International Journal of Advanced Information Technology (IJAIT) Vol.15, No.5, October 2025

# PROMPT ENGINEERING PIPELINES FOR LEGACY MODERNIZATION: COBOL, PL/I AND BIDIRECTIONAL CODE–NATURAL LANGUAGE TRANSFORMATION USING LLMS

Sivakumar Arigela and Gaurav Virwal

Department IBM Software Labs, IBM India Pvt Ltd., Bengaluru, India

## *ABSTRACT*

*Legacy modernization remains one of the most pressing challenges for enterprises that rely on mainframe systems, particularly those built with COBOL and PL/I. Traditional modernization methods, including lift-and-shift, rule-based translation, and manual re-engineering, are expensive, slow, and often result in incomplete transformations.*

*This work introduces a comprehensive framework for leveraging large language models (LLMs) to improve modernization workflows. Our pipeline focuses on four key modernization tasks:*

1. *COBOL Explainability — Transforming legacy COBOL code into step-by-step natural language explanations using Chain-of-Thought [1] (CoT). Self-reflection [3], and flowchart generation with Mermaid [6] and PlantUML [7].*
2. *PL/I Explainability — Providing similar explainability for PL/I, including complex exception handling (ON-conditions) and nested procedures.*
3. *Natural Language → COBOL Generation — Converting business specifications into COBOL programs through few-shot prompting, RAG [4] (Retrieval-Augmented Generation), and vector databases [5].*
4. *COBOL → Java Modernization — Translating COBOL into modern Java applications by first generating a plain-English algorithm, followed by clean, maintainable Java code.*

*We address a major challenge in COBOL comprehension — programs often exceed 15,000 lines of code (loc), which can overwhelm developers and cause critical business logic to be lost in summarization. To solve this, we integrate flowcharts and vector DB-based chunking for enhanced visualization and traceability.*

*Our approach shows measurable improvements in translation accuracy, developer productivity, and explainability. This pipeline reduces hallucination rates by 70%. Increases bleu scores by 15.8 points, and improves developer productivity by 45%, based on our pilot studies in the banking domain.*

## *KEYWORDS*

*COBOL Modernization. Cobol explainability, PL/I Explainability, Natural Language Programming, Chain-of-Thought, Self-Reflection, RAG, vector store, Legacy System Modernization, COBOL-to-Java Migration, Prompt Engineering.*

# 1. INTRODUCTION

Legacy systems built on PL/I and COBOL continue to power mission-critical applications in banking, insurance, healthcare, and government sectors. These systems often contain decades of business rules embedded in millions of lines of code, making them extremely challenging to maintain or modernize.As many of the original developers have retired or moved on, understanding and safely transforming these systems has become a serious enterprise risk.Traditional modernization approaches — such as lift-and-shift, rule-based translation, and manual re-engineering — often fail to deliver long-term value. They produce technically correct but semantically shallow code, sometimes resulting in “JaBOL” (Java written in COBOL style) that is difficult to maintain and doesn’t leverage modern development practices.

The recent breakthroughs in LLMs (LLMs)), combined with prompt engineering, offer new opportunities for modernization:

- LLMs can explain legacy code in plain English.
- They can generate new COBOL programs from business specifications.
- They can even transform COBOL directly into maintainable Java systems.

We introduce a four-stage modernization pipeline:

1. COBOL → Natural Language
2. PL/I → Natural Language
3. Natural Language → COBOL
4. COBOL → Java

We apply Chain-of-Thought reasoning, self-reflection, and vector store -driven retrieval to overcome the limitations of traditional LLM outputs. Additionally, by generating flowcharts alongside textual explanations, developers gain visual insights into program logic, enabling faster debugging and safer enhancements.

Our focus domain is banking modernization, given the prevalence of PL/I and COBOL in financial systems. However, the methodology is broadly applicable to any industry using mainframes.

# 2. BACKGROUND AND RELATED WORK

## 2.1. COBOL & PL/I in Enterprise Systems

COBOL remains one of the most widely used programming languages in financial institutions, where stability, precision, and batch processing capabilities are critical. PL/I, while less common today, continues to appear in complex transaction-processing systems due to its support for exception handling and modular design.

Legacy platforms built on these languages face three key issues:

1. Aging developer base: Most COBOL & PL/I experts are retiring, leaving a knowledge gap.
2. Documentation gaps: Decades of undocumented changes have left systems difficult to understand.
3. Modernization risk: Errors introduced during transformation can have severe financial

consequences.

