# **AI-Powered Scribe: A Framework for Analyzing and Documenting Legacy COBOL and ADABAS NATURAL Systems with Large Language Models**

## **Introduction**

The enterprise computing landscape is facing a critical inflection point. Mission-critical applications, many of which have reliably powered core business functions for decades, are built on robust legacy technologies like COBOL and ADABAS NATURAL.1 These systems are known for their stability and performance in high-volume transaction processing environments.3 However, the infrastructure supporting them is under increasing strain from a non-technical but equally potent challenge: a widening knowledge gap. The generation of developers and administrators with deep, tacit knowledge of these systems is steadily retiring, creating a skills scarcity that poses a significant operational risk.3 This problem is compounded by the state of system documentation, which is often sparse, outdated, or entirely non-existent after years of unrecorded modifications by numerous programmers.3 The result is a portfolio of opaque, high-value assets that are difficult to maintain, risky to modify, and a bottleneck to modern digital transformation initiatives.

In this context, Large Language Models (LLMs) have emerged as a potentially transformative technology. While not a panacea, LLMs represent a powerful new class of analytical tools capable of processing and understanding programming languages with remarkable proficiency.7 When guided by precise, context-aware instructions, these models can act as a force multiplier for modernization teams. They can parse vast and complex codebases, extract embedded business logic, identify hidden dependencies, and generate foundational documentation at a scale and speed previously unattainable. This capability to translate legacy code into human-readable explanations and structured data is a critical first step in de-risking maintenance and paving the way for more ambitious modernization efforts like refactoring, re-platforming, or API enablement.4

This report presents a comprehensive framework for leveraging LLMs to analyze and document legacy COBOL and ADABAS NATURAL applications. It moves beyond simplistic, ad-hoc queries to establish a strategic, multi-tiered prompting methodology. The following sections provide a foundational analysis of the target legacy languages, a detailed library of prompts for generating summaries and in-depth technical documentation, a guide to creating visual process flows with PlantUML, and a robust framework for evaluating the accuracy of the generated content. The report concludes with a review of the latest academic research on AI-driven code analysis from sources like arXiv and a set of actionable recommendations for implementing this framework within an enterprise setting.

## **I. Foundational Analysis of Legacy Code Structures**

The efficacy of any LLM-driven analysis is fundamentally dependent on the quality and specificity of the prompts provided. To construct prompts that yield accurate and insightful results, one must first possess a granular understanding of the language constructs being analyzed. This chapter deconstructs the core architectural and syntactical elements of COBOL and ADABAS NATURAL, establishing the technical foundation upon which the strategic prompting framework is built.

### **1.1 Deconstructing the COBOL Program**

COBOL (Common Business-Oriented Language) is defined by its hierarchical and highly structured nature, a design that emphasizes readability through an English-like syntax.9 This rigid organization, inherited from the era of 80-column punch cards, is comprised of four mandatory divisions that must appear in a specific order. This structure is not a mere stylistic convention; it enforces a separation of concerns that can be systematically exploited for targeted LLM analysis.10

* **IDENTIFICATION DIVISION:** This is the mandatory introductory block of every COBOL program. It serves as a metadata header, containing essential identifying information such as the PROGRAM-ID, which assigns the program its canonical name, and optional paragraphs like AUTHOR and DATE-WRITTEN.9 For an LLM, this division is the first stop for identifying the program unit under analysis.
* **ENVIRONMENT DIVISION:** This division acts as the critical bridge between the abstract logic of the program and the specific physical environment in which it executes.10 Its most important component for documentation is the  
  INPUT-OUTPUT SECTION. Within this section, the FILE-CONTROL paragraph explicitly maps the logical file names used within the program (e.g., SELECT CUST-FILE) to external system files, often referenced via Job Control Language (JCL) DD statements on the mainframe.10 Analyzing this section is key to understanding a program's primary data inputs and outputs.
* **DATA DIVISION:** This division provides an exhaustive definition of all data structures the program will use.10 Its meticulous organization is central to understanding a program's function. Key sections include:
  + **FILE SECTION:** This section defines the record layouts for every file declared in the FILE-CONTROL paragraph. It details the structure of data as it exists in external files, using hierarchical level numbers (e.g., 01, 05, 10) and PICTURE (PIC) clauses to define field names, types, and lengths.9
  + **WORKING-STORAGE SECTION:** This is where all internal variables, flags, counters, and temporary data structures are declared.9 These are variables that are not part of external file records but are essential for the program's internal processing.
  + **LINKAGE SECTION:** This section is of paramount importance for mapping application architecture. It defines the data structures for parameters that are passed to the program from an external calling program, or that the program passes to a subprogram it calls.9 The  
    LINKAGE SECTION is the formal data contract for inter-program communication via the CALL statement.13
* **PROCEDURE DIVISION:** This division contains the executable code—the program's logic.10 It is structured into  
  SECTIONs and PARAGRAPHs, which are collections of sentences that perform specific tasks. Execution flows sequentially through these paragraphs unless altered by control-flow verbs such as PERFORM (to execute a paragraph and return), GO TO (to transfer control permanently), CALL (to invoke a subprogram), and I/O statements like READ and WRITE.9

