Nice — let’s build a practical, open-source **COBOL → Java** conversion tool design plus a working starter scaffold you can iterate on. I’ll assume the common mainframe artefacts present in ~80% of enterprise systems: COBOL programs + copybooks, JCL, embedded DB2 SQL, VSAM/file I/O, CICS execs & BMS maps, and basic assembler/PL-I calls. The approach below is *IR-first* (recommended): parse → IR → annotate → generate Java + docs + tests.

I’ll give you:

1. architecture & rationale,
2. IR schema,
3. concrete open-source tech choices,
4. a minimal Python starter implementation (parser orchestration + simple IR emitter + Jinja2 Java generator) you can run and extend,
5. mapping rules and caveats, and
6. next steps & validation plan.

**Architecture (high level)**

scss

CopyEdit

[source folder: cobol, copybooks, jcl, sql, bms, logs]

│

▼

[Scanner & Preprocessor] → normalizes files, expands copybook refs (text level)

│

▼

[Parser / AST generator] → ANTLR (COBOL grammar) → ASTs

│

▼

[IR Builder] → produces structured JSON IR (programs, data defs, paragraphs, SQL, calls)

│

├─▶ [Analyzer / Annotator] → complexity, call graph, DB access, business-rule candidates (LLM optional)

│

▼

[Code Generator] → Jinja2 templates generate Java (domain, DAOs, services, DTOs, tests)

│

▼

[Validation/Test generator] → Unit tests, integration harness (dual-run), sample data

Why IR first: reuse, multi-target, explainability, easier to feed into LLMs for ambiguous logic.

**Open-source components to use**

* **Parser / Grammar**
  + ANTLR4 with a COBOL85 grammar (available in grammars-v4 repo) → generate a Python parser using antlr4-python3-runtime.
  + Alternative: OpenCobol / GnuCOBOL for quick tokenization/semantic checks.
* **IR / Processing & Generator**
  + Python 3.10+
  + antlr4-python3-runtime
  + jinja2 (templating Java)
  + networkx or graphviz for call graphs
  + sqlparse for embedded SQL snippets
  + black / google-java-format for Java formatting (post-gen)
* **Optional LLM assistance**
  + Gemini CLI or OpenAI GPT for business-rule summarization and ambiguous translations.
* **Testing**
  + pytest for Python parts
  + JUnit for generated Java tests (templates)

**IR design (JSON example)**

Make the IR rich but compact. Example for one program:

json

CopyEdit

{

"program\_id": "PAYROLL-CALC",

"source\_file": "cobol/payroll.cbl",

"loc": 1345,

"variables": [

{"name":"EMP-ID","pic":"9(6)","type":"NUMERIC","level":5,"scope":"WORKING-STORAGE"},

{"name":"SALARY","pic":"9(7)V99","type":"DECIMAL","scale":2}

],

"files": [

{"name":"EMPLOYEE-FILE","type":"VSAM","access":["READ","WRITE"],"ddname":"EMPFILE"}

],

"db2\_access": [

{"query\_type":"SELECT","tables":["EMP\_MASTER"],"sql":"SELECT ..."}

],

"paragraphs": [

{"name":"MAIN-LOGIC","statements":["READ EMPLOYEE-FILE","PERFORM CALC-TAX","IF ... MOVE ..."]}

],

"calls": ["CALC-TAX","CALC-BONUS"],

"business\_rules\_candidates": [

{"text":"If employee type is CONTRACT and hours>40 then overtime = true","confidence":0.86}

],

"metrics": {"cyclomatic":12,"max\_nesting":4}

}

Keep IR normalized: list of programs, copybook references resolved to data definitions, explicit DB accesses, call graph edges, mapping of PIC clauses.

**Key mapping rules (COBOL → Java)**

Data types:

* PIC 9(6) → int or long (based on length)
* PIC 9(7)V99 → BigDecimal with scale
* PIC X(n) → String
* Occurs clause → List<T> or generated POJO array
* Group items → nested POJO classes

Control flow:

* PARAGRAPH → Java void method (or service method)
* PERFORM → method call
* IF/EVALUATE → standard Java control flow
* READ/WRITE (VSAM) → DAO read/write methods (generate interfaces, then implement with adapters)

DB2 Embedded SQL:

* Map to JDBC / JPA (prefer JDBCTemplate for generated code), produce DAO with prepared statements.

