Leveraging Generative AI for Extracting Business Requirements from Legacy COBOL and PL/I Code

Abstract

Recovering business requirements from COBOL and PL/I portfolios is difficult because logic is scattered across interdependent programs and data definitions, and existing analyses seldom yield stakeholder-facing artifacts. We introduce an LLM-augmented reverse-engineering pipeline that provides deterministic parsing, schema-constrained LLM generation with bidirectional traceability to code. It couples grammar-based parsing and control-flow and data-flow analysis with a large language model to translate an enriched intermediate representation into structured specifications. This is not raw-code prompting or generic summarization, the novelty is the LLM-centered generation over an enriched IR, with structured JSON outputs and traceability for compliance-sensitive settings. The pipeline produces business requirements documents, explicit rule catalogs, end-to-end data lineage, create–read–update–delete matrices, and field-level source-to-target mappings, each linked to the supporting code. In a financial industry setting, containing 3.4M+ LoC including comments / 3.2M excluding comments of COBOL, the system achieves ***93%*** agreement with expert-authored business rules and reduces documentation effort by approximately ***70%***, as measured against manually produced requirement documents and rule sets. On the internal corpus spanning 3.4M lines across online, batch, and job control workloads, the approach yields approximately ***3.2–3.3×*** faster analysis while improving artifact consistency and traceability.

1 Introduction

This paper is concerned with the problem of automatically extracting high-quality business requirements from large-scale legacy COBOL (IBM, 2025a) and PL/I (IBM, 2025c) systems. These systems still underpin core processes in banking, insurance, government, and manufacturing, and encode decades of institutional knowledge in mainframe programs and data stores (Infosys, 2020). When organizations plan modernization, they need stakeholder-ready artifacts such as Business Requirements Documents (BRDs), rule catalogues, data lineage diagrams, and create–read–update–delete (CRUD) matrices. Today, however, these artifacts are rarely available, and their reconstruction from code is a major bottleneck for modernization programmes (Sneed and Verhoef, 2019; Comella-Dorda et al., 2000).

This problem is important both for software engineering and for natural language processing. Failure to recover "lost" business rules leads to defects, compliance risks, and costly rework when replaced systems behave differently from the legacy implementation. From an NLP and AI perspective, the task requires transforming low-level mainframe code into coherent, multi-view natural language specifications, thereby testing whether large language models (LLMs) (Zhang et al., 2024) can bridge the gap between procedural code and business-level abstractions at portfolio scale (Hemmat et al., 2025; Zadenoori et al., 2025).

Automatically deriving such artifacts from COBOL and PL/I is challenging for several reasons. Legacy landscapes routinely comprise tens of millions of lines of code, spanning batch and online programs, CICS and JCL orchestration, and heterogeneous data stores such as VSAM and DB2 (IBM, 2025b), with high cyclomatic complexity (McCabe, 1976) and dense inter-program dependencies. Business logic is often scattered across conditionals, table lookups, and copybooks, with domain semantics only implicitly represented in naming conventions and data encodings. Prior applications of off-the-shelf code summarization (Ahmad et al., 2020; Sun et al., 2025) or requirements-extraction models (Miskell et al., 2023; Luttmer et al., 2023) tend to operate at the level of single functions or files, lack global context, and do not

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 43–55

produce the structured, traceable artifacts that product owners and analysts require.

Existing work has not fully addressed these challenges. Traditional business rule extraction pipelines for COBOL rely on static analysis to build control-flow and data-dependency models and then mine candidate rules (Huang et al., 1996; Edwards and Munro, 1993), but typically culminate in technical views or rule lists that still demand substantial expert interpretation; most commercial offerings remain black-box and are evaluated on narrow metrics or small systems (SoftwareMining, 2025; Sasasystems, 2025; Datamatics, 2025; Micro Focus, 2025). In parallel, research on automated requirements extraction with NLP and LLMs generally assumes natural language inputs such as specifications or user stories (Zadenoori et al., 2025; Wang et al., 2025), while studies of LLMs for code understanding focus on modern languages and snippet-level summarization (Jelodar et al., 2025; Zhang et al., 2024). Taken together, these strands provide limited guidance on how to use LLMs to generate BRDs, rule catalogs, lineage diagrams, and CRUD matrices from legacy mainframe portfolios with end-to-end traceability.

To the best of our knowledge, this is the first study on using large language models for extracting BRDs and related business artifacts from industrial-scale COBOL and PL/I systems. The main contributions of the proposed system as part of this research are as follows:

1. Automatically parses COBOL and PL/I code and generates BRDs.
2. Extracts and outlines business rules embedded within the code.
3. Performs lineage analysis for data and system dependencies.
4. Constructs CRUD matrices and source-to-target data mappings to provide insights into the system's structure.
5. Reduces the time and complexity of translating legacy code into actionable business requirements.

2 Methodology

2.1 Architecture Overview

The tool comprises the following high-level components:

1. **Interfaces:** A GUI lets users upload COBOL/PL/I source and dependencies (zipped raw files or compiler listings). An integrated chatbot answers user questions after processing is complete.
2. **Source Code Processors:** Use ANTLR (Parr, 2025) and an AST Builder to derive syntax trees, control flow, and data flow from COBOL/PL/I. COBOL code is parsed via a COBOL grammar: the lexer tokenizes statements, the parser builds a parse tree, creating a structured, machine-readable representation for downstream analysis.
3. **Data Ingress Processors:** Validate, enrich, and store data in a database before sending it to the LLM. The token/parse tree is provided additional context, architecture info, adding metadata and serializing (e.g., JSON/XML) and passed via API.
4. **App Services:** Orchestrate interactions with the LLM using the enriched payload. The request is first authenticated via LLM API, JSON is embedded and prompt construction with task instructions are added with response handling to format.
5. **LLM Interpreter:** – Uses the enriched COBOL/PL/I representation to generate plain-language descriptions of business logic, rules, and system operations.

2.2 Input

COBOL programs are classified by their execution environment and interaction method. The primary types are Batch and Online, with CICS and JCL being the technologies/environments used to facilitate these modes of operation on mainframe systems.

