo che costruiva una rappresentazione interna completa della gerarchia dei dati, gestendo:

- I livelli 01 che definiscono record di primo livello
- I livelli 02-49 che creano strutture gerarchiche
- Il livello 66 per la ridefinizione di gruppi di campi
- Il livello 77 per variabili indipendenti
- Il livello 88 per valori condizionali (condition names)

```

01 WS-MOVIMENTO.
 05 WS-DATA-MOV      PIC X(10).
 05 WS-TIPO-MOV      PIC X(1).
    88 DEPOSITO        VALUE 'D'.
    88 PRELIEVO         VALUE 'P'.
 05 WS-IMPORTO-MOV   PIC S9(7)V99 COMP-3.

```

Listato 7: Struttura dati COBOL per la gestione dei movimenti bancari

La conversione in strutture Java appropriate si è rivelata particolarmente complessa. Un approccio naïve di mappare ogni elemento di gruppo su una classe Java e ogni elemento elementare su un campo produceva codice eccessivamente verboso e non idiomatico. Ho quindi sperimentato due strategie complementari:

- **Appiattimento selettivo:** per gerarchie semplici, i campi venivano appiattiti in una singola classe con nomi composti (es. CUSTOMER-NAME-FIRST diventava customerNameFirst)
- **Preservazione gerarchica:** per strutture complesse o quando la gerarchia aveva significato semantico, utilizzavo classi Java annidate che riflettevano la struttura originale

La gestione delle **PICTURE clauses** ha richiesto un’attenzione particolare. Queste clausole non definiscono solo tipo e dimensione dei dati, ma anche formattazione, gestione del segno, allineamento decimale e altre caratteristiche di presentazione. Ho sviluppato un mini-*parser* specializzato per le PICTURE clauses che le decomponne in attributi gestibili:

PIC Clause	Tipo Java	Note di conversione
PIC 9(5)	int	Validazione del range 0-99999
PIC X(30)	String	Lunghezza massima 30 caratteri
PIC 9(7)V99	BigDecimal	Precisione 9 cifre, 2 decimali
PIC S9(5) COMP-3	BigDecimal	Gestione speciale per packed decimal

Tabella 6: Mappatura delle PICTURE clauses COBOL verso tipi Java

3.3.2 Analisi critica e limiti dell’approccio

Dopo tre settimane di sviluppo intensivo, dedicando la maggior parte del tempo all’implementazione e raffinamento dell’analizzatore, i limiti intrinseci dell’approccio tradizionale sono diventati evidenti. Quello che era iniziato come un esercizio accademico

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: la complessità dei costrutti e delle loro interazioni suggeriva mesi aggiuntivi di sviluppo solo per una copertura parziale.

Il sistema che avevo sviluppato copriva circa il 25% dei costrutti presenti nei codici prodotti nel primo periodo ed una percentuale ancora meno soddisfacente rispetto ai costrutti necessari per gestire programmi del mondo reale.

Continuare l'esplorazione lineare suggeriva almeno altri 2-3 mesi di sviluppo solo per completare la copertura sintattica, ma la complessità non era lineare - ogni nuovo costrutto interagiva con quelli esistenti in modi che richiedevano ristrutturazione del codice esistente.

In una sessione di retrospettiva con la tutor aziendale, Arianna Bellino, abbiamo analizzato criticamente i risultati ottenuti e le prospettive future arrivando alla necessità di esplorazione di alternative più innovative.

3.4 Terzo periodo: pivot verso l'intelligenza artificiale

La necessità di identificare alternative metodologiche innovative, combinata con l'analisi delle soluzioni enterprise esistenti e le evidenze raccolte durante lo sviluppo del *parser* tradizionale, ha determinato l'adozione dell'intelligenza artificiale generativa come approccio risolutivo per il problema della migrazione COBOL-Java. L'evoluzione recente delle capacità dei modelli linguistici di grandi dimensioni (LLM) nel dominio della comprensione e generazione del codice ha aperto nuove prospettive per affrontare la complessità intrinseca della traduzione tra paradigmi di programmazione eterogenei.

3.4.1 Valutazione delle API di *AI* generativa

L'organizzazione ospitante, Miriade, disponeva di accesso enterprise all'API di Google Gemini Pro, fattore che ha determinato la scelta tecnologica e permesso di focalizzare l'analisi sulle capacità specifiche del modello per il task di traduzione del codice.