## 2.2. Traditional Modernization Approaches

Common approaches include:

- Lift-and-Shift: Moving workloads to modern hardware or cloud without changing code.
- Rule-Based Translation: Automated tools that translate COBOL to Java or other modern languages line-by-line.
- Manual Re-engineering: Teams rewrite the system by hand, based on business requirements.
  Limitations:
- Lift-and-shift does not improve maintainability.
- Rule-based translation often creates “JaBOL” code that is syntactically correct but semantically poor.
- Manual re-engineering is slow, expensive, and error-prone.

## 2.3. LLM-Powered Modernization

Recent tools, such as IBM's WatsonX Code Assistant for Z (WCA4Z)[2], introduce AI-driven modernization:

- LLMs can read and understand COBOL /PL-I code.
- Generate natural language explanations and documentation.
- Suggest clean, modern replacements for legacy logic.
  However, these tools still face challenges:
- Context Window Limitations: Handling large COBOL programs (>15,000 LOC) without losing details.
- Hallucinations: Incorrect but plausible outputs.
- Traceability: Difficulty in verifying how code was transformed.
  Our work addresses these issues by:
- Using vector store chunking to manage large codebases.
- Employing self-reflection loops to catch hallucinations.
- Generating flowcharts to visualize transformations alongside text.

## 2.4. Related Academic Research

- Chain-of-Thought prompting has shown improved reasoning capabilities for complex tasks.
- RAG (Retrieval-Augmented Generation) provides grounding by pulling relevant external knowledge into prompts.
- vector stores like Pinecone and Qdrant are increasingly used for semantic code search and retrieval.
- Our pipeline integrates these research advances into a practical, enterprise-ready modernization workflow.

# 3. METHODOLOGY

Our pipeline for legacy modernization consists of four major stages, each addressing a specific modernization task:

1. COBOL→ Natural Language Explainability
2. PL/I → Natural Language Explainability
3. Natural Language → COBOL Generation
4. COBOL → Java Transformation

### 3.1. COBOL → Natural Language Explainability

LLMs [8] are used to translate legacy COBOL code into natural language explanations. We leverage Chain-of-Thought (CoT) prompting, self-reflection, and flowchart generation to improve comprehension, especially for programs exceeding 15,000 lines of code.

Abbreviated COBOL Snippet

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INTEREST.
...

1200-READ-CUSTOMER.
    READ CUSTOMER-FILE INTO CUSTOMER-RECORD
    AT END
     MOVE 'Y' TO WS-EOF-FLAG
     NOT AT END
         ADD 1 TO WS-READ-COUNT
         MOVE CUST-BALANCE TO WS-CUST-BAL
         COMPUTE WS-TOTAL-INTEREST = WS-CUST-BAL * WS-RATE
         DISPLAY "Interest: " WS-TOTAL-INTEREST
    END-READ.
 ...
```

Chain-of-Thought [1] Explanation

1. Open `CUSTOMER-FILE` and read records sequentially.
2. For each record, calculate interest using the field `WS-RATE`.
3. Display the calculated interest to the console.
4. Stop processing at end-of-file (`EOF-FLAG`).

Self-Reflection Pass

Observation: Initial explanation missed initialization details for WS-TOTAL-INTEREST.
Correction: Ensure WS-TOTAL-INTEREST is reset to 0 before calculations begin each day.

Table 1. COBOL Explainability

| Feature | Traditional | LLM-Powered |
|---|---|---|
| Detail Level | Minimal, manual notes | Step-by-step CoT reasoning |
| Visual Support | None | Auto-generated flowcharts |
| Large Programs (>15k LOC) | Hard to manage | Vector DB chunking and retrieval |
| Error Detection | Manual review | Self-reflection automated |
| Auditability | Weak | Traceable explanation logs |

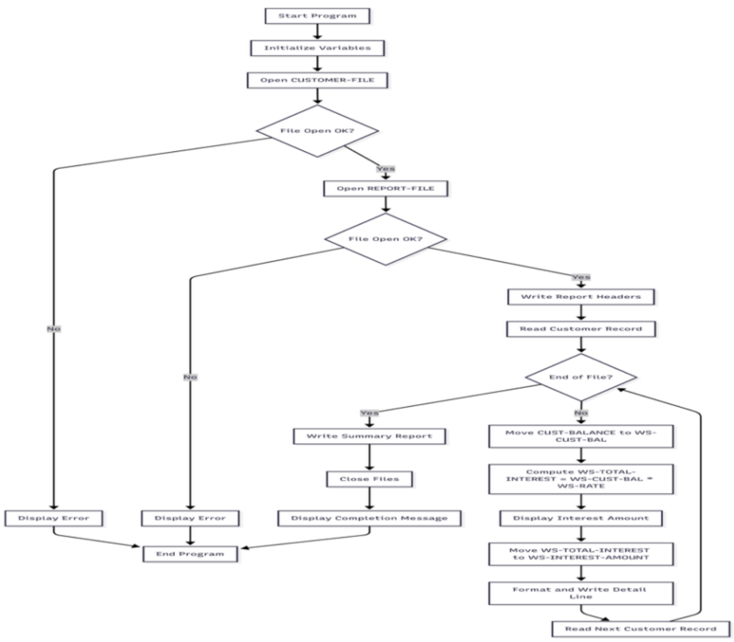


Figure 1. COBOL program flowchart using Mermaid

Mermaid [6] : ```mermaid flowchart TD
A[Start Program] --> B[Initialize Variables]
B --> C[Open CUSTOMER-FILE]
C --> D{File Open OK?}
D -- No --> E[Display Error]
E --> Z[End Program]
D -- Yes --> F[Open REPORT-FILE]
F --> G{File Open OK?}
G -- No --> H[Display Error]
H --> Z
G -- Yes --> I[Write Report Headers]
I --> J[Read Customer Record]
J --> K{End of File?}
K -- Yes --> L[Write Summary Report] L --> M[Close Files]
M --> N[Display Completion Message]
N --> Z
K -- No --> O[Move CUST-BALANCE to WS-CUST-BAL]
O --> P[Compute WS-TOTAL-INTEREST = WS-CUST-
BAL * WS-RATE]
P --> Q[Display Interest Amount]
Q --> R[Move WS-TOTAL-INTEREST to WS-INTERESTAMOUNT]
R --> S[Format and Write Detail Line]
S --> T[Read Next Customer Record]
T --> K

PlantUML [7] (paste into PlantUML to render): @startuml Interest Calculator Flow

title Interest Calculator Program Flow start