The hierarchical, four-division structure of COBOL, while a product of its time, provides a distinct advantage for automated analysis. It allows for a "divide and conquer" strategy where an LLM can be prompted to analyze each division sequentially, building a holistic understanding from the ground up. Rather than asking an LLM to interpret a monolithic block of code, a more effective approach involves a chain of targeted prompts. For instance, an initial prompt can ask the model to parse the ENVIRONMENT DIVISION to list all external file dependencies. A subsequent prompt can then instruct it to analyze the DATA DIVISION to describe the record layouts for those specific files. Only then is a final prompt used to ask how the PROCEDURE DIVISION manipulates that data. This structured prompting sequence mirrors the program's own logical design, guiding the LLM to construct a more accurate and comprehensive analysis by ensuring it understands the *what* (data) before attempting to decipher the *how* (logic).

### **1.2 Navigating the ADABAS NATURAL Environment**

ADABAS NATURAL is a fourth-generation programming language (4GL) designed for the rapid development of business applications, particularly those that interact with the high-performance ADABAS database.2 Unlike COBOL, which is a compiled language, NATURAL is typically interpreted; its source code is dynamically compiled into platform-independent bytecode for execution.14 Understanding its core components is essential for effective analysis.

* **Program Structure and Libraries:** NATURAL programs and other objects (such as maps for screen layouts, and data areas) are stored in libraries.14 These function like directories, organizing the components of an application.
* **The DEFINE DATA Block:** This is the central, mandatory data definition section in every NATURAL program, appearing at the top of the source.14 It is analogous to COBOL's  
  DATA DIVISION but consolidates all data definitions into one block. This includes local variables, views for database access, and definitions of parameter data areas for inter-program communication.16
* **Database Access via DDMs and Views:** NATURAL's interaction with the ADABAS database is highly structured. It does not interact with the physical database file directly. Instead, it uses a **Data Definition Module (DDM)**, which is a logical definition of the database file, describing its fields, types, and attributes.14 Within a program's  
  DEFINE DATA block, a developer defines a VIEW to select a specific subset of fields from a DDM that the program needs to access.14 This  
  VIEW then becomes the program's window into the database.
* **Procedural Logic:** The executable logic follows the DEFINE DATA block. Key statements for database interaction include READ, FIND, and HISTOGRAM, which operate on the defined views.15 Standard control structures like  
  REPEAT...END-REPEAT for loops and IF...END-IF for conditional logic are also used.14
* **Structured vs. Reporting Mode:** A critical concept in NATURAL is its two programming modes: structured and reporting.16  
  **Structured Mode** is the modern, preferred approach. It enforces clearer program structure by requiring every logical construct to be explicitly closed with a corresponding END- statement (e.g., a READ loop must be closed with END-READ). This explicitness makes the control flow much easier for both humans and LLMs to parse. In contrast, **Reporting Mode**, common in older code, is less strict. A single END statement can terminate all open loops, making the true scope of a loop ambiguous without careful analysis.16

In the NATURAL environment, the DEFINE DATA block serves as a comprehensive manifest of a program's external dependencies. This concentration of information provides a powerful shortcut for architectural analysis. By focusing an initial LLM prompt exclusively on this block, it is possible to construct a detailed map of the program's dependencies before examining a single line of procedural code. For example, a VIEW OF EMPLOYEES statement is an explicit, undeniable dependency on the EMPLOYEES ADABAS file.14 Similarly, a