Google Gemini Pro presentava caratteristiche tecniche particolarmente adatte al progetto di migrazione. Il modello, già dall'interazione via chat dal sito <https://gemini.google.com/>, con upload dei file da tradurre, dimostrava una comprensione sofisticata non solo della sintassi dei linguaggi di programmazione, ma anche della semantica sottostante. Questa comprensione contestuale si rivelava essenziale per produrre traduzioni che preservassero l'intento originale del codice mentre lo modernizzavano per l'ecosistema Java.

La valutazione delle prestazioni del modello ha comportato test sistematici su tre dimensioni principali:

1. **Consistenza delle traduzioni:** verifiche ripetute hanno confermato che input identici producevano output funzionalmente equivalenti *across* multiple sessioni, con variazioni minime limitate a scelte stilistiche non impattanti sulla logica.
2. **Gestione della complessità:** il modello dimostrava capacità di tradurre costrutti COBOL avanzati preservando la logica di business anche in presenza di pattern procedurali complessi e interdipendenze tra moduli.
3. **Qualità del codice prodotto:** l'output generato rispettava consistentemente gli standard Java moderni, producendo codice idiomatico che un developer Java considererebbe naturale e manutenibile.

Un aspetto cruciale nella valutazione riguardava la gestione dei limiti di *stand-up*. I programmi COBOL enterprise possono essere estremamente verbosi, con sezioni di dichiarazione dati che occupano centinaia di righe. Gemini Pro offriva un limite di *stand-up* sufficientemente elevato per gestire la maggior parte dei programmi senza necessità di segmentazione, caratteristica che semplificava notevolmente l'architettura del sistema eliminando la complessità della gestione di traduzioni parziali e successive riconciliazioni.

3.4.2 Design del sistema *AI-powered*

Il sistema segue tre principi architetturali:

- **Comprendere semantica olistica:** analisi del contesto aziendale oltre la forma sintattica
- **Integrazione contestuale:** analisi congiunta di codice e schema database
- **Automazione end-to-end:** produzione di progetti completi pronti per deployment

L'applicativo si articola in tre moduli principali, ciascuno con responsabilità ben definite ma interconnesse attraverso interfacce:

- Il **modulo di traduzione** costituisce il cuore del sistema. La sua architettura interna gestisce la costruzione di *parsing* ottimizzati che codificano la conoscenza domain-specific necessaria per guidare il modello nella traduzione. Il modulo implementa meccanismi di gestione dell'interazione con l'API, includendo retry logic con *backoff esponenziale* per gestire eventuali limitazioni di rate o errori transitori. La validazione dell'output avviene attraverso *parsing* del codice Java generato per assicurare completezza sintattica e presenza di tutti gli elementi strutturali attesi. La configurazione dei parametri generativi ha richiesto una fase di sperimentazione per identificare i valori di generazione ottimali:

- ▶ La temperatura è stata impostata a 0.1, valore estremamente basso che garantisce output deterministici e consistenti, essenziale per un processo di migrazione che richiede ripetibilità.
- ▶ Il parametro top-p, configurato a 0.9, e top-k, limitato a 20, sono stati calibrati per bilanciare la capacità del modello di esplorare soluzioni diverse mantenendo al contempo un controllo stretto sulla qualità e coerenza dell'output.
- ▶ Il limite di 20.000 *stand-up* di output era sufficiente per i codici sviluppati nel primo periodo.
- Il **modulo di packaging** trasforma il codice Java raw in un progetto *Maven* completo. Il punto di forza del modulo risiede nella sua capacità di analizzare il codice generato per identificare automaticamente le dipendenze necessarie. Utilizzando nuovamente Gemini, il modulo esamina gli import statements e l'uso effettivo delle API per determinare non solo quali librerie sono necessarie ma anche le versioni appropriate basandosi su compatibilità e best practice correnti. La generazione del file pom.xml avviene dinamicamente, producendo configurazioni complete che includono sia le dipendenze che anche la configurazione appropriata dei plugin per compilazione, testing, e packaging.
- Il **modulo di orchestrazione** fornisce il layer di coordinamento che trasforma i componenti individuali in un sistema coeso. Implementa una pipeline di esecuzione che gestisce il flusso dei dati tra i moduli, monitora il progresso della conversione, e gestisce condizioni di errore con strategie di recovery appropriate. Il logging da terminale strutturato fornisce visibilità completa sul processo di conversione, essenziale per debugging e audit in contesti enterprise.

Una rappresentazione semplificata e schematica del sistema è mostrata in [Figura 14](#).