For this study, we curate a mixed portfolio representative of typical financial-services mainframe estates: online transaction programs, batch workloads, CICS transactions, and JCL orchestration. The evaluation corpus contains approximately 3.4M lines of COBOL/JCL (Table 1) and associated copybooks, SQL (DB2) schemas, and interface metadata. To protect confidentiality, identifiers (program names, dataset names, table/column names, and literal business labels) are pseudonymized, while preserving structure, control-flow, and data-flow relationships required for analysis. Language semantics and compilation

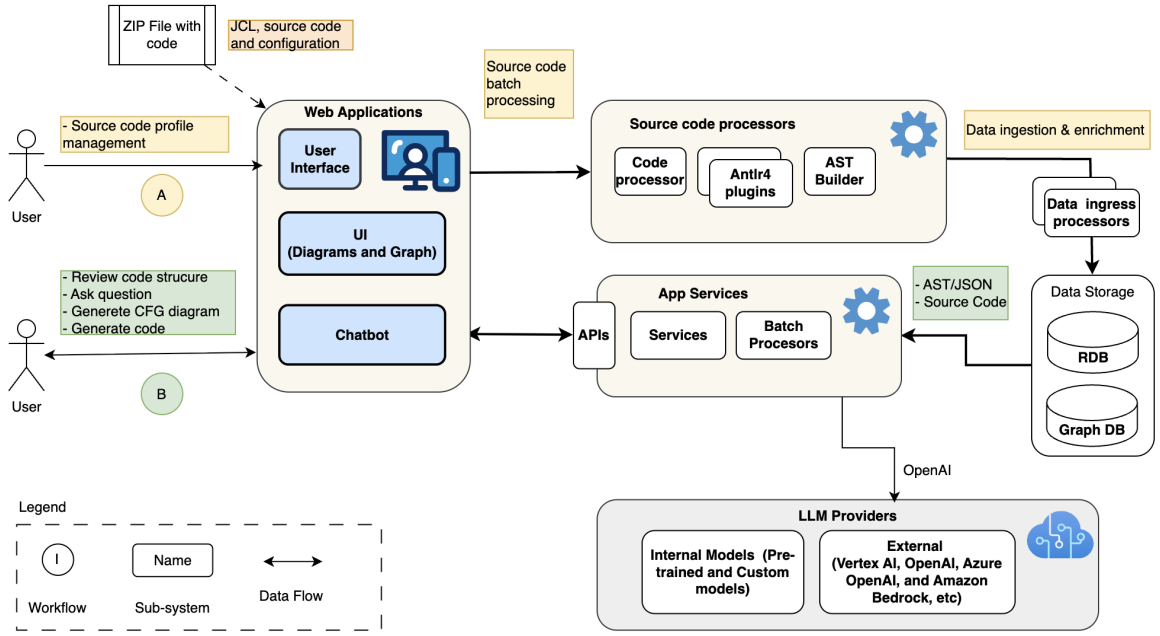

Figure 1: Architecture Overview

conventions follow IBM Enterprise COBOL and IBM PL/I references(IBM, 2025a,c).

COBOL Program Manual Analysis usually covers the following:

1. *Business Logic & Use:* Identifies what the COBOL program does end-to-end (e.g., calculate interest, post transactions) and when/why it is invoked in the business process.

2. *Business Rules within the program:* Extracts decision logic (IFs, EVALUATEs, validations, limits, thresholds) that govern how inputs are checked, transformed, or rejected.

3. *External References (DB Tables, MQ, Call chain, CICS, Files):* Lists and maps all external dependencies the COBOL code uses, such as DB2 tables, MQ queues, called subprograms, CICS transactions, and VSAM/flat files.

4. *Dead Code Analysis:* Detects statements, paragraphs, or branches that are never executed under any valid path or condition, indicating removable or obsolete logic.

5. *Lineage Analysis:* Traces how key data elements flow from input sources through intermediate fields and transformations to final outputs and downstream systems.

To generate the same using an automated tool which works as a combination of deterministic and generative and makes use of the LLM, the workflow is as below:

1. *Input:* COBOL and PL/I source code is fed into the system.

2. *Parsing:* The COBOL and PL/I parser identifies program modules, data definitions, and procedural flows.

3. *LLM Analysis:* The parsed code is processed by the LLM, which translates technical syntax into human-readable requirements and rules.

4. *Artifact Generation:* The resulting information is synthesized into structured documents.

5. *Output:* Provides detailed BRDs, lineage analysis, CRUD matrices, and source-to-target mappings.

2.3 Output Artifact Generated

1. *Business Logic and Usage:* Summarizes extracted information on the UI with defined sections of a business requirement. The LLM's natural language summary provides what and why of the COBOL program.

2. *Business Rules:* Identifies operational constraints, conditions, and policies. By analyzing the AI-generated explanation, the application can extract explicit business logic.

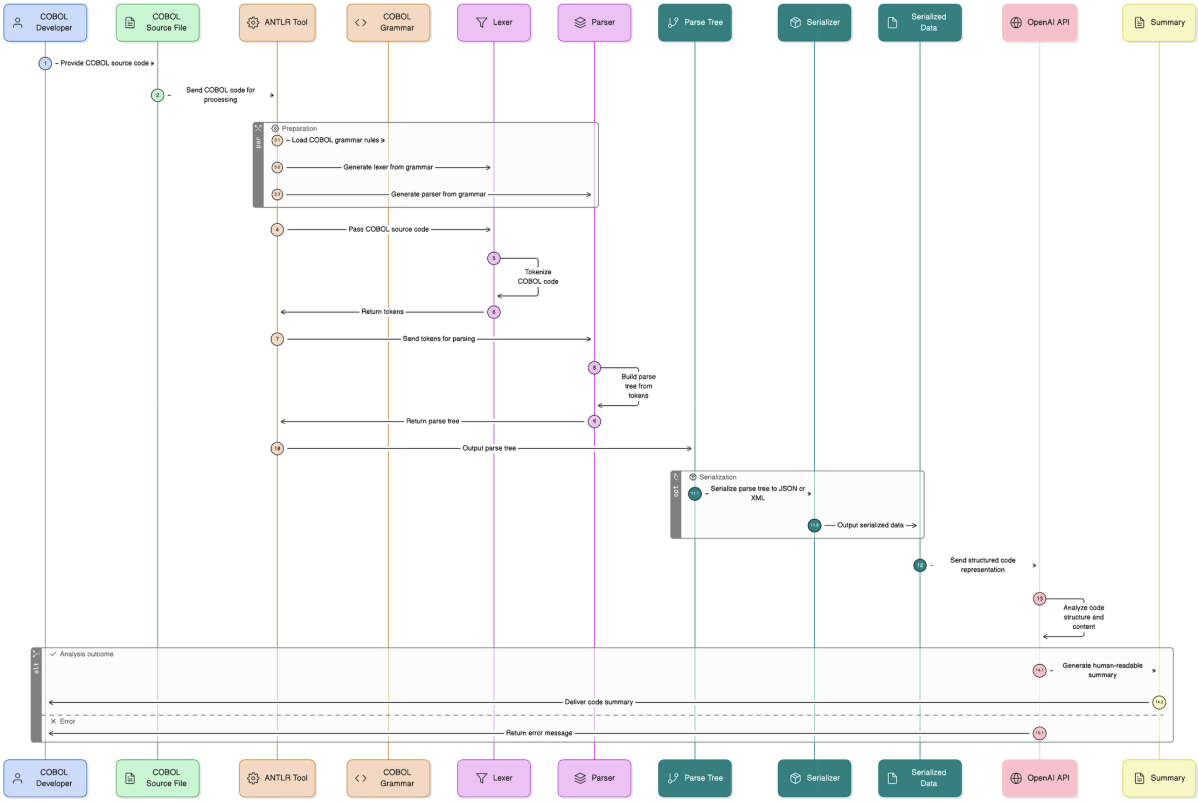

Figure 2: Workflow

3. *Lineage Analysis:* Tracks dependencies and relationships between program components. The LLM output explains how data flows and is modified.