PARAMETER USING MYPDA statement indicates a dependency on a Parameter Data Area named MYPDA, which defines the data interface for a called subprogram (CALLNAT).20 A prompt can be crafted to parse only the

DEFINE DATA block and produce a list of all database files accessed via VIEWs and all subprogram interfaces defined via USING clauses. This yields a complete dependency map far more efficiently and accurately than attempting to infer these connections from scattered READ or CALLNAT statements within the procedural logic.

### **1.3 Table: Comparative Analysis of Core Constructs**

To effectively create language-specific prompts, it is useful to have a direct comparison of the core constructs in COBOL and ADABAS NATURAL. The following table serves as a "Rosetta Stone," translating key concepts between the two environments. This reference helps prevent the misapplication of concepts from one language to the other, ensuring that prompts are tailored to the specific syntax and structure of the code being analyzed.

| Concept | COBOL Implementation | ADABAS NATURAL Implementation | Key Snippet(s) |
| --- | --- | --- | --- |
| **Program Unit** | A source file containing the four divisions, identified by PROGRAM-ID. | A source object of type Program, Map, Subprogram, etc., stored in a Library. | 9 |
| **Data Definition Area** | DATA DIVISION. | DEFINE DATA block. | 10 |
| **Variable Declaration** | 01 VAR-NAME PIC X(10). in WORKING-STORAGE SECTION. | 1 #VAR-NAME (A10) in LOCAL data area. | 9 |
| **File/DB Record Layout** | FD entry with 01-level record description in FILE SECTION. | VIEW of a DDM in the DEFINE DATA block. | 9 |
| **Reusable Code Block** | PERFORM PARAGRAPH-NAME. (internal) or CALL 'SUBPROG'. (external). | PERFORM SUBROUTINE-NAME. (internal) or CALLNAT 'SUBPROG'. (external). | 9 |
| **Reusable Data Block** | COPY CUST-REC. statement to include a copybook. | INCLUDE CUST-CODE. to include a copycode; LOCAL USING MYLDA. to include a data area. | 21 |
| **Inter-Program Call** | CALL 'PROGNAME' USING... | CALLNAT 'PROGNAME'... | 13 |
| **Parameter Passing** | LINKAGE SECTION defines the data structure for passed parameters. | A Parameter Data Area (PDA) defines the interface, referenced with DEFINE DATA PARAMETER USING MYPDA. | 10 |
| **Primary Loop Construct** | PERFORM... UNTIL condition. | REPEAT... END-REPEAT or READ/FIND loops. | 9 |
| **Conditional Logic** | IF... ELSE... END-IF. or EVALUATE. | IF... ELSE... END-IF or DECIDE. | 9 |

## **II. A Strategic Framework for LLM Prompting**

Having established the foundational structures of COBOL and NATURAL, this section transitions from theory to practice. It presents a systematic, tiered framework for constructing LLM prompts designed to elicit detailed and accurate documentation. This approach moves away from simple, generic queries toward a sophisticated methodology that guides the LLM through a structured analysis process.

### **2.1 Core Principles of Prompt Engineering for Legacy Code**

Effective interaction with LLMs, especially for a nuanced task like legacy code analysis, requires a disciplined approach to prompt design. The following principles are essential for maximizing the quality and relevance of the model's output.23

* **Be Specific and Provide Context:** Vague prompts such as "explain this program" are ineffective and lead to generic, often unhelpful, responses. Prompts must be precise, referencing specific language divisions, sections, statements, and variables. Providing context, such as the language and the goal of the analysis, focuses the LLM on the relevant details.
* **Chain-of-Thought and Step-by-Step Analysis:** Complex analytical tasks should be deconstructed into a sequence of simpler, logical steps.23 This technique, often called chain-of-thought prompting, guides the LLM's reasoning process, compelling it to show its work and build a conclusion from foundational evidence. This mirrors the human process of understanding code and leads to more reliable results.
* **Persona Adoption:** A powerful technique is to instruct the LLM to adopt a specific expert persona. Beginning a prompt with Act as an expert mainframe COBOL programmer with 30 years of experience... primes the model to utilize appropriate terminology, recognize common idioms, and focus on aspects of the code that a seasoned professional would find important.
* **Specify Output Format:** To ensure the LLM's response is programmatically useful and easy for humans to read, the prompt must explicitly define the desired output format.23 Requesting output in formats like Markdown tables, numbered lists, or JSON objects transforms the LLM from a conversationalist into a structured data generator. This structured output can then be easily ingested by other tools or used as input for subsequent analysis prompts.