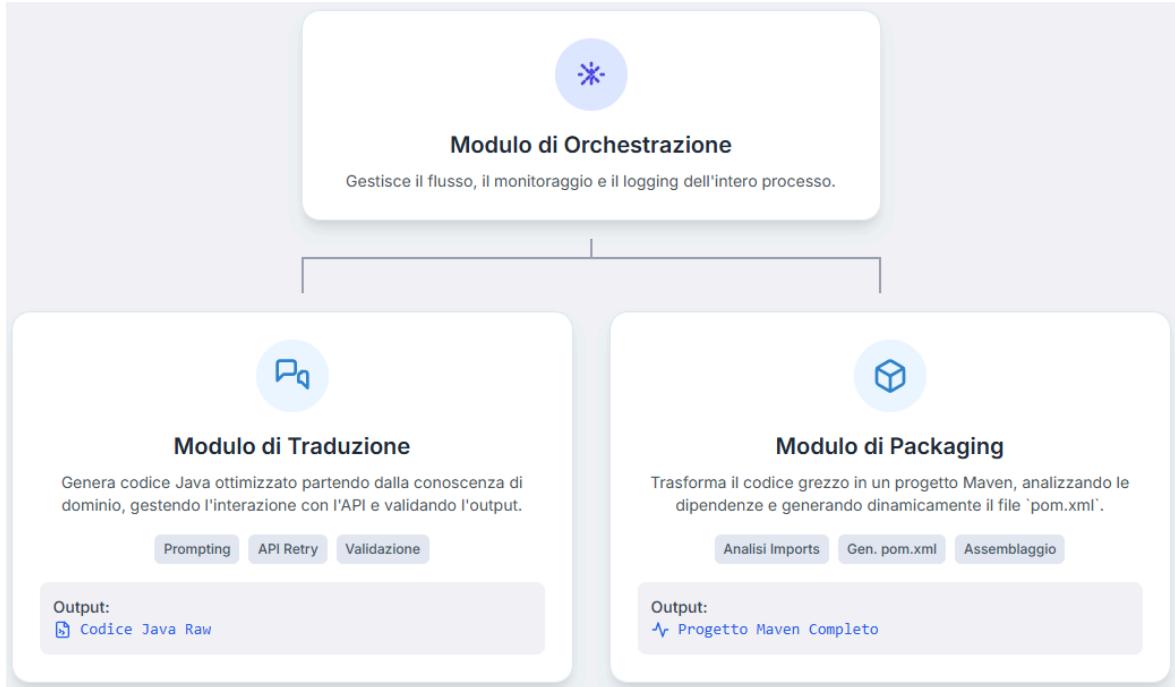


Figura 14: Rappresentazione schematica del sistema di migrazione AI-driven

Il flusso di elaborazione segue una sequenza logica che massimizza le probabilità di successo della conversione. La fase iniziale di acquisizione e validazione assicura che gli input siano completi e ben formati, prevenendo errori downstream. Segue una fase di pre-processamento dove il codice viene normalizzato e preparato per l'analisi, rimuovendo elementi non significativi, preservando struttura e commenti per la comprensione del contesto.

La fase di generazione del *parsing* rappresenta il momento critico dove la conoscenza sulla migrazione viene codificata in forma processabile dal modello. Il *parsing* non è una semplice richiesta ma una specifica dettagliata che include il ruolo del modello, il contesto della traduzione, esempi di pattern di trasformazione, e requisiti specifici per l'output. La costruzione del *prompt* sfrutta template parametrizzati che vengono istanziati con il codice specifico e lo schema database, assicurando consistenza mentre si adatta al contesto specifico.

L'elaborazione della risposta del modello richiede *parsing* e validazione. Il sistema deve estrarre il codice Java dalla risposta del modello, che può includere spiegazioni o metadati aggiuntivi, validare la completezza e correttezza sintattica del codice estratto, e prepararlo per le fasi successive di packaging assicurando che tutti gli elementi necessari siano presenti.

3.5 Quarto periodo: implementazione della soluzione *AI-driven*

L'implementazione operativa del sistema di migrazione basato su intelligenza artificiale ha trasformato il *design* concettuale in una soluzione funzionale sorprendente, capace di gestire

non solo la complessità del codice COBOL autoprodotto ma anche di eventuali necessità enterprise. Questo periodo è stato caratterizzato da un approccio iterativo dove ogni componente veniva sviluppato, testato sui programmi COBOL di esempio creati nella prima fase, e raffinato basandosi sui risultati ottenuti.

3.5.1 Sviluppo del *parsing* engineering

Il *prompt engineering* era l'elemento critico per il successo della traduzione *AI-driven*. Il processo di sviluppo del *parsing* ha seguito una metodologia empirica basata su cicli di sperimentazione e raffinamento.

