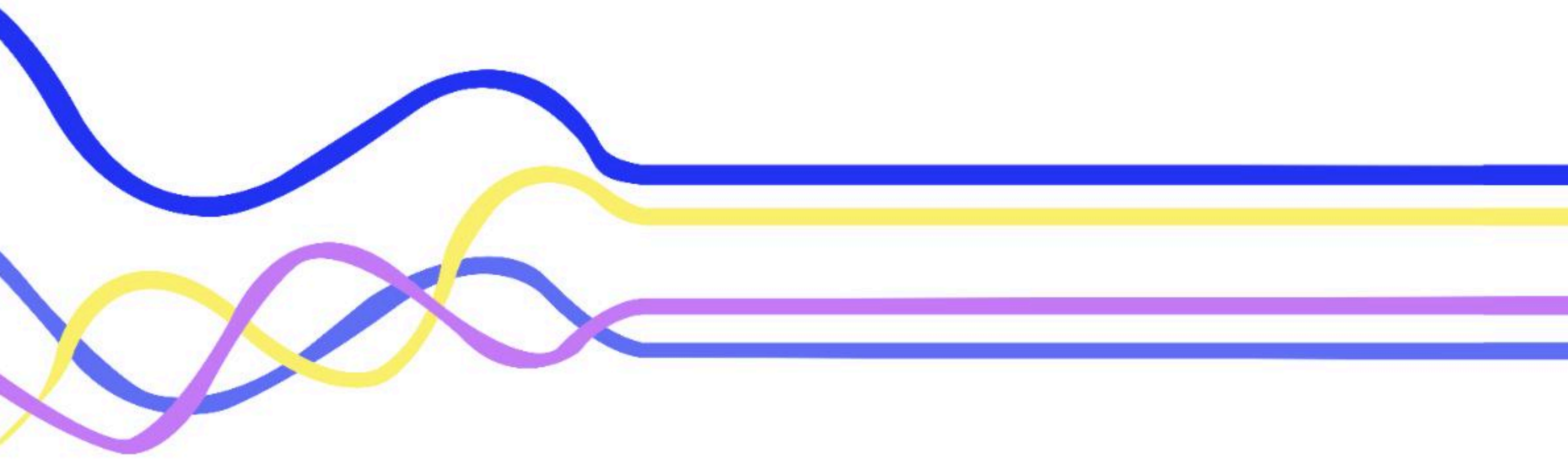
Primer

Shortcomings of traditional COBOL business rule extraction methods



Executive overview

Legacy COBOL applications remain the operational heart of global commerce, processing trillions of dollars daily across finance, insurance, and government sectors. Decades of investment are embedded within these systems, primarily as invaluable, often undocumented "business rules" that dictate core operations.

For example, a loan risk assessment may rely on dozens—or even thousands—of variables to determine whether a financial institution should issue a loan. One such rule might be as simple as "if the applicant's debt-to-income ratio exceeds 43%, flag for manual review" or as complex as a multi-tiered algorithm weighing credit history, employment stability, and regional economic forecasts.

Understanding and extracting these rules—which can number in the hundreds even in moderately sized applications (e.g., over 800 rules in a 30,000-line program)—is critical for modernization, compliance, agility, and risk mitigation. This need is increasingly urgent as the experienced COBOL workforce retires, taking critical system knowledge with them and making manual analysis unsustainable and prone to error.

While common automated approaches like static code analysis, Large Language Models (LLMs), and AI code assistants offer potential assistance, they possess fundamental limitations when faced with the complexity, scale, and environmental dependencies of real-world COBOL systems.

The deep challenge: The nature of business rules in COBOL

Extracting business rules is inherently difficult because:

→ **Rules are Implicit and Distributed:** Many rules aren't coded as simple IF condition THEN rule. They are embedded in data transformations, program control flow, file processing logic, error handling routines, and interactions between different programs and system components. Logic is often scattered across the system.