### **2.2 Tier 1 Prompts: High-Level Functional Summarization**

The first step in analyzing any unfamiliar program is to determine its purpose. Tier 1 prompts are designed to produce a concise, business-oriented summary, allowing an analyst to quickly triage a large codebase and prioritize which programs require deeper investigation.

* **COBOL High-Level Summary Prompt:**  
  Act as an expert COBOL systems analyst. Your task is to analyze the provided COBOL program. Based on the PROGRAM-ID, any comments in the IDENTIFICATION DIVISION, the file names referenced in the FILE-CONTROL paragraph of the ENVIRONMENT DIVISION, and the names of the main processing paragraphs in the PROCEDURE DIVISION, provide a high-level summary of the program's primary business function. The summary should be two to three paragraphs long. Conclude by listing the main inputs (files it reads from) and outputs (files it writes to or reports it generates).
* **ADABAS NATURAL High-Level Summary Prompt:**  
  Act as an expert ADABAS NATURAL developer. Your task is to analyze the provided NATURAL program. Based on the program's name, any introductory comments, the database VIEWs defined in the DEFINE DATA block, and the primary READ, FIND, or WRITE loops in the procedural code, provide a high-level summary of the program's primary business function. The summary should be two to three paragraphs long. Conclude by listing the main ADABAS files it interacts with and the primary information it produces or updates.

### **2.3 Tier 2 Prompts: Detailed Technical Documentation**

Once a program's high-level function is understood, Tier 2 prompts are used to perform a deep dive into its technical implementation. These prompts are highly specific and designed to extract critical details about data structures, dependencies, and internal logic.

#### **2.3.1 Identifying Inter-Program Communication**

A program rarely exists in isolation. Understanding how it communicates with other components is crucial for mapping application architecture and assessing the impact of any changes. This communication can be explicit (direct calls) or implicit (shared data).

* **COBOL CALL Statement Analysis:**
  + **Prompt:**  
    You are a mainframe systems analyst specializing in application dependency mapping. Analyze the PROCEDURE DIVISION of the provided COBOL program and identify all `CALL` statements. For each `CALL` statement found, provide the following information in a Markdown table with columns: "Called Program", "Call Type", "Parameters Passed (USING clause)", and "Parameter Data Structures".  
      
    1. In the "Called Program" column, list the program name being called (the literal string or the variable name).  
    2. In the "Call Type" column, specify if the call is static (literal) or dynamic (variable).  
    3. In the "Parameters Passed" column, list the names of the data items from the `USING` phrase.  
    4. In the "Parameter Data Structures" column, for each parameter, cross-reference it to its definition in the `DATA DIVISION` (WORKING-STORAGE or FILE SECTION) and describe its structure (e.g., "01-level record with fields X, Y, Z" or "PIC 9(5) COMP-3 counter").
  + **Rationale:** This prompt moves beyond simply listing CALL statements. It forces the LLM to perform a critical cross-referencing task, linking the call site in the PROCEDURE DIVISION to the detailed data definitions in the DATA DIVISION.13 This reconstructs the "data contract" between the calling and called programs, which is essential for understanding the flow of data through the application.
* **NATURAL CALLNAT Statement Analysis:**
  + **Prompt:**  
    You are an ADABAS NATURAL expert systems analyst. Analyze the provided NATURAL program and identify all `CALLNAT` statements used to call subprograms. For each `CALLNAT` found, list:  
    1. The name of the called subprogram (the literal string).  
    2. All parameters passed to the subprogram.  
    3. For each parameter, locate its definition within the `DEFINE DATA` block and describe its format (e.g., #CUST-ID (A10), #TOTAL-AMT (P9.2)).  
    4. If the source for the corresponding Parameter Data Area (PDA) is available, explain the purpose of each passed parameter based on the PDA's field descriptions.
* **Implicit Communication via Shared Files/Databases:**
  + **Prompt (for COBOL or NATURAL):**  
    Analyze the program's input/output statements (e.g., COBOL: `WRITE`, `REWRITE`; NATURAL: `UPDATE`, `STORE`, `DELETE`). Create a list of all files or database tables that are created, updated, or written to by this program. These represent potential points of asynchronous communication where this program produces data that may be consumed by other programs. For each such file/database, describe the nature of the data being modified (e.g., "Updates the customer address record," "Writes a new transaction record to the daily batch file"). Explain the potential business significance of this data output.
  + **Rationale:** In many legacy batch systems, the most critical dependencies are not through direct, synchronous CALLs but through shared datasets. A program run at 2 AM might update a file that is the primary input for another program run at 4 AM.5 A simple analysis focusing only on  
    CALL statements would completely miss this hidden, asynchronous dependency. This prompt is specifically designed to uncover these data-driven workflows by focusing on where the program *changes the state of the system* through its data outputs. Identifying these "outputs to the world" is a foundational step in creating a complete and accurate application dependency graph.