Il *parsing* doveva stabilire chiaramente il contesto operativo, definendo il modello come un compilatore capace di comprendere non solo sintassi ma semantica e intento aziendale. Questa definizione di ruolo si è dimostrata cruciale per orientare il comportamento del modello verso traduzioni che privilegiassero la preservazione della logica di business rispetto alla traduzione letterale.

Il corpo del *parsing*, di cui un esempio in [Listato 8](#), include sezioni strutturate che guidano il modello attraverso il processo di traduzione:

- La sezione di input fornisce il codice COBOL completo insieme allo schema SQL quando disponibile, permettendo al modello di comprendere il contesto completo dell'applicazione
- Le istruzioni di traduzione specificano come gestire costrutti specifici, fornendo mappature esplicite per tipi di dato, pattern di trasformazione per strutture di controllo, e linee guida per la gestione di costrutti senza equivalenti diretti in Java.

Sei un compilatore avanzato e un traduttore di codice sorgente da COBOL a Java. Il tuo compito è analizzare il seguente codice sorgente COBOL e tradurlo in un singolo file Java moderno, completo, leggibile e compilabile. Il codice Java deve utilizzare JDBC per le operazioni SQL presenti nel programma COBOL.

```
Codice Sorgente COBOL da Tradurre:** {cobol_code}  
Codice SQL associato (se disponibile): {sql_code}
```

Istruzioni di Traduzione Dettagliate e Obbligatorie:

1. Analisi Strutturale:

- IDENTIFICATION DIVISION: Usa il `PROGRAM-ID` per definire il nome della classe Java (es. `GESTIONE-CONTI` -> `GestioneConti`).

...

2. Configurazione Database:**

- Dichiara un campo `private Connection connection;`

- Implementa metodi `connectDatabase()` e `disconnectDatabase()` usando JDBC

...

3. Mappatura Tipi di Dato (dalla `DATA DIVISION`):

- Nomenclatura: Converti le variabili COBOL (es. `WS-NOME`) in camelCase Java (es. `wsNome`).

- `PIC X(n)": Deve diventare `String`.

...

4. Traduzione SQL con JDBC:

- EXEC SQL CONNECT: Traduci in connessione JDBC usando `DriverManager.getConnection()`

...

Listato 8: Esempio di prompt per la traduzione COBOL → Java

L'ottimizzazione iterativa del *parsing* ha richiesto analisi sistematica dei risultati di traduzione. Ogni fallimento o traduzione sub-ottimale forniva informazioni preziose su ambiguità o lacune nelle istruzioni. Pattern comuni di errore includevano:

- Gestione inadeguata delle transazioni database, inizialmente risolta aggiungendo istruzioni specifiche sulla struttura dei blocchi try-catch e la gestione del rollback
- Traduzione letterale di costrutti COBOL che produceva codice Java non idiomático, affrontata attraverso esempi di pattern di trasformazione preferiti

- Perdita di informazioni sui tipi di dato durante la conversione, corretta specificando mappature esplicite e regole di inferenza

La gestione dei casi speciali ha richiesto particolare attenzione nella formulazione del *parsing*. Costrutti COBOL come REDEFINES, che permettono interpretazioni multiple della stessa area di memoria, non hanno equivalenti diretti in Java. Il *parsing* è stato evoluto per includere strategie specifiche di gestione, suggerendo l'uso di classi wrapper con metodi di conversione espliciti. Similmente, la gestione delle tabelle COBOL con OCCURS DEPENDENT ON ha richiesto istruzioni per la creazione di strutture dati dinamiche appropriate in Java, tipicamente ArrayList o array ridimensionabili.

Un aspetto innovativo dello sviluppo del *parsing* è stata l'inclusione di meta-istruzioni che guidano il processo di ragionamento del modello. Invece di fornire solo regole di traduzione meccaniche, il *parsing* incoraggia il modello a considerare, tramite analisi autonoma, l'intento del codice originale, e produrre soluzioni che un developer Java moderno considererebbe naturali. Questo approccio ha prodotto traduzioni significativamente migliori rispetto a *parsing* puramente prescrittivi.

3.5.2 Implementazione del translator completo

Lo sviluppo del sistema di conversione end-to-end ha richiesto l'integrazione di tutti i componenti in un flusso operativo coerente. L'implementazione si è concentrata sulla creazione di un sistema capace di gestire la varietà e complessità del codice COBOL reale.