→ **Environment Dependence:** The behavior of a rule can change based on runtime parameters, JCL overrides, control cards, or specific values within input data files – factors outside the COBOL source code itself.

Comparison of common automated approaches for COBOL business rule extraction

Core functionality needed	Static analysis	LLMs / AI assistants
Parse COBOL syntax & structure	Good	Partial
Analyze complex control flow	Partial	Partial
Trace data flows across programs	Partial	Poor
Interpret external dependencies (JCL etc.)	Partial	Poor
Identify semantic meaning / business intent	No	Partial
Handle ambiguity & legacy code quirks	Partial	Partial
Provide holistic system view	Poor	Poor
Validate extracted rules accuracy	No	No

[1] LLMs/AI Assistants show partial ability but often struggle with COBOL specifics, lack deep context, and can hallucinate, requiring significant human validation.

2 \ [2] ılidation requires execution context or deep domain expertise, which these approaches lack inherently.

[3] Table Value Definitions:

- Good: The approach generally handles this functionality well for its intended scope within the context of COBOL analysis.

- Partial: The approach has some capability but faces significant limitations, inaccuracies, or requires heavy human intervention/validation for effective use in business rule extraction.
- Poor: The approach is generally unsuitable or ineffective for this functionality in this context.
- No: The approach inherently does not address this functionality.

The limits of static analysis tools

→ Static analysis tools examine source code without executing it. When it comes to extracting business rules from COBOL, they face significant hurdles:

1. **Syntax over semantics:** These tools primarily understand code structure, not the underlying business meaning or intent. They can parse an IF statement but cannot inherently know if that IF statement represents a specific business policy (e.g., "Apply a 10% discount if customer type is 'PREMIUM'"), or a technical check (e.g., “IF you got an ERROR value, set an error flag”).
2. **Ignoring the ecosystem**: Business logic in mainframe environments doesn't always reside purely within COBOL programs. They may depend heavily on external components that cause problems for static analyzers that don't interpret in context (some static analyzers may rely on external resources as well):

→ **JCL (Job Control Language):** Defines program execution flow, parameters, file handling, and conditional execution crucial for understanding batch processes.

→ **Data:** Rules are often implicit in data structures (copybooks), file formats (VSAM, sequential files), database schemas (DB2, IMS), and specific data values used in conditional logic.

- **System utilities & schedulers**: External utilities and job scheduling systems dictate when and how processes run, affecting the overall business flow.
- **Configurations & control cards:** Parameters external to the code often modify program behavior and implement specific rules.

Example: Rule Dependent on External Parameter **ENVIRONMENT DIVISION.**

```
INPUT-OUTPUT SECTION.
      FILE-CONTROL.
          SELECT OPTIONAL PARM-FILE ASSIGN TO "PARMIN"
          FILE STATUS IS WS-PARM-STATUS.
      DATA DIVISION.
      FILE SECTION.
      FD PARM-FILE.
      01 PARM-RECORD          PIC X(10).
      WORKING-STORAGE SECTION.
      01 WS-PARM-STATUS       PIC XX.
      01 WS-PROCESS-FLAG      PIC X VALUE 'N'.
      01 WS-DISCOUNT-RATE     PIC 9V99 VALUE 0.05.
      01 WS-PARM-VALUE        PIC X(10).

      PROCEDURE DIVISION.
      OPEN INPUT PARM-FILE.
      IF WS-PARM-STATUS = '00' THEN
         READ PARM-FILE INTO WS-PARM-VALUE
         IF WS-PARM-VALUE = 'REGION-EAST' THEN
            MOVE 0.10 TO WS-DISCOUNT-RATE

         END-IF
      END-IF.
      CLOSE PARM-FILE.

      IF WS-DISCOUNT-RATE > 0.05 THEN
         PERFORM APPLY-SPECIAL-DISCOUNT
      ELSE
         PERFORM APPLY-STANDARD-DISCOUNT
      END-IF.
      *> ... rest of program ...
      STOP RUN.

      APPLY-SPECIAL-DISCOUNT.
      *> Apply discount using WS-DISCOUNT-RATE (e.g., 10%)
          DISPLAY 'Applying Special Discount: ' WS-DISCOUNT-RATE.
      APPLY-STANDARD-DISCOUNT.
      *> Apply discount using WS-DISCOUNT-RATE (e.g., 5%)
          DISPLAY 'Applying Standard Discount: ' WS-DISCOUNT-RATE.
```