#### **2.3.2 Analyzing Reference Data (Copybooks & Copycode)**

Code and data reuse are achieved in legacy systems through the inclusion of external source members. Understanding these is key to understanding the full data structures and logic a program uses.

* **COBOL COPY Statement Analysis:**
  + **Prompt:**  
    The following COBOL program uses a `COPY` statement to include the member 'CUSTREC'. The source code for both the main program and the 'CUSTREC' copybook is provided below.  
      
    First, provide a brief explanation of the purpose and function of the `COPY` statement in COBOL.  
      
    Then, perform a detailed analysis of how the 'CUSTREC' copybook is used within the main program. Specifically, describe the full data structure of the record it defines (e.g., `CUSTOMER-RECORD`). List all elementary and group fields, their hierarchical level numbers, and their `PICTURE` (PIC) clauses as defined within the copybook.  
      
    <COBOL Program Source Code>

...DATA DIVISION.FILE SECTION.FD CUSTOMER-FILE.01 CUSTOMER-RECORD.COPY CUSTREC....<CUSTREC Copybook Source Code>  
05 CUST-ID PIC X(10).  
05 CUST-NAME PIC X(50).  
05 CUST-BALANCE PIC S9(9)V99 COMP-3.  
</CUSTREC Copybook Source Code>  
```

* + **Rationale:** This prompt structure is crucial because it provides the LLM with all the necessary context at once. It forces the model to integrate the content of the external copybook member into its analysis of the main program's DATA DIVISION, precisely mirroring how the COBOL compiler processes the COPY statement during compilation.12
* **NATURAL INCLUDE Statement Analysis:**
  + **Prompt:**  
    The following NATURAL program uses an `INCLUDE` statement to incorporate the copycode 'AUDIT-LOGIC'. The source code for both the main program and the 'AUDIT-LOGIC' copycode is provided.  
      
    First, explain the purpose of the `INCLUDE` statement and the concept of copycode in NATURAL programming.  
      
    Next, analyze the statements within the 'AUDIT-LOGIC' copycode and describe the specific functionality they add to the main program at the point of inclusion. Identify any parameterized placeholders (e.g., &1&, &2&) within the copycode. Explain how these placeholders are dynamically replaced by the values provided in the main program's `INCLUDE` statement.  
      
    <NATURAL Program Source Code>

...INCLUDE AUDIT-LOGIC 'EMPLOYEES-VIEW' '\*USER'...<AUDIT-LOGIC Copycode Source Code>  
\* This copycode logs the update action  
MOVE &2& TO #AUDIT-USER  
MOVE \*TIMD TO #AUDIT-TIMESTAMP  
WRITE WORK FILE 1 #AUDIT-USER #AUDIT-TIMESTAMP 'Updated' '&1&'  
</AUDIT-LOGIC Copycode Source Code>  
```

* + **Rationale:** This prompt specifically targets the parameterization feature (&n&) of NATURAL copycode, a key differentiator from static COBOL copybooks.22 It requires the LLM to understand not just the inclusion of code, but the dynamic substitution that occurs at compile time, leading to a more accurate interpretation of the final executed logic.