Il sistema analizza la gerarchia dei level numbers per costruire una rappresentazione interna della struttura dati. Questa analisi identifica:

- Record di primo livello (level 01) che diventano classi Java principali
- Strutture subordinate che vengono mappate a inner classes o campi semplici basandosi sulla complessità
- Elementi ripetuti (OCCURS) che richiedono array o collezioni
- Ridefinizioni (REDEFINES) che necessitano di gestione speciale attraverso union-like patterns

Il processo di generazione delle classi Java corrispondenti applica convenzioni di naming standard, trasformando nomi COBOL in stile KEBAB-CASE in camelCase Java, un sempio in [Listato 9](#). La generazione include automaticamente metodi getter e setter appropriati, costruttori per inizializzazione, e metodi utility per conversioni quando ritenuto necessario dal modello.

```
MOVE WS-IMPORTO TO WS-SALDO  
ADD 100 TO WS-SALDO
```

```
wsSaldo = wsImporto;  
wsSaldo = wsSaldo.add(new  
BigDecimal("100"));
```

Listato 9: Confronto tra sintassi COBOL e Java per operazioni aritmetiche

La traduzione dell'SQL embedded ha richiesto particolare attenzione alla preservazione della semantica transazionale. Il sistema identifica i blocchi EXEC SQL attraverso pattern matching, estrae gli statement SQL e le variabili host coinvolte, e genera codice JDBC equivalente. La generazione utilizza sempre PreparedStatement per prevenire SQL injection, implementa gestione appropriata delle connessioni con pattern try-with-resources, preserva la logica di gestione errori COBOL attraverso mappature SQLCODE appropriate, e mantiene la semantica transazionale con commit e rollback esplicativi.

Un aspetto critico dell'implementazione riguarda la preservazione della logica di business durante la trasformazione. Il translator riconosce pattern comuni nel codice COBOL procedurale e li trasforma in equivalenti object-oriented appropriati:

- I PERFORM statements vengono analizzati per determinare se rappresentano semplici chiamate di subroutine o pattern più complessi come iterazioni
- Le SECTION e PARAGRAPH della PROCEDURE DIVISION vengono trasformate in metodi Java, come illustrato nell'esempio [Listato 10](#), preservando la struttura logica mentre si adotta l'organizzazione object-oriented

```
PROCEDURE DIVISION.  
  
DISPLAY-MENU.  
    DISPLAY "1. Apertura conto"  
    DISPLAY "2. Deposito"  
    DISPLAY "0. Esci".  
  
PROCESS-CHOICE.  
    ACCEPT WS-SCELTA  
    EVALUATE WS-SCELTA  
        WHEN '1' PERFORM OPEN-ACCOUNT  
        WHEN '2' PERFORM MAKE-DEPOSIT  
    END-EVALUATE.
```

```

public class GestioneConti {
    private void displayMenu() {
        System.out.println("1. Apertura conto");
        System.out.println("2. Deposito");
        System.out.println("0. Esci");
    }

    private void processChoice() {
        String scelta = scanner.nextLine();
        switch (scelta) {
            case "1": openAccount(); break;
            case "2": makeDeposit(); break;
        }
    }
}

```

Listato 10: Trasformazione di SECTION e PARAGRAPH COBOL in metodi Java

Per quanto riguarda la gestione degli errori, COBOL spesso utilizza gestione degli errori implicita attraverso controlli di status code, mentre Java favorisce exception handling esplicito. Il translator aggiunge automaticamente blocchi try-catch appropriati dove necessario, preserva i codici di errore COBOL per compatibilità mentre aggiunge exception handling Java, implementa logging strutturato per facilitare debugging e manutenzione, e crea classi di eccezione custom quando pattern di errore specifici lo richiedono.

3.5.3 Generazione automatica di progetti *Maven*

La fase finale del processo di migrazione trasforma il codice Java generato in un progetto completo pronto per il deployment. Questa fase sfrutta nuovamente le capacità di Gemini per analizzare il codice e determinare tutti i requisiti di progetto.

L'analisi delle dipendenze inizia con l'esame degli import statements nel codice Java generato. Il sistema utilizza Gemini per comprendere non solo quali classi vengono importate ma come vengono utilizzate nel codice. Questa analisi contestuale permette di identificare le librerie necessarie con le versioni appropriate, determinare dipendenze transitive che potrebbero essere richieste, escludere dipendenze non necessarie che potrebbero essere state importate ma non utilizzate, e risolvere potenziali conflitti di versione basandosi su best practice correnti.