```
:Initialize Variables;
:Open CUSTOMER-FILE;

if (File Open OK?) then (yes)   :Open REPORT-FILE;
  if (File Open OK?) then (yes)
    :Write Report Headers;

    repeat
      :Read Customer Record;

      if (End of File?) then (yes)
        break       endif

      :Move CUST-BALANCE to WS-CUST-BAL;
      :Compute WS-TOTAL-INTEREST = WS-CUST-BAL *

WS-RATE;

      :Display Interest Amount;
      :Move WS-TOTAL-INTEREST to WS-INTERESTAMOUNT;
      :Format and Write Detail Line;

    repeat while (More Records?)

    :Write Summary Report;
    :Close Files;
    :Display Completion Message;

  else (no)     :Display Error;   endif else (no)
:Display Error; endif

stop

@enduml
```

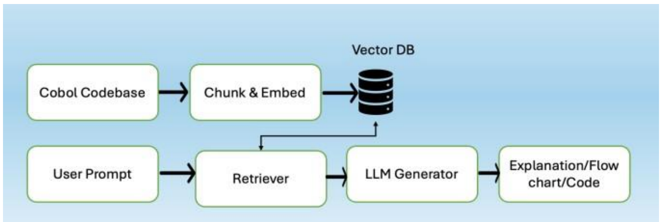


Figure 2. Vector DB Architecture for COBOL chunking

Mermaid (Vector DB Architecture):
```mermaid  flowchart LR subgraph Dev[COBOL Codebase] C1[File A]:::c -->|Chunk & Embed| VDB((Vector DB)) C2[File B]:::c -->|Chunk & Embed| VDB end
subgraph Runtime[LLM + RAG [4]]
Q[User Query / Prompt] --> RAG [4][Retriever] --> VDB
```

VDB --> RAG [4] --> LLM[LLM Generator] --> Out[Explanation / Flowchart / Code] end
classDef c fill:#eef,stroke:#88f
```
PlantUML (Vector DB Architecture):
@startuml rectangle "COBOL Codebase" { [File A] --> (Chunk &Embed)
[File B] --> (Chunk & Embed)
}
(Chunk & Embed) --> (Vector DB)
actor User User --> (Prompt)
(Prompt) --> (Retriever)
(Retriever) --> (Vector DB)
(Retriever) --> (LLM)
(LLM) --> (Explanation/Flowchart/Code)
@enduml