4. *Source to Target Mapping:* Highlights create-read-update-delete operations and data transformations in a graph structure to represent the flow (Neo4j, 2025).

5. *External Dependencies:* Provides external dependencies of SQL calls, DB2 Tables, CICS, Files and MQ. It provides the details around each of the external references with data points as inputs and outputs and how it is referenced in the program.

6. *Interactive Chat Capabilities:* Once the program is traversed deterministically with AST generated and then generatively for each core function through LLM, the output is stored in a persistent storage and tool provides interactive capabilities to ask questions about anything specific to the programs.

2.4 Prompting, Structured Outputs, and Requirement Templates

We formulate requirement extraction as a set of schema-driven generation tasks. Each task requests a specific BRD section (e.g., Purpose, Inputs/Outputs, Business Rules, Exceptions, External Dependencies, CRUD, and Data Mapping) and requires the model to emit JSON following a fixed schema. The model receives structured static-analysis facts (calls, DB2/MQ/VSAM usage, copybooks, field flows, dependencies), not raw code alone; generation is schema-constrained; separate calls are used for BRD summarization, rule extraction, capability mapping, and chat explanation. Using structured outputs reduces ambiguity and allows deterministic post-processing (validation, deduplication, traceability linking). In our implementation, the orchestration layer uses the OpenAI Chat Completions endpoint and tool/function calling style schemas to constrain output and to attach metadata such as file identifiers, paragraph labels, and referenced data elements (OpenAI, 2025a,b).

2.5 Post-processing and Traceability Linking

After generation, we apply rule-based validators to ensure (i) required fields are present, (ii) enu-

merations conform to controlled vocabularies (e.g., rule types, CRUD operations), and (iii) references to programs, paragraphs, copybooks, and data elements exist in the parsed symbol tables. We then produce a traceability map that links each generated requirement sentence or rule back to the supporting code locations (procedure division blocks, data division items, SQL statements). This enables analyst review workflows, targeted re-prompting, and audit-friendly evidence trails.

2.6 Deployment, Runtime and Cost

Pilot deployed on customer infrastructure, Konvoy-based Kubernetes / VM cluster, about 4 LLM calls per program, initial long generations hit 2-minute timeout, streaming resolved it, and chatbot is DB-first with LLM invoked only for extra reasoning/summarization

2.7 Human-in-the-loop Review Workflow

Because legacy code often encodes implicit domain assumptions, we incorporate a lightweight human review loop: (1) the tool proposes BRD sections with confidence signals (coverage and rule density), (2) analysts accept/edit items in a review UI, (3) corrections are fed back as few-shot exemplars or glossary constraints for subsequent runs. This workflow is designed to minimize manual reading of raw source while retaining accountability for final requirements.

3 Evaluation

3.1 Dataset and Application

As part of the financial industry use case with total of 3.4+ million LoC of COBOL, 881 total programs, 158 sampled programs (18%), chosen with SMEs to cover Batch COBOL, Online COBOL, VSAM, IBM MQ, DB2, Infopak, and high-complexity programs. Then define the gold set: 984 business rules, 120 BRD items, 38 common COBOL modules. The pipeline was utilized to perform the reverse engineering on a program, the task was to take the BRD outcome to be taken up by product owner for forward engineering and product development on the open-source platform. The results were encouraging, and it improved the efficiencies of manual reverse engineering by 93% precision against the manual analysis of the code, sampled rules were judged correct by two SMEs under rubric R; disagreements adjudicated by third reviewer.

3.2 Key Performance Indicators (KPIs)

We evaluate the proposed approach using a balanced set of productivity and quality KPIs:

1. *Throughput/Effort-Reduction:* analyst-hours per 100K LoC; speedup vs. manual baseline.
2. *Requirements fidelity:* correctness and completeness of extracted BRD items against a human-created gold set based on table 1, authored by mainframe SMEs, total of 881 programs.
3. *Traceability coverage:* percentage of generated requirements/rules linked to at least one code or data element.
4. *Rule extraction quality:* precision/recall of detected business rules and validations (sampled and adjudicated).
5. *Lineage and dependency accuracy:* agreement on external calls, files, queues, and DB2 table usage.
6. *Portfolio-level operational errors:* Average error rate at portfolio level.

3.3 Results

The results are categorized against the following criteria in Table 2.

1. *Manual Analysis Errors:* Number of errors while doing the manual analysis of the code set.
2. *Tool Analysis Errors:* Number of errors found using the automated tool
3. *Error/LoC (Manual):* Number of errors divided by total LoC when analyzed manually.
4. *Error/LoC (Tool):* Number of errors divided by total LoC when analyzed using tool.
5. *Error % Manual vs Tool:* Error rate Manual minus Error Rate Tool (C – D)
6. *Average Error Rate:* Average Error Rate across all Types (Online, Batch, CICS, JCL) of programs. AVERAGE(D)
7. *Average Error Rate:* Average Error Rate across all Types (Online, Batch, CICS, JCL) of programs. Average(E)

	Online COBOL	Batch COBOL	CICS	JCL
LoC	1.7M LoC	1.1M LoC	480K LoC	122K LoC
Cyclomatic Complexity	39	42	30	38
Data Structures	VSAM, DB2	VSAM	VSAM	Control Cards
Data Integrity & Relationships	High	High	Medium	Low
Copybooks usage	Yes	Yes	No	No
Domain Data Ownership	Low	Medium	Low	Low

Table 1: Evaluation Data

Metric	Online COBOL	Batch COBOL	CICS	JCL
LoC	1.7M LoC	1.1M LoC	480K LoC	122K LoC
Manual Analysis Errors	2230	1789	898	575
Tool Analysis Errors	2398	1935	1022	595
Error/LoC Manual	0.13%	0.16%	0.19%	0.47%
Error/LoC Tool	0.14%	0.18%	0.21%	0.49%
Error % Manual vs Tool	0.01%	0.01%	0.03%	0.02%
Average Error Rate		0.25%		
Average Error Rate vs Manual		0.02%		

Table 2: Portfolio-level error analysis.