La struttura del progetto *Maven* viene generata seguendo le convenzioni standard. Il sistema crea automaticamente la gerarchia di directory appropriata, posiziona il codice sorgente nelle location corrette secondo il package structure, prepara directory per risorse, configurazioni, e test, e predispone la struttura per facilitare future estensioni e manutenzione.

La generazione del file pom.xml rappresenta un elemento critico del processo. Il sistema produce configurazioni complete che specificano:

- Coordinate del progetto (groupId, artifactId, version) derivate dal nome del programma COBOL originale
- Proprietà del progetto includendo versioni Java, encoding, e altre configurazioni standard
- Dipendenze identificate attraverso l'analisi del codice con versioni appropriate
- Configurazione dei plugin *Maven* per compilazione, testing, e packaging
- Profili per diversi ambienti di deployment quando identificati dal contesto

La configurazione dei plugin *Maven* riceve particolare attenzione per assicurare che il progetto possa essere costruito e deployato senza modifiche manuali:

- Il *maven-compiler-plugin* viene configurato con la versione Java appropriata basata sulle feature utilizzate nel codice
- Il *maven-jar-plugin* include configurazione per generare JAR eseguibili con manifest appropriato
- Il *maven-assembly-plugin* viene configurato per creare «*fat JARs*» che includono tutte le dipendenze, semplificando il deployment, come esemplificato nel [Listato 11](#)

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.7.1</version>
</dependency>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>11</source>
        <target>11</target>
    </configuration>
</plugin>
```

Listato 11: Esempio di pom.xml generato automaticamente

Il processo di *build* automatizzato verifica la correttezza della configurazione attraverso l’invocazione di *Maven* per compilare il codice, risolvere e scaricare tutte le dipendenze, eseguire eventuali test di base generati, e produrre gli artifact finali. Qualsiasi errore in questa fase viene catturato e reportato con suggerimenti per la risoluzione.

La generazione della documentazione completa il processo. Il sistema preserva i commenti COBOL originali e li arricchisce con informazioni sulla migrazione. I commenti di intestazione dei programmi COBOL vengono trasformati in JavaDoc comprensivi che includono informazioni sull’origine COBOL, data e versione della migrazione, note su trasformazioni significative applicate, e riferimenti alla documentazione originale quando disponibile.

Il risultato finale del processo è un progetto Java completo, moderno, e immediatamente utilizzabile.

3.6 Risultati raggiunti

3.6.1 Impatto dell’AI sui tempi di sviluppo

L’adozione dell’intelligenza artificiale generativa ha determinato una riduzione significativa dei tempi di sviluppo rispetto all’approccio basato su *parsing* deterministico, mantenendo il risultato finale sorprendentemente accurato, come mostra la Figura 15.

```
===== SISTEMA GESTIONE MAGAZZINO =====
1. Carico merce
2. Scarico merce
3. Visualizza giacenza articolo
4. Lista articoli sottoscorta
5. Valorizzazione magazzino
6. Movimenti articolo
7. Rettifica inventario
8. Gestione ordini fornitori
9. Report inventario fisico
10. Analisi ABC articoli
0. Esci
=====
Scelta: 6

== MOVIMENTI ARTICOLO ==
Codice articolo: ART0000006

Ultimi movimenti di: Guanti Nitrile L

TIPO DATA/ORARIO DOCUMENTO QUANTITA VALORE
RI 2025-06-25 13:05 INV20250625 + 2,00 17,00
Rettifica inventario positiva - TestDatabase

Continuare? (S/N):
```

Figura 15: Esecuzione del sistema di gestione conti correnti bancari convertito in Java

Questa riduzione temporale deriva da diversi fattori tecnici.

- L’approccio tradizionale richiedeva l’implementazione esplicita di regole di trasformazione per ogni costrutto COBOL, con complessità computazionale crescente per le

interazioni tra costrutti diversi. L'*AI*, invece, opera attraverso comprensione contestuale del codice, permettendo di gestire simultaneamente aspetti sintattici e semantici della traduzione. La capacità del modello di interpretare l'intento del codice, oltre alla sua struttura formale, ha eliminato la necessità di codificare manualmente centinaia di eccezioni, casi speciali e specifici.

- L'accelerazione del processo ha inoltre consentito un approccio iterativo al *prompt engineering*. Durante lo sviluppo, è stato possibile raffinare progressivamente le istruzioni fornite al modello basandosi sui risultati ottenuti, ottimizzando la qualità delle traduzioni attraverso cicli rapidi di sperimentazione. Questo approccio empirico non sarebbe stato praticabile con tempi di sviluppo nell'ordine dei mesi per ogni iterazione.

Un aspetto rilevante riguarda la consistenza dei risultati. Mentre lo sviluppo manuale tende a introdurre variabilità nelle convenzioni di codifica e negli approcci implementativi man mano che il progetto evolve, l'*AI* mantiene coerenza stilistica e architettonica attraverso l'intero processo di conversione, applicando uniformemente i pattern di trasformazione identificati.

3.6.2 Analisi qualitativa dei risultati

L'analisi del codice prodotto rivela caratteristiche qualitative che non si limitano alla correttezza funzionale. L'[Figura 16](#) illustra il processo completo di conversione sul terminale, dalla lettura dei file sorgente alla generazione del progetto *Maven* finale.