JCL:

* Generate job metadata and identify batch entry points; model as main() classes or Spring Batch jobs.

CICS:

* Transform to REST controllers (Spring Boot) with DTOs mapped from BMS maps.

**Minimal runnable starter (Python) — scaffold + core scripts**

Below is a compact starter that:

* Scans a folder
* Splits copybooks and COBOL files
* Uses a lightweight regex-based extractor for variables & calls (starter for AST later)
* Emits IR JSON
* Generates a Java class from a simple Jinja2 template

This is intentionally simple — real parser (ANTLR) replaces the regex extractor for production.

**requirements.txt**

CopyEdit

antlr4-python3-runtime==4.11.1

jinja2

sqlparse

networkx

**project layout**

markdown

CopyEdit

cobol2java/

├── README.md

├── requirements.txt

├── scanner.py

├── ir\_builder.py

├── generator.py

├── templates/

│ └── java\_class.jinja

└── samples/

└── payroll.cbl

**scanner.py** (scans files & extracts text)

python

CopyEdit

import os

from pathlib import Path

def collect\_sources(root="samples"):

files = []

for p in Path(root).rglob("\*"):

if p.suffix.lower() in {".cbl", ".cpy", ".jcl", ".sql", ".bms"}:

files.append(str(p))

return files

if \_\_name\_\_ == "\_\_main\_\_":

print(collect\_sources("samples"))

**ir\_builder.py** (very small extractor)

python

CopyEdit

import re, json

from pathlib import Path

PIC\_RE = re.compile(r'PIC\s+([A9X9\(\)Vv9]+)', re.IGNORECASE)

PROG\_RE = re.compile(r'PROGRAM-ID\.\s\*([\w-]+)\.', re.IGNORECASE)

CALL\_RE = re.compile(r'\bCALL\s+"?([\w-]+)"?', re.IGNORECASE)

PERFORM\_RE = re.compile(r'\bPERFORM\s+([\w-]+)', re.IGNORECASE)

def build\_ir(file\_path):

text = Path(file\_path).read\_text()

prog = PROG\_RE.search(text)

prog\_id = prog.group(1) if prog else Path(file\_path).stem

pics = PIC\_RE.findall(text)

calls = CALL\_RE.findall(text)

performs = PERFORM\_RE.findall(text)

# quick LOC

loc = len(text.splitlines())

return {

"program\_id": prog\_id,

"source\_file": file\_path,

"loc": loc,

"pics": pics,

"calls": calls + performs

}

if \_\_name\_\_ == "\_\_main\_\_":

from scanner import collect\_sources

files = collect\_sources("samples")

irs = [build\_ir(f) for f in files]

print(json.dumps(irs, indent=2))

**templates/java\_class.jinja**

jinja

CopyEdit

package generated;

public class {{class\_name}} {

{% for field in fields %}

private {{field.type}} {{field.name}};

{% endfor %}

public {{class\_name}}() {}

{% for field in fields %}

public {{field.type}} get{{field.name|capitalize}}(){ return this.{{field.name}}; }

public void set{{field.name|capitalize}}({{field.type}} v){ this.{{field.name}} = v; }

{% endfor %}

}

**generator.py**

python

CopyEdit

import json

from jinja2 import Environment, FileSystemLoader

from ir\_builder import build\_ir

from scanner import collect\_sources

from pathlib import Path

env = Environment(loader=FileSystemLoader("templates"))

def pic\_to\_java(pic):

# naive mapping: if contains X -> String, if V -> BigDecimal, else int

if "X" in pic.upper(): return "String"

if "V" in pic.upper(): return "java.math.BigDecimal"

return "int"

def generate():

files = collect\_sources("samples")

for f in files:

ir = build\_ir(f)

fields = []

for i,p in enumerate(ir["pics"][:5]): # limited fields example

fields.append({"name": f"field{i+1}", "type": pic\_to\_java(p)})

tmpl = env.get\_template("java\_class.jinja")

class\_txt = tmpl.render(class\_name=ir["program\_id"].replace("-","\_"), fields=fields)

out = Path("out")

out.mkdir(exist\_ok=True)