Workload	LoC	(M) LoC/hr	(T) LoC/hr	Speedup (×)	(M) Hrs.	(T) Hrs.	Effort Saving
Online	1,700,000	312.5	1042.9	3.34	5440	1630	70%
Batch	1,100,000	312.5	1000.0	3.20	3520	1100	69%
CICS	480,000	312.5	1021.3	3.27	1536	470	70%
JCL	122,000	312.8	1016.7	3.25	390	120	70%

Table 3: Comparative analysis of manual vs. automated tool throughput and speedup.

We achieve significant time savings, reducing documentation efforts by **70%** as compared to manual analysis of the above code base. Derived throughput from Table 3 indicates an average manual pace of $\sim$ 313 LoC/hour across categories, while the automated pipeline achieves $\sim$ $1,000$ – $1,043$ LoC/hour, corresponding to ~ 3.2 – 3.3 speedup depending on workload type (online, batch, CICS, JCL).Positive feedback from business analysts is also provided, emphasizing the readability of generated artifacts.

3.4 Failure Analysis

Missing rules, incorrect primary business-capability assignment, and missing lineage links. Human correction was typically needed and that prompt refinements improved these areas. Informal LLM-only trials had more hallucinations, higher token cost, and weaker structural coverage, which is why deterministic parsing remains first-stage.

4 Conclusion

Coupling deterministic legacy-code analysis with OpenAI LLM–based synthesis can reliably generate business-facing artifacts from COBOL and PL/I— business process flows, lineage, CRUD matrices, business rules, and source-to-target mappings, external dependencies (files, MQ, copybooks, CICS). By grounding outputs in parsed code structure and linking each statement to concrete evidence, the pipeline improves explainability and enables essential human-in-the-loop validation for compliance-sensitive requirements. It also strengthens traceability by keeping requirements, rules, and data mappings continuously tied to the exact program and field locations that produced them.

Results show major productivity gains (70% effort reduction and 3–4× throughput) while maintaining strong alignment with expert-authored rule sets. The code-to-specs parser further supports building scalable applications from extracted requirements before rewriting in open-source languages, reducing downstream refactoring and hard-

ening costs and improving time to market. Future work will refine accuracy, expand language/support coverage, add domain customization, and evolve toward an agentic workflow that composes validated artifacts into implementation-ready low-level designs and API specifications.

This is a human-in-the-loop modernization accelerator, with the main current limitations being capability mapping and occasional missed rules, not a fully autonomous replacement for analysts.

5 Ethical Considerations

This work analyzes legacy enterprise code and related artifacts to generate stakeholder-facing outputs such as business rules, data lineage, CRUD matrices, and source-to-target mappings. Because these inputs may contain sensitive business logic and proprietary definitions, the pipeline is designed to reduce confidentiality risk and preserve human accountability.

1. **Data confidentiality and access control:** All analyses run in an enterprise-controlled environment with role-based access, audit logging, and least-privilege controls. Inputs and outputs are treated as confidential, stored in approved repositories, and shared only with authorized stakeholders. For reporting or publication, program identifiers, schema and field names, and business labels are pseudonymized or aggregated to protect proprietary information and client identity.

2. **Handling sensitive content and privacy:** Legacy code is usually not personal data, but it may contain identifiers, comments, or field names tied to individuals or regulated attributes. To reduce risk, we sanitize inputs before LLM processing where feasible by redacting known identifiers and removing non-essential comments. Extracted examples are restricted to non-identifying, synthetic, or heavily masked snippets.

3. **Model security and prompt-injection resilience:** LLM components can be affected by adversarial strings in comments or literals, such as prompt-injection or secret-exfiltration instructions. We mitigate this by enforcing schema-bound outputs, isolating untrusted text where possible, validating results against deterministic analysis signals, and applying post-generation checks to flag or reject disallowed content.

4. **Reliability, hallucination, and accountability:** Generated artifacts may include omissions or incorrect inferences. To reduce risk, the pipeline provides traceability from each extracted rule, entity, and lineage link to supporting code and intermediate outputs. Human-in-the-loop review remains mandatory, with analysts and SMEs validating, correcting, or rejecting outputs before use. The system is an assistive accelerator for analysis, not an autonomous authority.

5. **Fairness and downstream impact:** Extracted business requirements may reflect outdated or inequitable legacy policies. Since the task is descriptive, recovered rules should be treated as review candidates, not approved policies. Organizations should validate them through domain review, policy governance, and legal or compliance checks before implementing them in new platforms.

6. **Reproducibility under proprietary constraints:** Industrial datasets are often confidential and cannot be shared. To support transparency, we report the methodology, evaluation process, and annotation rubrics, and recommend releasing anonymized schemas, validation logic, and synthetic examples that reflect real inputs. This supports assessment and partial replication while preserving confidentiality.

7. **Environmental considerations:** LLM inference can add compute and energy costs. We reduce overhead by using deterministic parsing and static analysis to narrow LLM calls to relevant code slices, and by using structured prompts to limit retries. Organizations should also monitor usage, cache intermediate outputs, and choose model sizes that balance accuracy and latency.

8. **Limitations:** Despite these safeguards, risks remain: outputs may still contain subtle errors, sanitization may overlook sensitive data, and traceability alone cannot ensure correctness. We therefore recommend combining automated checks, strict access controls, and mandatory human review for any artifact used in production modernization decisions.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4998–5007, Online. Association for Computational Linguistics.

Santiago Comella-Dorda, Kurt Wallnau, Robert Seacord, and John Robert. 2000. A survey of legacy system modernization approaches. Technical Report CMU/SEI-2000-TN-003.

Datamatics. 2025. Modernize legacy code with KaiBRE. https://www.datamatics.com/resources/case-studies/demos/modernize-legacy-code-with-kaibre. Accessed: 2025-12-10.

Helen M Edwards and Malcolm Munro. 1993. Recast: Reverse engineering from cobol to ssadm specification. In *Proceedings of 1993 15th International Conference on Software Engineering*, pages 499–508. IEEE.

Arshia Hemmat, Mohammadreza Sharbaf, Shekoufeh Kolahdouz-Rahimi, Kevin Lano, and Sobhan Y Tehrani. 2025. Research directions for using llm in software requirement engineering: A systematic review. *Frontiers in Computer Science*, 7:1519437.

Hai Huang, Wei-Tek Tsai, Sourav Bhattacharya, XP Chen, Yamin Wang, and Jianhua Sun. 1996. Business rule extraction from legacy code. In *Proceedings of 20th International Computer Software and Applications Conference: COMPSAC'96*, pages 162–167. IEEE.

IBM. 2025a. COBOL on z/OS. IBM Documentation. Accessed: Nov. 2025.

IBM. 2025b. Db2 for z/OS: Referential constraints. IBM Documentation. Accessed: Nov. 2025.

IBM. 2025c. PL/I for MVS & VM documentation library. IBM Support. Accessed: Nov. 2025.