```

PROBLEMS 3 OUTPUT TERMINAL PORTS
=====
===== COBOL to Java Converter - Processo Completo =====
=====

[PREPARAZIONE] 🔍 Verifica file di input
    📁 Ricerca in: /Users/luca/Downloads/COBOL_to_Java_Converter/bank_system_cobol/
        ✓ File COBOL: bank_system_cobol.cbl
        ✓ File SQL: bank_schema.sql
    📂 Nome progetto: bank_system_cobol

[FASE 1/3] 🚀 Traduzione COBOL → Java
    📁 Lettura file COBOL: bank_system_cobol.cbl
        ✓ File COBOL letto correttamente
    📁 Lettura schema SQL: bank_schema.sql
        ✓ Schema SQL caricato
    🚧 Traduzione in corso...
        📡 Invio richiesta a Gemini API...
        ✓ Risposta ricevuta da Gemini
        ✓ File Java salvato: /Users/luca/Downloads/COBOL_to_Java_Converter/bank_system_cobol/target/generated-sources/main/java/com/gestioneconti/bank_system_cobol/BankSystemCobol.java
    ✅ Traduzione completata
    ✓ Fase 1 completata

[FASE 2/3] 🏺 Creazione progetto Maven e JAR
    📁 File sorgente: bank_system_cobol.java
        ✓ Nome progetto: bank_system_cobol
        ✓ Classe principale: GestioneConti
    🚧 Creazione struttura Maven...
        ✓ Directory create: bank_system_cobol/
    🚧 Preparazione file Java...
        ✓ Package declaration aggiunto
        ✓ File Java copiato
    🚧 Generazione pom.xml...
        📡 Invio richiesta a Gemini API...
        ✓ Risposta ricevuta da Gemini
        ✓ pom.xml creato
    🚧 Compilazione e creazione JAR...
        ✓ JAR creato con successo
    ✅ Processo completato
        ✓ JAR creato: GestioneConti-1.0.0-jar-with-dependencies.jar
    ✓ Fase 2 completata