Path(out / f"{ir['program\_id'].replace('-','\_')}.java").write\_text(class\_txt)

print("Generated:", ir['program\_id'])

if \_\_name\_\_ == "\_\_main\_\_":

generate()

Run:

bash

CopyEdit

python -m pip install -r requirements.txt

python generator.py

# check out/ for generated Java

This scaffold is intentionally small — replace the ir\_builder with an ANTLR-backed AST processor for production grade.

**Replacing regex with ANTLR (production parser)**

1. Get ANTLR4 and COBOL grammar (grammars-v4/cobol85).
2. Generate Python parser:

bash

CopyEdit

antlr4 -Dlanguage=Python3 cobol85.g4 -o parser

pip install antlr4-python3-runtime

1. Walk the tree to extract:
   * Program ID, divisions (DATA DIVISION STRUCTURE), PIC clauses, PERFORM/CALL, EXEC SQL blocks, COPY includes, file descriptors.

Use ANTLR listener/visitor pattern to populate the IR nodes instead of naive regex.

**Handling copybooks & includes**

* Preprocess: locate COPY 'X' lines, resolve to actual copybook text using a copybook path map.
* Inline or map copybook definitions in IR but maintain reusability: reference copybook id and include resolved data structures.

**Business rules extraction (LLM assisted)**

* For each paragraph/compound IF/EVALUATE, create a snippet and call LLM:
  + Prompt: “Extract business rule(s) from this COBOL paragraph and return structured JSON {condition, action}.”
* Attach results into business\_rules\_candidates in IR with confidence scores.
* Always include human-in-loop verification.

**Generating Java architecture (recommended target)**

Produce code patterns, not 1:1 line translations:

* Domain POJOs (from working-storage / file records)
* DAO layer (JDBC/DAO or Spring Data) for DB2/VSAM (adapter)
* Service layer (one service per COBOL program)
* Controller layer (for interactive CICS screens → REST controllers)
* Batch jobs (Spring Batch job per JCL job)

Prefer generating *clean, idiomatic* Java over exact line-for-line.

**Validation strategy (must have)**

1. **Unit tests**: Generate JUnit tests exercising the same input→output for small programs with sample data.
2. **Dual-run testing**: Run mainframe output vs new Java output on same dataset for parity.
3. **SME review**: Provide business-rule diffs to domain experts.
4. **Performance tests**: Some converted logic needs optimization (BigDecimal, loops).

**Limitations & caveats**

* Automated conversion cannot fully understand intent in all cases — you need SME validation.
* COBOL dialects vary (Enterprise COBOL, Micro Focus). ANTLR grammar may need dialect tweaks.
* I/O semantics (VSAM indexed vs KSDS) and transaction semantics in CICS require adapters, not simple code transforms.
* Embedded assembler or OS-specific macros may require manual re-implementation or wrappers.

**Roadmap & deliverables I can produce for you (pick any)**

* A full starter repo (the scaffold above expanded) including:
  + ANTLR build steps and Python AST visitor for COBOL.
  + Full IR JSON schema + sample IRs from provided files.
  + Jinja2 template suite for POJOs, DAOs, Services, Spring Boot app.
  + LLM prompts for rule extraction and integration hooks (Gemini/OpenAI).
  + CI pipeline (GitHub Actions) to run tests and generate artifacts.
* Or I can immediately generate the expanded Python project scaffolding (files above as real files) — want me to produce that here?

If you want, I’ll now:

* generate the **full starter repository** (expanded scripts, ANTLR integration guidance, Jinja templates) that you can clone and run, **or**
* produce a **detailed implementation plan** with required effort estimate and milestone tasks so you can staff it.

Which would you like me to do next?

can we model it as a multiagent system?

**ChatGPT said:**

Yes — in fact, modeling mainframe-to-Java migration as a **multi-agent system** is a solid approach, especially if you want automation, modularity, and adaptability.