### **2.4 Table: Master Prompt Library for Code Analysis**

To make this framework easily implementable, the following table centralizes the developed prompts into a ready-to-use resource. This master library serves as a starting point for any technical team embarking on an LLM-assisted documentation project. The rationale provided for each prompt captures the strategic thinking behind its construction, enabling teams to not only apply the prompts but also to adapt them intelligently for unique or complex scenarios.

| Prompt ID | Language | Task Category | Prompt Title | Full Prompt Text | Expected Output Format | Rationale/Notes |
| --- | --- | --- | --- | --- | --- | --- |
| **SUM-CBL-01** | COBOL | Summarization | High-Level Business Function | Act as an expert COBOL systems analyst... provide a high-level summary... | 2-3 paragraphs of text, followed by bulleted lists of inputs and outputs. | Quickly ascertains the program's purpose for triage and prioritization. Focuses on business terms. |
| **SUM-NAT-01** | NATURAL | Summarization | High-Level Business Function | Act as an expert ADABAS NATURAL developer... provide a high-level summary... | 2-3 paragraphs of text, followed by bulleted lists of files and outputs. | Tailored to NATURAL constructs (VIEWs, READ/FIND loops) to achieve the same goal as SUM-CBL-01. |
| **DEP-CBL-01** | COBOL | Dependency Analysis | Explicit Program Calls | You are a mainframe systems analyst... identify all 'CALL' statements... | Markdown table with columns: "Called Program", "Call Type", "Parameters Passed", "Parameter Data Structures". | Forces deep analysis by requiring cross-referencing between PROCEDURE and DATA divisions to map the data contract. |
| **DEP-NAT-01** | NATURAL | Dependency Analysis | Explicit Subprogram Calls | You are an ADABAS NATURAL expert... identify all 'CALLNAT' statements... | Numbered list detailing called program, parameters, and their data formats for each CALLNAT. | Focuses on the CALLNAT verb and the associated Parameter Data Areas (PDAs) that define the interface. |
| **DEP-ALL-01** | Both | Dependency Analysis | Implicit Data Dependencies | Analyze the program's input/output statements... list of all files or database tables that are created, updated, or written to... | Bulleted list of files/DBs, with a description of the data modified and its business significance. | Crucial for uncovering hidden, asynchronous dependencies common in batch systems, which are often undocumented. |
| **REF-CBL-01** | COBOL | Reference Data | Copybook Integration Analysis | The following COBOL program uses a 'COPY' statement... explain... analyze... | Textual explanation followed by a detailed breakdown of the copybook's data structure (fields, levels, PIC clauses). | Provides all necessary context (main program + copybook) to the LLM at once, mimicking the compiler's view. |
| **REF-NAT-01** | NATURAL | Reference Data | Copycode and Parameter Analysis | The following NATURAL program uses an 'INCLUDE' statement... explain... analyze... | Textual explanation followed by an analysis of the included logic and the role of its parameters (&n&). | Specifically tests the LLM's ability to understand the dynamic parameter substitution feature of NATURAL INCLUDEs. |

## **III. Generating Visual Process Flows with PlantUML**

While textual documentation is foundational, visual diagrams are unparalleled for communicating complex process flows and control logic to a broad audience. PlantUML is an open-source tool that generates diagrams from a plain text language, making it ideal for integration into a documentation-as-code workflow.27 Activity diagrams, in particular, are well-suited for representing the procedural logic of legacy programs.27

### **3.1 Principles of Effective Code Visualization**

The goal of generating a diagram from code is to enhance clarity, not to create a one-to-one visual replica of every line of code, which would be overwhelmingly complex. The focus should be on illustrating the high-level control flow, decision points, and major processing loops, abstracting away the low-level details of data manipulation.

* **Clarity over Clutter:** A good diagram simplifies. It should represent major logical blocks, such as COBOL PARAGRAPHs or NATURAL subroutines, as single activities.
* **Use Partitions and Notes:** PlantUML's partition keyword is an excellent way to visually group activities that correspond to a specific code block (e.g., a single COBOL PARAGRAPH), making the diagram's structure reflect the code's structure.30 The  
  note keyword can be used to attach explanations of complex business rules or decision criteria directly onto the diagram.30
* **Focus and Scope:** Diagrams should be scoped to a single program or a single critical path within a very large program. Attempting to diagram an entire complex system in one activity diagram will result in an unreadable and useless artifact.