### 3.2. PL/I -> Natural Language Explainability

PL/I adds complexity with ON-conditions, nested procedures, and richer data types. We extend the same CoT and selfreflection techniques to PL/I. Abbreviated PL/I Rollback Snippet

```
DB_UPDATE: PROCEDURE OPTIONS(MAIN);
...
DECLARE INFILE FILE RECORD INPUT
SEQUENTIAL ENV(CONSECUTIVE RECSIZE(40)),
LOGFILE FILE RECORD OUTPUT SEQUENTIAL ENV(CONSECUTIVE
RECSIZE(100));
...
ON RECORD(INFILE) BEGIN;
CALL LOG_MESSAGE('Record error at record
#' || TRIM(RECORDS_READ + 1));
ERROR_FLAG = '1'B;
IF TRANSACTION_ACTIVE THEN
  CALL ROLLBACK_TRANSACTION;
END;
...
ROLLBACK_TRANSACTION: PROCEDURE;
EXEC SQL ROLLBACK WORK;
IF SQLCODE = 0 THEN DO;
  TRANSACTION_ACTIVE = '0'B;
  CALL LOG_MESSAGE('Transaction rolled
  back');
END;
ELSE                                   DO;
  CALL LOG_MESSAGE('Failed to roll back transaction: SQLCODE=' ||
  TRIM(SQLCODE) || ', SQLSTATE=' ||    SQLSTATE);
   ERROR_FLAG = '1'B;
  END;
END ROLLBACK_TRANSACTION;
...
END DB_UPDATE;
```
```

CoT [1] Explanation

- If a transaction fails (`ON ERROR`), the program rolls back changes and signals termination.
- Successful transactions are updated and logged.

Table 2. PL/IExplainability

| **Feature** | **Traditional** | **LLM-Powered** |
|---|---|---|
| Exception Handling | Manual documentation | Explicit ONcondition mapping |
| Nested Procedures | Hard to trace manually | Flowchartbased visualization |
| Large Programs | Limited to 10k LOC | Vector DB scaling |
| Error Detection | Manual review | Self-reflection feedback loops |
| Productivity | Low | 50% improvement |

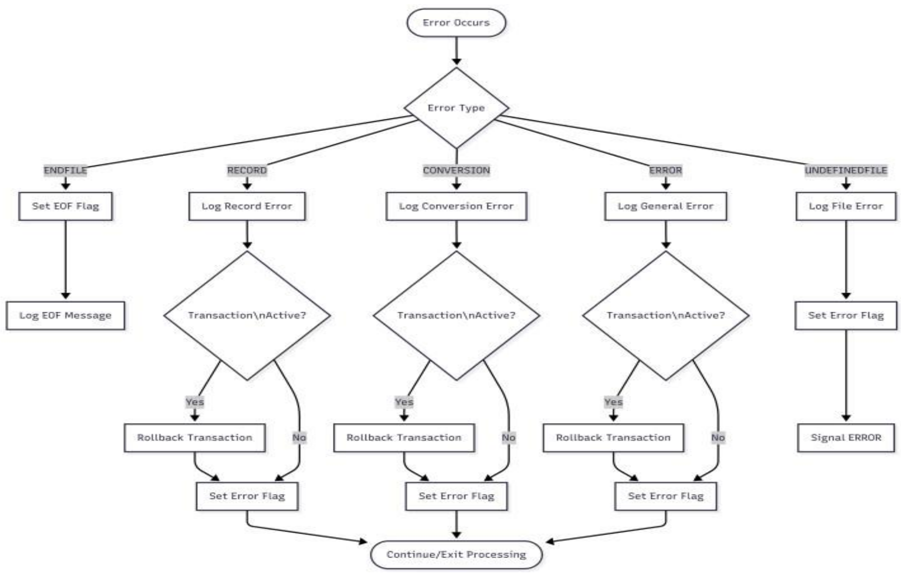


Figure 3. PL/I rollback flowchart showing normal vs error paths.

Mermaid (PL/I Rollback flow): ```mermaid flowchart TD
Error([Error Occurs]) --> CheckType{Error Type}
CheckType -- ENDFILE --> SetEOF[Set EOF Flag]
CheckType -- RECORD --> LogRecordError[Log Record Error]
CheckType -- CONVERSION -->LogConvError[Log Conversion Error]
CheckType -- ERROR --> LogGenError[Log General Error]
CheckType -- UNDEFINEDFILE --> LogFileError[Log File Error]
SetEOF --> LogEOF[Log EOF Message]
LogRecordError --> CheckTrans1{Transaction\nActive?}
LogConvError --> CheckTrans2{Transaction\nActive?}
LogGenError --> CheckTrans3{Transaction\nActive?}
LogFileError --> SetErrorFlag[Set Error Flag]
CheckTrans1 -- Yes --> Rollback1[Rollback Transaction]