Here’s how it can work step-by-step:

**1. Why Multi-Agent for Migration?**

Mainframe migration is not just “code translation.” You need:

* **Language parsing** (COBOL, JCL, etc.)
* **Business logic extraction**
* **Data structure mapping** (VSAM → RDBMS)
* **Integration mapping** (CICS, MQ, batch jobs)
* **Modernization patterns** (monolith → services)
* **Testing & validation**

Different agents can specialize in each area, work in parallel, and communicate findings.

**2. Proposed Multi-Agent Architecture**

**A. Input Collection Layer**

* **Artifact Collector Agent**
  + Scans repo/folders for .cbl, .jcl, .cpy, .pli, .map, .dbd, .psb files
  + Extracts metadata (file size, program name, dependencies)
  + Feeds into parsing agents

**B. Analysis Layer**

* **Code Parsing Agents**
  + **COBOL Parser Agent** – builds AST (abstract syntax tree) using open-source parsers like [antlr4-cobol](https://github.com/uwolfer/antlr4-cobol) or [Koopa](https://github.com/uwolfer/koopa).
  + **JCL Parser Agent** – maps batch jobs, step execution, dependencies.
  + **Copybook & Data Definition Agent** – translates COBOL copybooks into equivalent Java POJOs or database schemas.
* **Dependency Mapper Agent**
  + Maps cross-program calls, copybook usage, file/database dependencies.
* **Business Rules Extraction Agent**
  + Uses LLM to annotate important business rules in natural language.

**C. Transformation Layer**

* **Intermediate Representation (IR) Agent**
  + Converts AST + metadata into an intermediate JSON/YAML representation.
  + IR acts as a “pivot” format so you can support **multiple target languages** later.
* **Java Code Generator Agent**
  + Takes IR and produces Java code following enterprise patterns.
  + Uses templates for persistence, API, and batch services.
* **Database Migration Agent**
  + Converts VSAM/IMS definitions into relational schema DDL (PostgreSQL, Oracle, etc.)

**D. Validation & Refinement Layer**

* **Test Harness Agent**
  + Generates JUnit tests from COBOL test scenarios or sample data.
* **Refactoring Agent**
  + Runs static analysis (via SonarQube or OpenRewrite) to make the code cleaner.

**3. Example Tech Stack**

* **Parsing**: Koopa (COBOL), ANTLR4 grammars (JCL, PLI)
* **LLM Layer**: Gemini 1.5 Pro / GPT-4o for natural language + transformation logic
* **Agents**: LangGraph, CrewAI, or OpenAI Swarm
* **Version Control Integration**: Git hooks to process code in batches
* **Build System**: Maven or Gradle for generated Java projects

**4. High-Level Flow**

java

CopyEdit

Mainframe Artifacts Folder

↓

Artifact Collector Agent

↓

Parsing Agents (COBOL, JCL, Data)

↓

Intermediate Representation (IR)

↓

Java Code Generator + DB Migrator

↓

Test Harness + Refactoring

↓

Migration Package

**5. Why IR First?**

Direct COBOL → Java can be messy.  
COBOL → IR → Java:

* Keeps transformation rules clear
* Allows multi-target support (Java, Python, .NET)
* Easier to debug and test in stages

If you want, I can sketch a **first working prototype** of this multi-agent pipeline using **LangGraph + open-source COBOL parser** so you can feed your folder of mainframe code and get Java output.

Do you want me to prepare that blueprint next?

[ANTLR Magic — Developing Mainframe Language Applications Using Language Recognizer | by Azat Satklyčov | CodeX | Medium](https://medium.com/codex/antlr-magic-developing-mainframe-language-applications-using-language-recognizer-5262726e1e93)

ANTLR for cobol in Bing

[cschneid-the-elder/mapa: mainframe application portfolio analysis, antlr4 grammars and java for static analysis of cobol+cics+db2+sqlims+execdli, jcl](https://github.com/cschneid-the-elder/mapa)

[Mainframe Application Portfolio Analysis | by Craig Schneiderwent | Medium](https://medium.com/@craig.schneiderwent/mainframe-application-portfolio-analysis-3ee859c62225)

[ArturHD/cobol-to-java-transpiler: A Cobol to Java Transpiler](https://github.com/ArturHD/cobol-to-java-transpiler)

Sure! Here’s clear, leadership-friendly documentation to accompany the diagram with explanations for each layer and component. You can use this as a briefing note or slide notes.