Infosys. 2020. Business rules extraction for mainframe applications. White paper, Infosys. Accessed: 2025-12-10.

Hamed Jelodar, Mohammad Meymani, and Roozbeh Razavi-Far. 2025. Large language models (llms) for source code analysis: applications, models and datasets. *arXiv preprint arXiv:2503.17502*.

Janosch Luttmer, Vitalijs Prihodko, Dominik Ehring, and Arun Nagarajah. 2023. Requirements extraction from engineering standards–systematic evaluation of extraction techniques. *Procedia CIRP*, 119:794–799.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Micro Focus. 2025. Business rule manager. Micro Focus Product Documentation. Accessed: 2025-12-10.

Cameron Miskell, Richard Diaz, Parth Ganeriwala, Khaled Slhoub, and Fitzroy Nembhard. 2023. Automated framework to extract software requirements from source code. In *Proceedings of the 2023 7th International Conference on Natural Language Processing and Information Retrieval*, pages 130–134.

Neo4j. 2025. What is data lineage? Neo4j Blog. Accessed: Nov. 2025.

OpenAI. 2025a. API reference (introduction). OpenAI Documentation. Accessed: Nov. 2025.

OpenAI. 2025b. Function calling. OpenAI Guides. Accessed: Nov. 2025.

Terence Parr. 2025. ANTLR parser generator. Official ANTLR Website. Accessed: Nov. 2025.

Sasasystems. 2025. COBOL RuleXtract: Automated business rule extraction for legacy COBOL. https://www.sasasystems.ai/cobol-rulextract/. Accessed: 2025-12-10.

Harry M Sneed and Chris Verhoef. 2019. From cobol to business rules—extracting business rules from legacy code. In *Integrating Research and Practice in Software Engineering*, pages 187–208. Springer.

SoftwareMining. 2025. COBOL business rule extraction. https://softwaremining.com/services/COBOL-Business-Rule-Extraction.jsp. Accessed: 2025-12-10.

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source code summarization in the era of large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1882–1894. IEEE.

Lei Wang, Ming-Chao Wang, Yuan-Rong Zhang, Jian Ma, Hong-Yu Shao, and Zhi-Xing Chang. 2025. Automated identification and representation of system requirements based on large language models and knowledge graphs. *Applied Sciences*, 15(7):3502.

Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. 2025. Large language models (llms) for requirements engineering (re): A systematic literature review. *arXiv preprint arXiv:2509.11446*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Transactions on Machine Learning Research*.

Appendix (Sample Outputs)