Summary:

A static analyzer sees the IF WS-DISCOUNT-RATE > 0.05 condition. It might trace the MOVE 0.10 but cannot know why the rate changes. The actual business rule ("Apply a 10% discount for the Eastern region, otherwise 5%") depends on the content of the external PARMIN file, which is determined at runtime (likely via JCL) and invisible to the static analysis of the COBOL code alone.

3. Complexity and obfuscation:

Real-world COBOL codebases, often decades old, present significant challenges:

- **Accumulated changes:** Years of patches, workarounds, and modifications by different developers lead to inconsistent styles and complex, hard-to-follow logic.
- **Archaic constructs:** Heavy use of GO TO, altered GO TO, complex PERFORM structures, and large monolithic programs obscure the actual flow of business logic. (See COBOL landmines for an example)
- **"Dead" code:** Code may appear unused but could be activated under specific, rare conditions defined externally.
- **Lack of business context:** Tools see variable names (often cryptic like WRK-FLG-01), arithmetic operations, and data movements. They don't understand the business concepts these represent (e.g., "Eligibility Flag for Special Promotion") or why a calculation is performed.

The limits of Large Language Models (LLMs) and AI coding assistants

LLMs have shown promise in understanding and generating human language and code. However, applying them to extract nuanced business rules from legacy COBOL has critical limitations. Similarly AI code assistants (integrated into developer IDEs for code completion, generation, and local refactoring) represent a specific application of LLMs, but they also face distinct limitations regarding deep business rule extraction from COBOL.

1. **Plausible but incorrect ("Hallucination"):** LLMs can generate explanations of COBOL code that sound reasonable but are subtly or entirely wrong. They may misinterpret complex logic, misunderstand archaic syntax, or invent connections that don't exist, especially with non-standard or heavily modified code.

Example: Ambiguous Flag Logic **WORKING-STORAGE SECTION.**

```
       01 CUSTOMER-RECORD.
          05 CUST-ID         PIC 9(10).
          05 CUST-TYPE       PIC X. *> 'S'=Standard, 'P'=Premium,
'G'=Gov
          05 CUST-STATUS     PIC 9. *> 1=Active, 5=Hold, 9=Inactive
          05 CUST-FLAG-XYZ   PIC X. *> Set by PROGXYZ based on
complex criteria

       01 TRANSACTION-RECORD.
          05 TRANS-AMOUNT    PIC S9(7)V99.

          05 TRANS-CODE      PIC XX.

       01 WS-APPROVAL-NEEDED  PIC X VALUE 'N'.

       PROCEDURE DIVISION.
       *> ... code reads customer and transaction records ...

       EVALUATE TRUE
          WHEN CUST-TYPE = 'G' AND CUST-STATUS = 1
             CONTINUE *> Government always approved
          WHEN CUST-TYPE = 'P' AND TRANS-AMOUNT > 10000.00
             MOVE 'Y' TO WS-APPROVAL-NEEDED
          WHEN CUST-FLAG-XYZ = 'A' AND TRANS-CODE = 'W1'
             MOVE 'Y' TO WS-APPROVAL-NEEDED
```

```
WHEN OTHER
  IF TRANS-AMOUNT > 5000.00 THEN
              MOVE 'Y' TO WS-APPROVAL-NEEDED
           END-IF
       END-EVALUATE.

       IF WS-APPROVAL-NEEDED = 'Y' THEN
          PERFORM ROUTE-FOR-MANUAL-APPROVAL
       ELSE
          PERFORM PROCESS-AUTOMATICALLY
       END-IF.
       *> ... rest of program ...
```