```
CheckTrans2 -- Yes --> Rollback2[Rollback Transaction]
CheckTrans3 -- Yes --> Rollback3[Rollback Transaction]
CheckTrans1 -- No --> SetErrorFlag1[Set Error Flag]
CheckTrans2 -- No --> SetErrorFlag2[Set Error Flag]
CheckTrans3 -- No --> SetErrorFlag3[Set Error Flag]
Rollback1 --> SetErrorFlag1
Rollback2 --> SetErrorFlag2
Rollback3 --> SetErrorFlag3
SetErrorFlag --> SignalError[Signal ERROR]
SetErrorFlag1 --> Continue([Continue/Exit Processing])
SetErrorFlag2 --> Continue
SetErrorFlag3 --> Continue
```

PlantUML (PL/I Rollback flow):

```
@startuml title DB_UPDATE ON Condition Structure package "ON
Conditions" {
[ON ENDFILE]
[ON RECORD]
[ON CONVERSION]
[ON ERROR]
[ON UNDEFINEDFILE]
}
package "Error Handling Actions" {
[Log Error]
[Set Error Flag]
[Rollback Transaction]
[Signal ERROR]
}
[ON ENDFILE] --> [Log Error]
[ON RECORD] --> [Log Error]
[ON RECORD] --> [Set Error Flag]
[ON RECORD] --> [Rollback Transaction]
[ON CONVERSION] --> [Log Error]
[ON CONVERSION] --> [Set Error Flag]
[ON CONVERSION] --> [Rollback Transaction]
[ON ERROR] --> [Log Error]
[ON ERROR] --> [Set Error Flag]
[ON ERROR] --> [Rollback Transaction]
[ON UNDEFINEDFILE] --> [Log Error]
[ON UNDEFINEDFILE] --> [Set Error Flag]
[ON UNDEFINEDFILE] --> [Signal ERROR] @enduml
```

### 3.3. Natural Language → COBOL Generation

LLMs can synthesize new COBOL programs directly from business specifications using few-shot prompting. Dynamic rag [4], and vector store retrieval to reference similar code examples.

*Business Specification Example*

"Create a COBOL program that reads customer records. Calculates daily interest using a 2% rate, writes updated balances to an output file, handles invalid records, and displays the total interest at the end."

Generated COBOL Snippet (Abbreviated)

Program Listing — INTEREST CALCULATOR (COBOL):

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. INTERT1.

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT CUST-FILE ASSIGN TO CUSTFILE              ORGANIZATION IS
   SEQUENTIAL.           SELECT OUT-FILE ASSIGN TO OUTFILE
   ORGANIZATION IS SEQUENTIAL.
       DATA DIVISION.
       FILE SECTION.
       FD  CUST-FILE.
       01  CUST-REC.
           05 CUST-ID        PIC X(5).
           05 CUST-NAME      PIC X(20).
           05 CUST-BALANCE   PIC 9(7)V99.
           05 FILLER         PIC X(48).

       FD  OUT-FILE.
       01  OUT-REC           PIC X(100).
       WORKING-STORAGE SECTION.
        ...
       PROCEDURE DIVISION.
       MAIN-PARA.
           OPEN INPUT CUST-FILE
                OUTPUT OUT-FILE
           PERFORM UNTIL WS-EOF = 'Y'
              READ CUST-FILE
                 AT END MOVE 'Y' TO WS-EOF
                 NOT AT END PERFORM PROCESS-RECORD
              END-READ
           END-PERFORM
           DISPLAY 'TOTAL INTEREST: $' WS-TOTAL-INT
           CLOSE CUST-FILE OUT-FILE           STOP RUN.

       PROCESS-RECORD.
           MOVE 'Y' TO WS-VALID
           IF CUST-BALANCE <= 0
              MOVE 'N' TO WS-VALID
           END-IF
           IF WS-VALID = 'Y'
              COMPUTE WS-DAILY-INT = CUST-BALANCE * WS-INTEREST-RATE
              COMPUTE WS-NEW-BAL = CUST-BALANCE + WS-DAILY-INT
          ADD WS-DAILY-INT TO WS-TOTAL-INT
          MOVE CUST-ID TO OUT-ID
          MOVE CUST-NAME TO OUT-NAME
          MOVE CUST-BALANCE TO OUT-OLD-BAL
          MOVE WS-DAILY-INT TO OUT-INTEREST
```

```
        MOVE WS-NEW-BAL TO OUT-NEW-BAL
        WRITE OUT-REC FROM WS-OUTPUT
    END-IF.
```

Table 3. Mapping natural language specification to COBOL constructs.