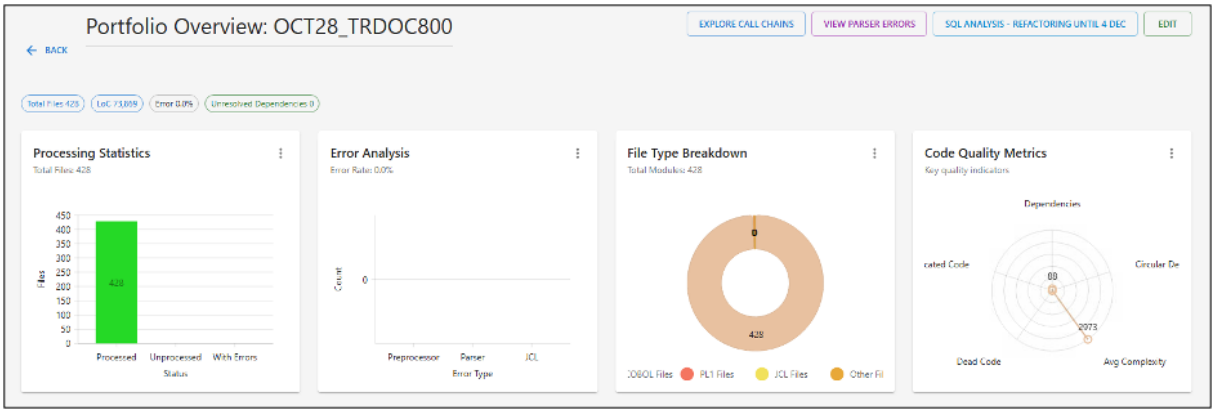

Figure 3: Portfolio Analysis

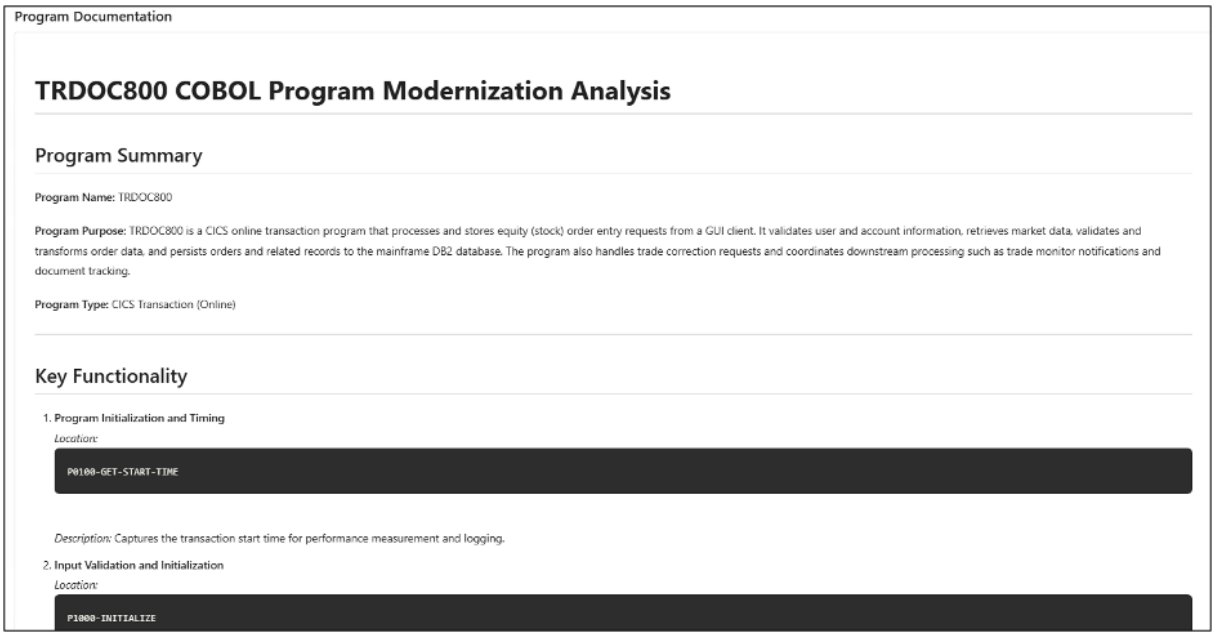

Figure 4: Business Requirements

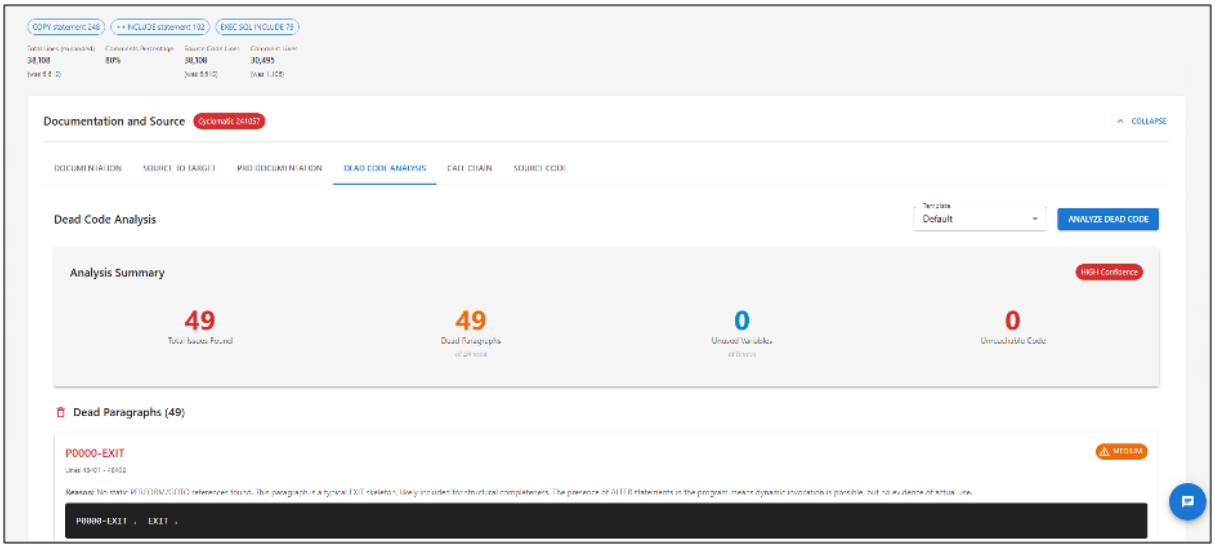

Figure 5: Dead Code Analysis

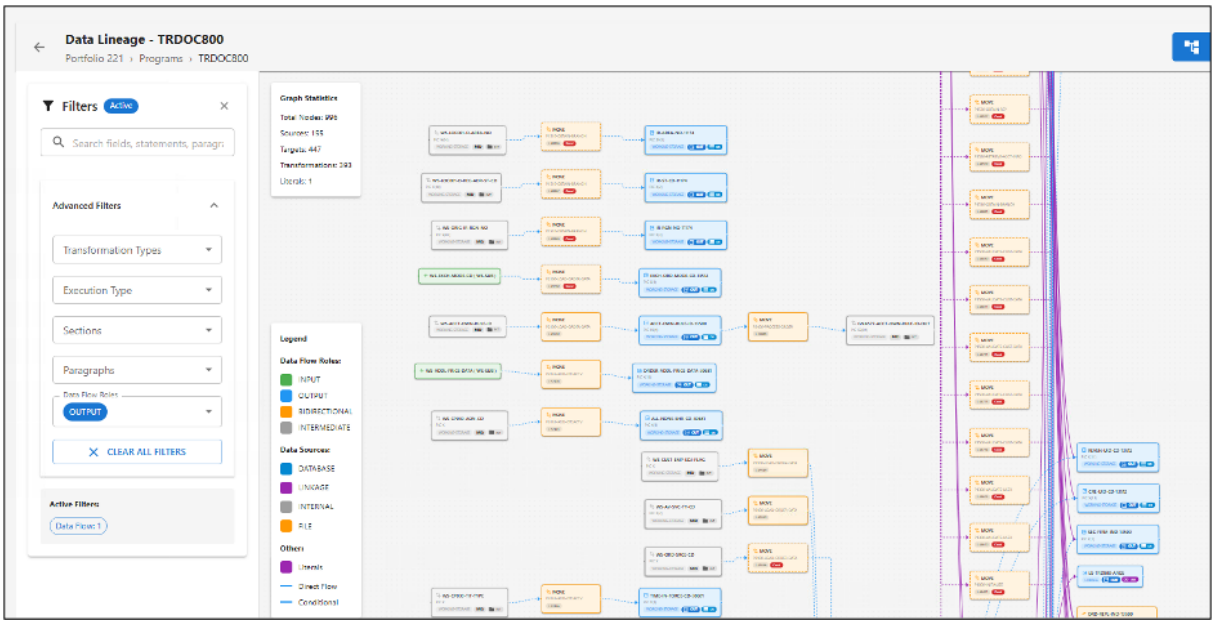