### **3.2 Crafting Prompts for PlantUML Generation**

The challenge in generating a PlantUML diagram is not just getting the LLM to understand the code's logic, but to translate that procedural logic into the specific declarative syntax of PlantUML. The prompt must act as a bridge, providing an explicit mapping between the constructs of the source language (COBOL/NATURAL) and the keywords of PlantUML. This transforms the LLM from a code *commentator* into a code *translator*, converting from one formal language to another to produce a directly usable output.

* **COBOL to PlantUML Activity Diagram Prompt:**  
  Act as a systems analyst and PlantUML expert. Analyze the PROCEDURE DIVISION of the provided COBOL program. Your task is to generate the syntax for a PlantUML activity diagram that represents the program's high-level control flow. Adhere to the following rules for the translation:  
    
  1. The diagram must start with `@startuml` and end with `@enduml`.  
  2. The start of the diagram's logic should correspond to the first executable statement in the PROCEDURE DIVISION.  
  3. Represent each major COBOL PARAGRAPH as a distinct activity, using the paragraph name as the activity label (e.g., `:100-PROCESS-RECORDS;`).  
  4. Represent `IF...ELSE` statements using PlantUML's `if (...) then (yes)... else (no)... endif` syntax.  
  5. Represent `PERFORM... UNTIL` loops using PlantUML's `repeat... :action in loop;... while (...) is (no)` syntax. The loop continues while the UNTIL condition is false.  
  6. Represent `CALL` statements as distinct activities, clearly labeling the activity with the name of the called program (e.g., `:CALL 'SUBPROG1';`).  
  7. Use PlantUML `partition` blocks to visually group all activities that belong to the same COBOL PARAGRAPH. The partition should be labeled with the paragraph name.  
  8. Do not generate an image or an explanation of the diagram. Output \*only\* the raw PlantUML syntax enclosed in a single Markdown code block.
* **ADABAS NATURAL to PlantUML Activity Diagram Prompt:**  
  Act as a systems analyst and PlantUML expert. Analyze the provided ADABAS NATURAL program. Your task is to generate the syntax for a PlantUML activity diagram that represents the program's high-level control flow. Adhere to the following rules for the translation:  
    
  1. The diagram must start with `@startuml` and end with `@enduml`.  
  2. Represent major processing blocks as activities (e.g., `:Initialize variables;`).  
  3. Represent `READ`, `FIND`, or `HISTOGRAM` loops using PlantUML's `repeat... :Process record;... while (more records?) is (yes)` syntax.  
  4. Represent `IF...END-IF` statements using PlantUML's `if (...) then (yes)... else (no)... endif` syntax.  
  5. Represent `CALLNAT` statements as distinct activities, clearly labeling the activity with the name of the called subprogram (e.g., `:CALLNAT 'SUBPROG1';`).  
  6. Use PlantUML `note`s to explain the criteria in complex `WHERE` clauses or `IF` conditions.  
  7. Do not generate an image or an explanation of the diagram. Output \*only\* the raw PlantUML syntax enclosed in a single Markdown code block.

## **IV. Evaluating and Refining LLM-Generated Documentation**

The integration of LLMs into the documentation process offers unprecedented speed and scale, but it is not infallible. The output of an LLM, no matter how sophisticated the model or the prompt, must be considered a high-quality first draft, not an unquestionable final product. A rigorous evaluation and validation process, with a human expert at its center, is not an optional step but a mandatory component of a responsible AI-driven workflow.