Summary:

An LLM might explain each WHEN condition correctly in isolation. However, understanding the combined business rule for requiring manual approval is complex. Crucially, the meaning of CUST-FLAG-XYZ = 'A' combined with TRANS-CODE = 'W1' is entirely dependent on external domain knowledge (what does PROGXYZ do? What does flag 'A' signify? What is transaction 'W1'?). An LLM lacking this specific context might guess ("Flag XYZ likely indicates high risk") or hallucinate a plausible but incorrect business reason, failing to capture the precise rule ("Approval needed for wire transfers over $5000, Premium customer wires over $10000, or any 'W1' transaction for customers flagged 'A' by the external risk assessment program PROGXYZ, unless it's an active Government customer").

When it comes to code, in many cases there is a deterministic answer - a single code flow that will be executed according to the instructions. Static analysis excels at that, while LLMs fail in resolving ambiguities and complex code flows.

2. Lack of deep context:

- **Domain specificity:** LLMs lack the specific, ingrained business knowledge of a particular organization or industry vertical embedded implicitly in the code and its surrounding processes.
- **Scale and interconnection:** Understanding rules often requires analyzing millions of lines of code across hundreds or thousands of programs, JCL scripts, copybooks, and database interactions – a scale and complexity that can exceed practical LLM context windows and processing, leading to fragmented understanding.

3. Weak COBOL understanding (Training data gap):

AI assistants and LLMs are trained predominantly on vast repositories of modern programming languages (like Python, Java, JavaScript) available publicly (e.g., GitHub). Publicly available COBOL code is scarce compared to proprietary codebases. This training data disparity means these tools often struggle with:

- **COBOL's unique syntax and procedural structure.**
- **Archaic or vendor-specific COBOL dialects and idioms.**
- **Generating accurate, or even compilable, COBOL code suggestions.**

4. Static viewpoint:

Like static analyzers, LLMs don't execute code. They cannot observe runtime behavior, debug scenarios, or understand how input data values dynamically alter execution paths, which is often essential for rule validation.

5. Sensitivity to code quality:

LLMs perform best on clean, well-documented code. Legacy COBOL is often the opposite, hindering the LLM's ability to accurately parse and interpret it.

6. Focus on generation, not discovery:

AI Assistants are primarily designed to write new code or complete/refactor small, existing snippets based on local context. They are not built for the deep, system-wide analysis required to discover complex, embedded business rules across a large legacy application.

7. Limited context scope:

AI assistants typically analyze only the code in the current file, immediately related open files, or user declared files. They lack the broader application-level view needed to understand rules that span multiple COBOL programs, copybooks, JCL procedures, and external dependencies. They cannot easily trace complex data flows or control logic across the entire system.

8. Surface-level explanations:

While AI assistants might "explain" a highlighted COBOL code block, this explanation is often syntactic ("this code moves data from A to B if C is true"). It typically fails to capture the business significance (the "why") or the downstream impacts of that logic within the larger business process.

Conclusion and a better way

Using static analysis alone for extracting business rules from aging and complex COBOL systems often yields unreliable results, leading to project delays and flawed execution. Similarly, relying solely on LLMs carries the possibly higher risk of plausible but wrong outputs.

Swimm offers a modern solution that overcomes these limitations by intelligently combining a deterministic static analysis engine with the power of off-the-shelf LLMs. This integrated approach at each step in the process enables the accurate extraction of business rules from even the largest and most intricate COBOL codebases.

The result is comprehensive and trustworthy documentation, freeing critical engineering resources from the time-consuming task of understanding legacy systems and allowing them to advance organizational priorities.

swimm

Swimm is an AI solution that accelerates mainframe modernization

Learn more about Swimm