Figure 6: Lineage Analysis

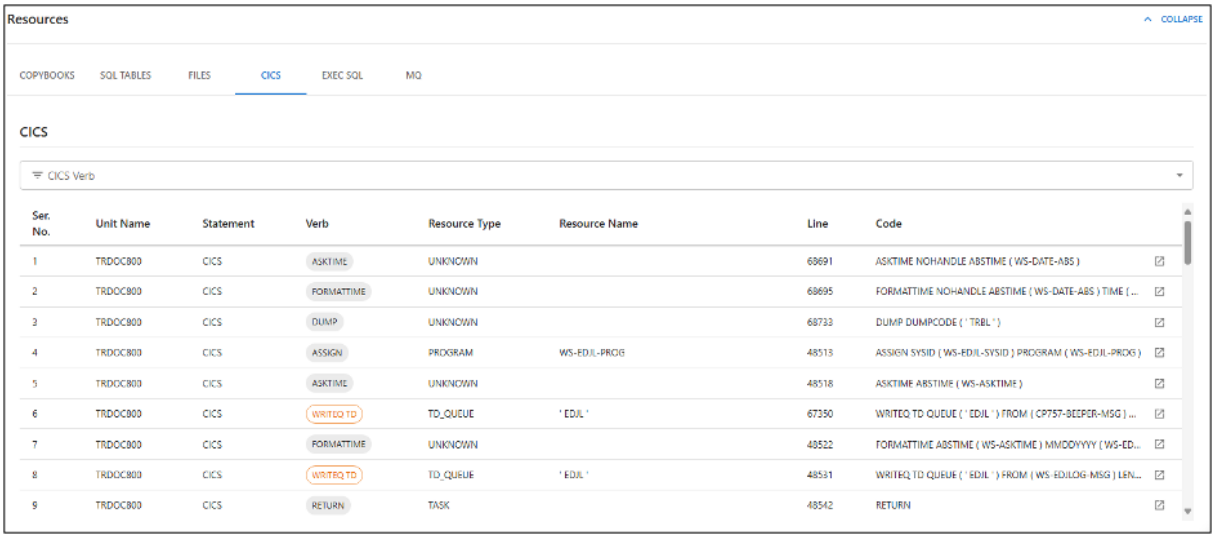

Figure 7: External Dependencies CICS

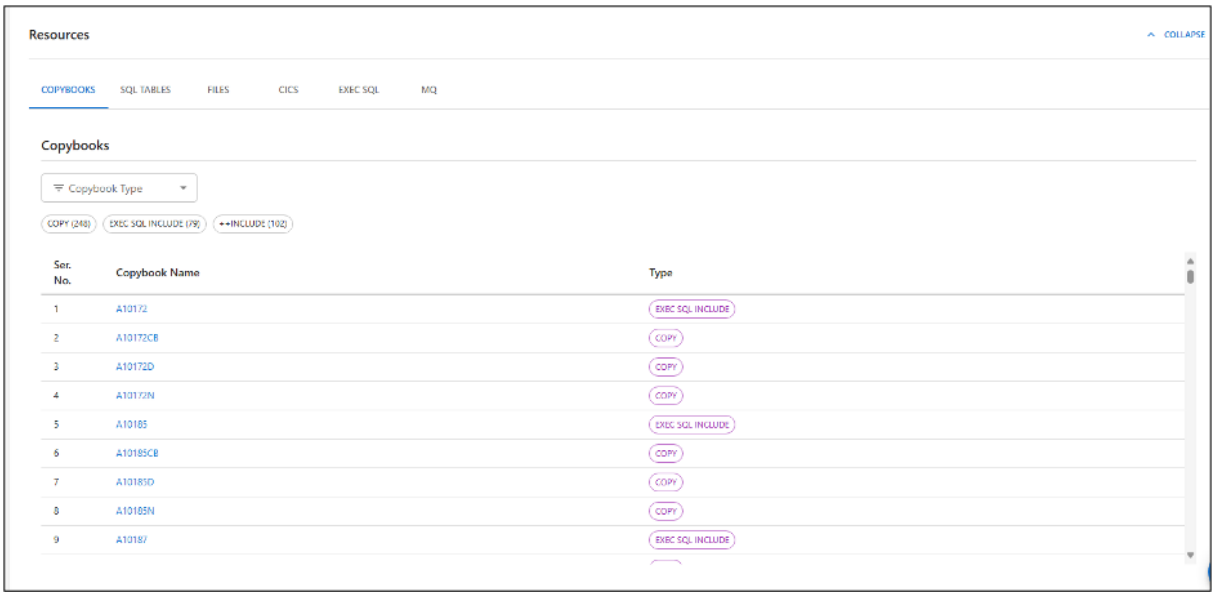

Figure 8: External Dependencies CopyBooks

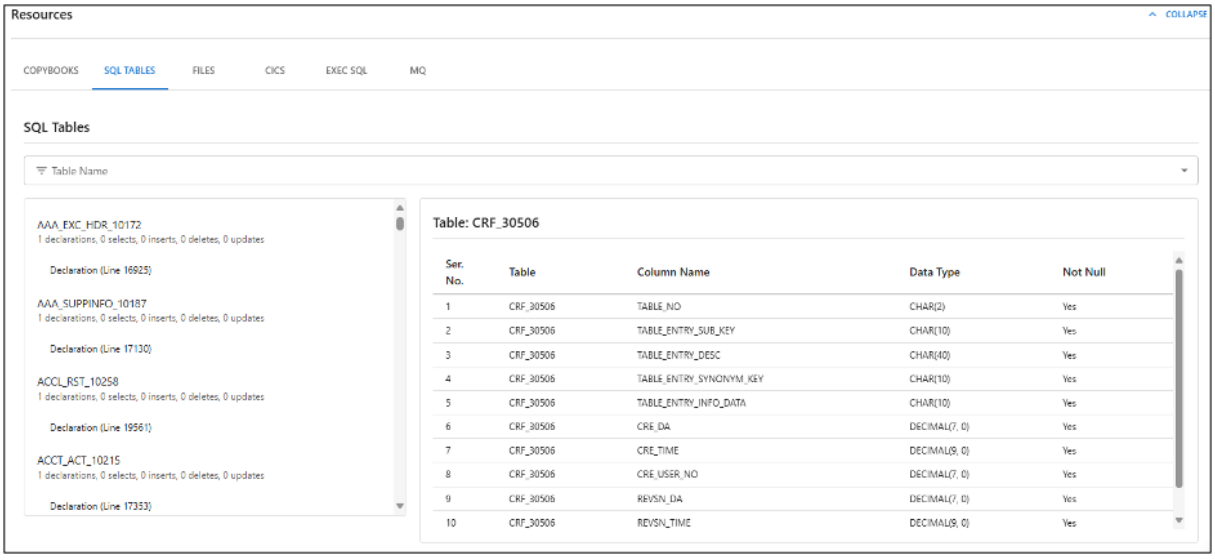

Figure 9: External Dependencies SQL