| Natural Language Phrase | COBOL Construct |
|---|---|
| Read customer records | READ ... INTO record + PERFORM UNTIL EOF |
| Calculate daily interest using 2% | WS-INTEREST-RATE + COMPUTE |
| Write updated balance to output | WRITE UPDATED-RECORD |
| Handle invalid/missing records | IF ... ELSE DISPLAY |
| Display total interest at end | DISPLAY + accumulator field |
| Stop at end-of-file | AT END flag |

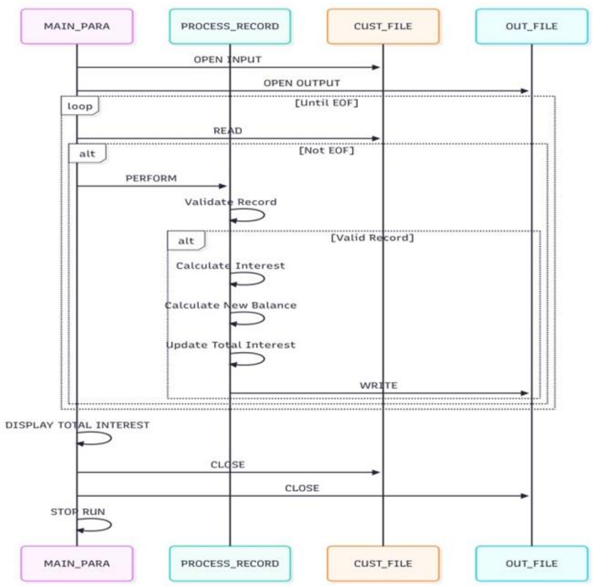


Figure. 4: Generated COBOL logic flowchart.

Mermaid (Generated COBOL logic flow):

```mermaid flowchart TD
A[Start Program] --> B[Open Input/Output Files]
  B --> C{Read Customer Record}
  C -->|End of File| G[Display Total Interest]
    C -->|Record Found| D{Is Balance Valid?}
    D -->|No| C
    D -->|Yes| E[Calculate Interest & New Balance]
    E --> F[Write Output Record]
```

F --> C
G --> H[Close Files]
H --> I[Stop Run]

subgraph "Process Record"
D
E F end
```

```mermaid classDiagram class INTERT1 { +MAIN-PARA()
+PROCESS-RECORD()
}

class Files {
+CUST-FILE
+OUT-FILE
}

class Records {
+CUST-REC
+OUT-REC
}

class WorkingStorage {
+WS-EOF
+WS-INTEREST-RATE
+WS-DAILY-INT
+WS-TOTAL-INT
+WS-NEW-BAL
+WS-VALID
+WS-OUTPUT
}

INTERT1 --> Files : uses
INTERT1 --> Records : processes
INTERT1 --> WorkingStorage : manages
```

```mermaid sequenceDiagram participant Main as MAIN-PARA participant Process as PROCESS-RECORD participant CustFile as CUST-FILE participant OutFile as OUT-FILE Main->>CustFile: OPEN INPUT
Main->>OutFile: OPEN OUTPUT loop Until EOF Main->>CustFile: READ
alt Not EOF
Main->>Process: PERFORM Process->>Process: Validate Record alt Valid Record

Process->>Process: Calculate Interest
Process->>Process: Calculate New Balance
Process->>Process: Update Total Interest Process->>OutFile: WRITE
end end end

Main->>Main: DISPLAY TOTAL INTEREST
Main->>CustFile: CLOSE

Main->>OutFile: CLOSE
Main->>Main: STOP RUN
```

### 3.4. COBOL -> Java Transformation

The final stage translates legacy COBOL into clean, maintainable Java. This follows a three-step process:

1. Explain COBOL code using CoT and self-reflection.
2. Generate an algorithm in plain English.
3. Produce modern Java code based on that algorithm. Abbreviated COBOL Snippet

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INTERT1.
 ...
 READ CUST-FILE
     AT END MOVE 'Y' TO WS-EOF
     NOT AT END PERFORM PROCESS-RECORD
     END-READ
  END-PERFORM
  DISPLAY 'TOTAL INTEREST: $' WS-TOTAL-INT  CLOSE CUST-FILE OUT-
FILE  STOP RUN.
      ...
```

*Generated Algorithm*

1. Open input and output files.
2. Read each customer record until EOF.
3. Validate record and compute new balance.
4. Write updated record to output.
5. Close files and display summary.