[FASE 3/3] 📁 Archiviazione file originali
    📁 Copia file in: input/
        ✓ bank_system_cobol.cbl
        ✓ bank_schema.sql
    ✓ Fase 3 completata

```

Figura 16: Output nel terminale del sistema di migrazione AI-driven

Il sistema ha dimostrato capacità di gestire la complessità semantica della traduzione COBOL-Java. Per le strutture dati gerarchiche, caratteristica distintiva di COBOL, l'*AI* ha applicato strategie di conversione appropriate al contesto d'uso.

La gestione dell'SQL (SQL) embedded ha evidenziato capacità di trasformazione multi-livello. Il sistema ha identificato i blocchi EXEC SQL, estratto le query, e generato codice JDBC idiomatico con gestione appropriata di connessioni, prepared statements e transazioni. La conversione ha mantenuto la semantica transazionale originale adattandola ai pattern Java, includendo gestione delle eccezioni e rilascio delle risorse secondo le *best practice* del linguaggio target.

La documentazione generata attraverso *JavaDoc* rappresenta un valore aggiunto significativo. Il sistema è in grado di preservare i commenti originali e li contestualizzarli nel nuovo ambiente, aggiungendo metadati sulla migrazione e mappature tra costrutti COBOL e Java.

Questa documentazione facilita la manutenzione futura fornendo contesto storico e il valore razionale delle scelte implementative.

3.6.3 Risultati quantitativi

I risultati ottenuti possono essere valutati sistematicamente in relazione agli obiettivi definiti nel Capitolo 2, organizzati secondo i tre livelli di priorità stabiliti.

Obiettivi obbligatori

Il sistema ha conseguito il pieno raggiungimento di tutti gli obiettivi obbligatori:

- OO01 - Prodotti tre progetti COBOL completi
- OO02 - Esplorate strategie di migrazione che prendevano in considerazione diverse tecnologie
- OO03 - Copertura delle divisioni: Il requisito minimo del 75% è stato superato con una conversione completa (100%) di tutte e quattro le divisioni COBOL fondamentali (*Identification, Environment, Data, Procedure*).
- OO04 - Esplorate strategie open-source e commerciali, documentandone le peculiarità
- OO05 - Progetti migrati: Rispetto all'obiettivo di completare almeno una migrazione funzionante, sono stati convertiti con successo tutti e tre i progetti autoprodotti:
 - GESTIONE-CONTI: sistema di gestione conti correnti bancari
 - GESTIONE-PAGHE: sistema di gestione risorse umane e stipendi
 - GESTIONE-MAGAZZINO: sistema di logistica e inventario
- OO06 - Qualità del codice Java: Il codice generato rispetta pienamente le convenzioni Java moderne, includendo:
 - Struttura standard dei *package*
 - Nomenclatura consistente con le convenzioni del linguaggio
 - Documentazione *JavaDoc* completa e professionale
 - Oltre 2000 linee di codice che superano verifiche di analisi statica
- OO08 - Produzione di un *README* dettagliato e documentazione del processo.

Obiettivi desiderabili

Anche gli obiettivi desiderabili sono stati largamente conseguiti:

- OD01 - Copertura completa: Raggiunto il 100% di conversione automatica del codice COBOL autoprodotto.
- OD02 - Gestione costrutti complessi: Dimostrata attraverso la conversione efficace di:
 - Strutture dati gerarchiche con livelli multipli
 - SQL embedded con transazioni complesse
 - Logiche procedurali articolate e interdipendenti

- OD03 - Ottimizzazione del codice: Sebbene non implementata come fase separata, il codice prodotto dall'*AI* risulta intrinsecamente ottimizzato secondo pattern idiomatici Java.

Risultati aggiuntivi non previsti

Un risultato particolarmente significativo non contemplato negli obiettivi iniziali è la generazione automatica di progetti *Maven* completi, [Figura 17](#). Questa funzionalità include:

- Struttura di progetto standard con directory appropriate
- File `pom.xml` con gestione completa delle dipendenze
- Configurazione del *build* per generazione di JAR eseguibili
- Integrazione immediata in pipeline CI/CD moderne

```
=====
CONVERSIONE COMPLETATA CON SUCCESSO
=====

└─ Progetto creato in: bank_system_cobol/
└─ JAR eseguibile: target/GestioneConti-1.0.0-jar-with-dependencies.jar

└─ JAR disponibili:
    - target/GestioneConti-1.0.0.jar
    - target/GestioneConti-1.0.0-jar-with-dependencies.jar

└─ Per eseguire il programma:
    $ cd bank_system_cobol
    $ java -jar target/GestioneConti-1.0.0-jar-with-dependencies.jar
```

Figura 17: Progetto Maven generato automaticamente dal sistema di migrazione

Questi risultati dimostrano come l'approccio basato su *AI* non solo abbia soddisfatto i requisiti progettuali stabiliti, ma abbia anche aperto possibilità non inizialmente contemplate, trasformando il progetto da prototipo dimostrativo a soluzione potenzialmente applicabile in contesti produttivi reali.

4 Valutazioni retrospettive e prospettive future

Qui introdurrò brevemente il contenuto delle sezioni sottostanti.

4.1 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.

4.2 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.

4.3 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 sistematica della modernizzazione IT maturata e la capacità di valutare e integrare pragmaticamente tecnologie emergenti.

4.4 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».

4.5 Roadmap evolutiva e opportunità di sviluppo

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

5 Lista degli acronimi

AI: Artificial Intelligence

ANTLR: ANother Tool for Language Recognition

API: Application Programming Interface

AST: Abstract Syntax Tree

CLI: Command Line Interface

COBOL: Common Business-Oriented Language

IT: Information Technology

JDBC: Java Database Connectivity

JSON: JavaScript Object Notation

LLM: Large Language Model

6 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

backoff esponenziale: Strategia di retry con attese progressivamente più lunghe

legacy: Sistemi informatici datati ma ancora in uso

mainframe: Computer di grandi dimensioni per elaborazioni complesse

microservizi: Architettura software basata su servizi indipendenti

parser: Analizzatore sintattico che interpreta la struttura del codice

prompt engineering: Tecnica di formulazione di istruzioni per modelli AI

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

token: Unità base di testo processata dai modelli linguistici

7 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>