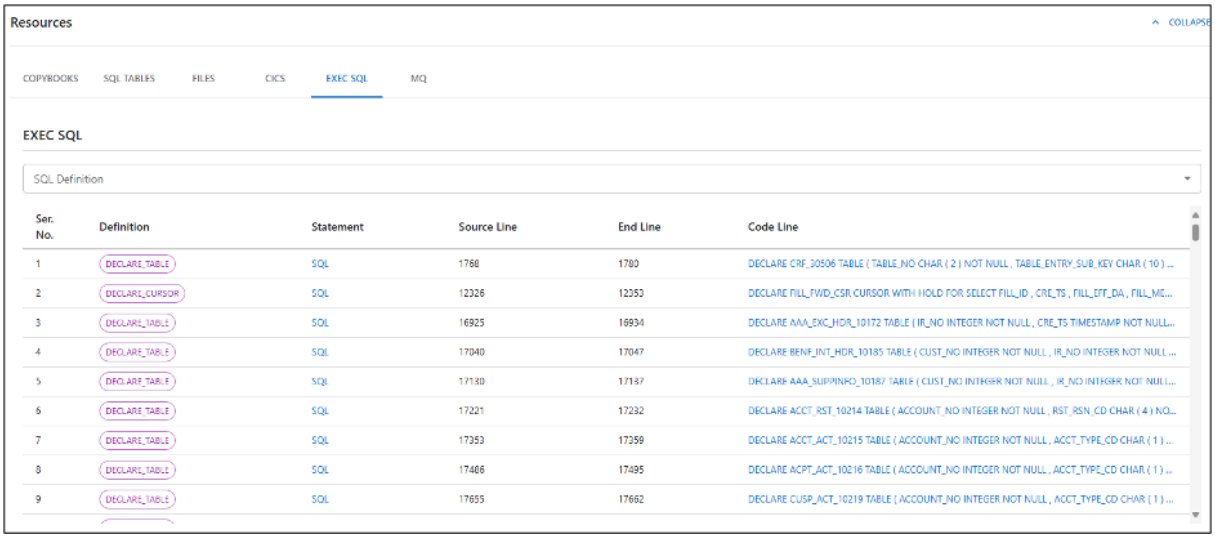

Figure 10: External Dependencies Exec SQL

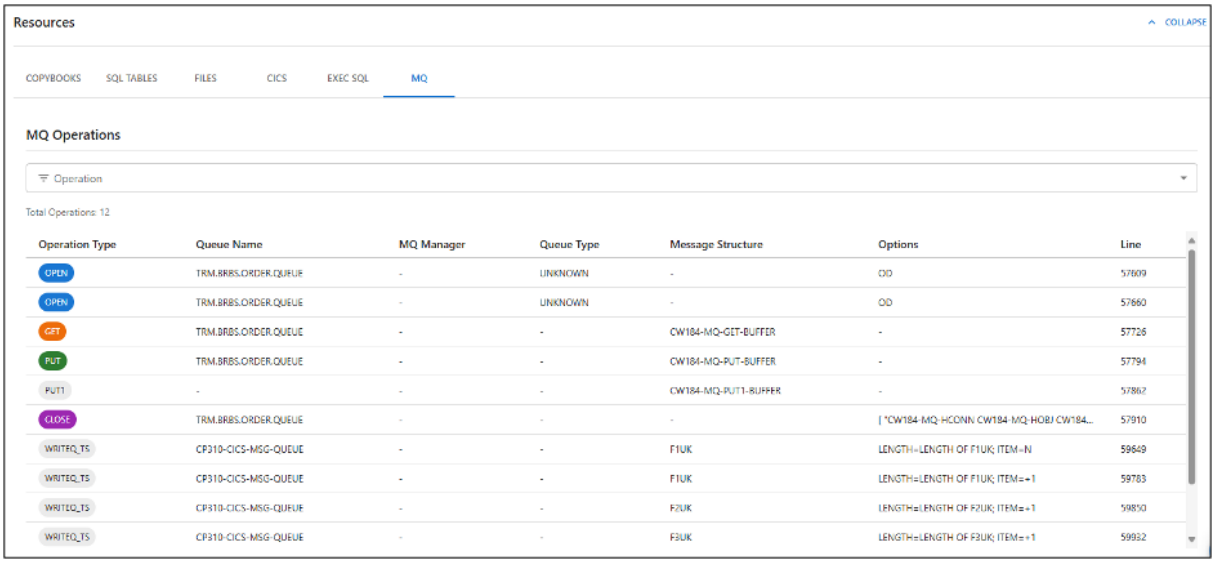

Figure 11: External Dependencies MQ

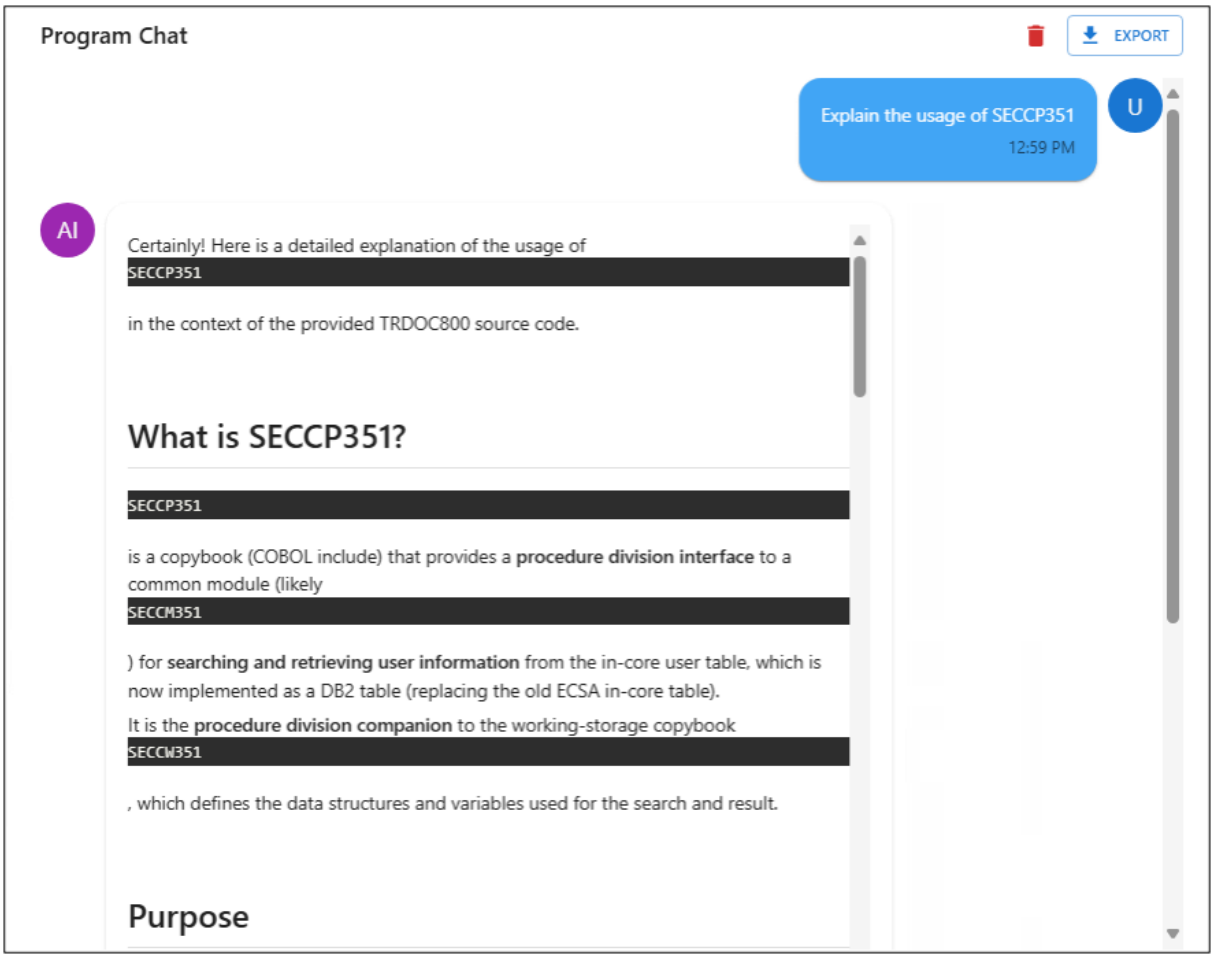

Figure 12: Interactive Chat Capabilities