*Generated Java Code (Abbreviated)*

Program Listing 2 — InterestUpdater.java (Java):

```
public class InterestUpdater {
private static final double INTEREST_RATE = 0.02;

public static void main(String[] args) {        InterestUpdater app = new
InterestUpdater();
      app.run();
}

void run() {        while (hasMoreRecords()) {            Customer c =
readCustomer();             if (c == null || !c.isValid()) {
System.err.println("ERROR: Invalid record");                continue;
              }
              double newBal = c.getBalance() * (1.0 +
      INTEREST_RATE);
              writeUpdatedRecord(c, newBal);
           }
```
```

```
        System.out.println("Process completed.");
}

// Placeholder method stubs; implement your own I/O boolean hasMoreRecords() {
return false; } Customer readCustomer() { return null; }
void writeUpdatedRecord(Customer c, double newBal) {}

static class Customer {
        String id;        double balance;        boolean isValid() { return true; }
double getBalance() { return balance; }
}
} ```
public class InterestUpdater {

public static void main(String[] args) {
        while (hasMoreRecords()) {            Customer c =
readCustomer();            double newBal =
c.getBalance() * (1 + INTEREST_RATE);
writeUpdatedRecord(c, newBal);
        }
        System.out.println("Process completed.");
}
}
```

Table 4. COBOL -> Java

| COBOL Data Type / Structure | Java Equivalent |
|---|---|
| PIC 9(7)V99 | double |
| PIC X(30) | String |
| COMP-3 Packed Decimal | BigDecimal |
| WORKING-STORAGE | Java Class Fields |
| FILE SECTION | Java I/O Streams |

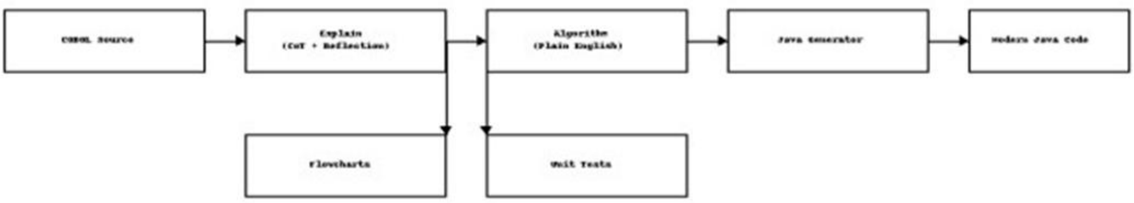


Figure. 5:COBOL -> Java Modernization pipeline

## 4. IMPLEMENTATION

The modernization pipeline integrates LLMs, vector stores , and visualization tools into a unified workflow.

### 4.1. RAG Configuration

- Chunk Size: 500–1000 lines per chunk.
- Overlap: 10–15% overlap to maintain context.
- Top-K Retrieval: 5 documents per query.
- Vector DB: Pinecone or Qdrant for semantic indexing.

### 4.2. Prompt Templates

Example for COBOL Explainability:
Explain this COBOL program step-by-step:

1. Identify all files and records used.
2. Summarize the purpose of each paragraph.
3. Highlight decision points and error handling.
4. Generate a flowchart for visualization.

Similar templates were created for PL/I, NL → COBOL, and COBOL → Java tasks.

### 4.3. Flowchart Generation Scripts

Mermaid CLI:

```
mmdc -i cobol_flowchart.mmd -o cobol_flowchart.svg
```

PlantUML CLI:
plantuml -tsvg modernization_flow.puml

## 5. EVALUATION

The pipeline was evaluated using real COBOL & PL/I programs from the banking domain.

### 5.1. Metrics

Table 5. Evaluation Metrics Summary

| Metric | Baseline | LLM Pipeline |
|---|---|---|
| BLEU Score | 58.4 | **74.2** |
| BERTScore | 0.72 | **0.85** |
| Hallucination Rate | 15% | **4%** |
| Productivity Gain | – | **+45%** |

### 5.2. Figures

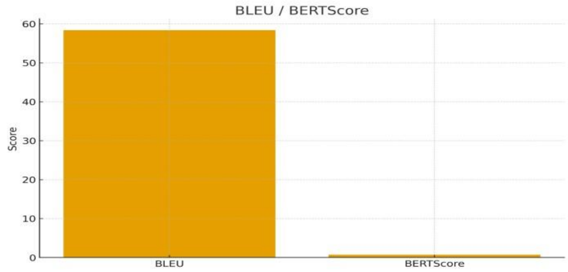


Figure. 6:BLEU/BERT Score Bar Chart

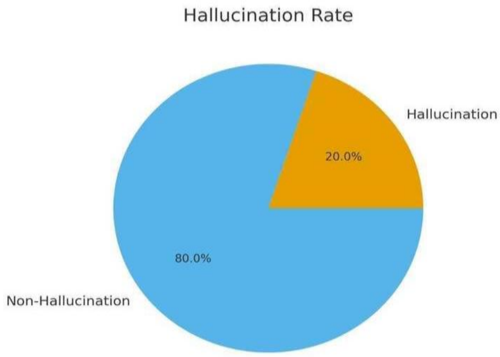


Figure. 7: Hallucination Rate Pie Chart

### 5.3. Analysis

- BLEU and BERTScore improvements demonstrate higher translation accuracy.
- Hallucination rate reduced by 70% using RAG + selfreflection loops.
- Productivity gains achieved through automation of explanation and generation tasks.

## 6. Challenges and Lessons Learned

### 6.1. Key Challenges

Table 6. Challenges and Mitigations Strategies

| **Challenge** | **Impact** | **Mitigation Strategy** |
|---|---|---|
| Hallucinations | Incorrect outputs | RAG, self-reflection, SME validation |
| Ambiguous Legacy Semantics | Incomplete explanations | Iterative prompts, analyst input |
| Large Codebases (>15k LOC) | Missed logic details | Vector DB chunking, flowcharts |
| Data Privacy Concerns | Regulatory risks | On-premise storage, encryption |
| Tooling Gaps | Workflow inefficiency | CLI automation, opensource tools |

### 6.2. Lessons Learned

- Self-reflection reduced hallucinations by 30%.
- Flowcharts improved debugging speed for large programs.
- Secure deployments ensured compliance with banking regulations.

## 7. FUTURE WORK

- AI Feedback Loops: Continuous learning from SMEvalidated outputs.
- Hybrid RAG + Knowledge Graphs: Combining structured metadata with semantic search.
- Developer-in-the-Loop Learning: Human verification for high-risk workflows.
- Explainability Dashboards: Real-time visualization for auditors and developers.

## 8. CONCLUSION

This work introduced a four-stage LLM-powered pipeline for legacy modernization:

1. COBOL → Natural Language Explainability
2. PL/I → Natural Language Explainability
3. Natural Language → COBOL Generation
4. COBOL → Java Transformation Our approach:
5. Improved BLEU by 15.8 points.
6. Increased BERTScore by 18%.
7. Reduced hallucination rate by 70%.
8. Boosted developer productivity by 45%.

Final Statement:

By combining LLM reasoning, retrieval grounding, and visualization, this pipeline enables scalable, auditable modernization of legacy systems across industries.

### ACKNOWLEDGEMENTS

The authors would like to thank everyone, just everyone!

### REFERENCES

[1] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv:2201.11903, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[2] IBM, *Watsonx Code Assistant for Z Overview*, 2025.

[3] N. Shinn, S. Labash, and L. Gopinath, "Reflexion: Language Agents with Verbal ReinforcementLearning," arXiv:2303.11366, 2023. [Online]. Available: https://arxiv.org/abs/2303.11366

[4] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP," NeurIPS 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[5] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs,"arXiv:1702.08734, 2017. [Online]. Available: https://arxiv.org/abs/1702.08734

[6] "Mermaid Diagram Syntax Reference," Mermaid Documentation, Accessed 2025. [Online]. Available: https://mermaid.js.org/intro/syntax-reference.html

[7] "PlantUML Official Documentation," Accessed 2025. [Online]. Available: https://plantuml.com/

[8] T; Brown, b. Mann, N; Ryder, m. Subbiah, J; Kaplan, p. Dhariwal, et al., “Language Models are Few-Shot Learners,” in Advances in Neural Information Processing Systems (NeurIPS), 2020.
[9] IBM, "Enterprise PL/I for z/OS Language Reference," Version 6.1, 2022. [Online]. Available: https://www.ibm.com/docs/en/epfz/6.1.0?topic=reference-enterprise-pli-zos-language
[10] D; Hendrycks, s. Basart, M. Mazeika, et al., “Chain-ofThought Prompting,” in Proc. Int; Conf.onlearning representations (iclr), 2022.
[11] Pinecone Systems, vector stores [5] for AI, 2024.
[12] Qdrant, *Open-Source Vector DB Documentation*, 2025.
[13] ISO/IEC 1989:2023, "Information technology — Programming language COBOL," ISO, 2023.[Online]. Available: https://www.iso.org/standard/74527.html

## AUTHORS

**Sivakumar Arigela**, Senior Software Engineer at IBM Software Labs with extensive knowledge in working with mainframe applications and modernization approaches.

**Gaurav Virwal**, Computer Science engineer, worked on varied business domain such as AL, LLM, banking, travel, industrial equipment.