**Mainframe to Java Migration: Multi-Agent Architecture Documentation**

**Overview**

This architecture models a modular, multi-agent system designed to automate the migration of enterprise mainframe applications (typically COBOL, JCL, VSAM, Copybooks) to modern Java applications. It leverages AI and traditional parsing technologies orchestrated by an AI Orchestrator (Google ADK or LangGraph) to enable scalable, maintainable, and transparent migration.

**Layers and Components**

**1. Parsing Layer**

Responsible for ingesting and syntactically analyzing raw mainframe artefacts. It converts code and metadata into structured, machine-readable formats for downstream analysis.

* **COBOL Parser (ANTLR Grammar)**
  + Parses COBOL source code using ANTLR-based grammar.
  + Extracts Abstract Syntax Tree (AST), program structure, variables, and control flow.
* **JCL Parser**
  + Parses Job Control Language scripts.
  + Identifies job steps, resources, and execution dependencies.
* **Copybook Parser**
  + Processes COBOL copybooks defining data structures.
  + Extracts detailed data layout and field metadata.
* **VSAM Schema Parser**
  + Parses VSAM dataset definitions.
  + Extracts data organization and access methods.

**2. Analysis Layer**

Performs deeper semantic analysis on parsed artefacts to extract business logic, dependencies, data mappings, and migration insights.

* **Dependency Mapper**
  + Builds call graphs and data flow maps.
  + Identifies code module dependencies, usage patterns, and sequence of execution.
* **Business Rules Extractor**
  + Analyzes ASTs and metadata using NLP and AI techniques.
  + Extracts explicit and implicit business rules embedded in code logic.
* **Data Model Mapping**
  + Maps legacy data structures to modern relational or NoSQL schemas.
  + Aligns VSAM and DB2 structures with Java persistence models.
* **Migration Report Generator**
  + Produces detailed reports capturing complexity, migration risks, dependencies, and estimated effort.
  + Provides actionable insights for planning and risk mitigation.

**3. Transformation Layer**

Converts analyzed artefacts and rules into deployable Java code and related artifacts, ready for testing and deployment.

* **Intermediate Representation (IR) Generator**
  + Converts parsed and analyzed data into an abstract intermediate format.
  + Serves as a neutral bridge between legacy code and target Java constructs.
* **Java Code Generator**
  + Translates IR into idiomatic Java code.
  + Applies best practices, design patterns, and integrates business rules.
* **Test Harness Generator**
  + Automatically generates unit and integration tests for migrated components.
  + Ensures behavior correctness and facilitates regression testing.
* **Deployment Package Builder**
  + Prepares deployable Java artifacts, including JARs, WARs, configuration files.
  + Supports integration into CI/CD pipelines.

**AI Orchestrator Agent (Google ADK / LangGraph)**

* **Role**
  + Coordinates all agents across layers.
  + Manages workflows, data exchange, error handling, and iterative improvements.
  + Leverages AI capabilities for intelligent decision-making, such as prioritizing complex modules, suggesting code optimizations, or identifying migration bottlenecks.
* **Benefits**
  + Enables scalable, modular, and maintainable migration processes.
  + Supports incremental migration and continuous validation.
  + Facilitates human-in-the-loop interventions where necessary.

**Inputs & Outputs**

* **Inputs**
  + Enterprise mainframe artefacts including COBOL source, JCL scripts, copybooks, VSAM definitions, DB2 DDL.
* **Outputs**
  + Fully migrated, maintainable Java application code.
  + Comprehensive migration reports outlining progress, risks, and recommendations.

**Summary**

This multi-agent architecture enables a **holistic, AI-driven migration** from mainframe to Java, minimizing manual effort and maximizing automation, transparency, and control. It modularizes complex tasks into specialized agents coordinated by a central orchestrator, paving the way for efficient enterprise modernization.