### **4.1 The Human-in-the-Loop Imperative**

Recent academic research has provided empirical evidence for the vulnerabilities of LLMs when analyzing legacy code. A key study demonstrated that the quality of existing documentation, specifically code comments, has a direct and significant impact on an LLM's comprehension.33 While accurate comments improve comprehension, the study found that providing code with completely inaccurate comments caused a significant degradation in the LLM's analytical performance. The models exhibited a tendency to "believe that all provided input is true and correct," attempting to reconcile the faulty comments with the actual code logic, often leading to erroneous conclusions.33

This finding has profound implications for legacy systems. These codebases are frequently littered with comments that are decades old, written for previous versions of the logic, or are simply wrong. An LLM, lacking the human ability to skeptically dismiss an obviously incorrect comment, can be actively misled. Therefore, any documentation generated by an LLM cannot be trusted blindly. It must be subjected to a thorough review by a human domain expert who can validate its accuracy against the true behavior of the code and the business context.

### **4.2 Prompts for Quality Assurance and Validation**

To facilitate this essential human review, a set of evaluation prompts can be used. These prompts instruct the LLM to critique its own work or to perform comparative analysis, helping to surface potential errors, ambiguities, and inconsistencies for the human reviewer to investigate.

* **Self-Correction and Ambiguity Identification Prompt:**  
  You have previously generated the following analysis of a program:  
    
  [Insert the LLM's previous output here]  
    
  Now, adopt the persona of a skeptical senior code reviewer with 30 years of experience. Your task is to critically re-examine both the original source code and your own analysis. Identify and list any potential weaknesses, ambiguities, or areas of uncertainty in your initial analysis. Specifically consider:  
  1. Are there any cryptic variable names (e.g., `WK-FLAG-01`, `TEMP-SWITCH`) whose purpose might be misinterpreted or could have multiple meanings depending on the context?  
  2. Are there any complex conditional statements (`IF` or `EVALUATE`) where the business logic might be more nuanced than your summary suggests?  
  3. Did you make any assumptions about the program's logic that are not explicitly supported by the code?  
  List these points of uncertainty for a human expert to verify.
* **Comment-vs-Logic Inconsistency Check Prompt:**  
  The provided legacy source code contains embedded comments. It is highly possible that these comments are outdated, incomplete, or inaccurate. Your task is to perform a two-stage analysis to identify discrepancies.  
    
  1. \*\*Stage 1 (Logic-Only Analysis):\*\* Analyze the procedural logic of the code \*while completely ignoring all comments\*. Generate a brief, bullet-point summary of what the code appears to be doing based solely on its executable statements.  
  2. \*\*Stage 2 (Comparison):\*\* Now, carefully read the comments within the code. Compare the summary you generated in Stage 1 with what the comments claim the code does.  
  3. \*\*Output:\*\* Produce a report that explicitly lists any discrepancies or contradictions found between the code's actual logic and its comments. For each discrepancy, state what the comment says and what the code actually does.  
  + **Rationale:** This prompt directly operationalizes the findings from the academic research on LLM vulnerabilities.33 It forces the LLM to perform two distinct analytical passes—one based on logic and one based on comments—and then to act as a differential engine. This structured process is designed to systematically uncover the very inconsistencies that the research proves can otherwise mislead the model into making false conclusions.

### **4.3 Table: Validation Checklist for Human Reviewers**

The following table provides a structured checklist to guide the human expert's review of LLM-generated documentation. It transforms the vague task of "review this output" into a formal audit, ensuring that all critical aspects of the documentation are systematically verified for correctness and clarity.

| Validation Area | Question/Prompt for the Human Reviewer | Method | Relevant Snippet(s) |
| --- | --- | --- | --- |
| **Functional Correctness** | Does the high-level summary accurately describe the program's business purpose? Does it align with your domain knowledge of the application? | Manual Review | 6 |
| **Data Flow Accuracy** | Are all input files, output files, and database interactions correctly identified? Is the description of data being read/written accurate? | Manual Review + Cross-reference with JCL/System Docs | 10 |
| **Control Flow Logic** | Does the PlantUML diagram (or textual description of logic) correctly represent the main loops and decision branches in the code? | Manual Review + Compare Diagram to Code | 27 |
| **Dependency Identification** | Have all CALL/CALLNAT statements been correctly identified? Is the analysis of passed parameters and shared data areas correct? Has implicit communication via shared files been considered? | Manual Review + Cross-reference with other program sources | 3 |
| **Clarity and Readability** | Is the documentation written in clear, unambiguous language? Are technical terms used correctly? Is the formatting clean and easy to follow? | Manual Review | 9 |
| **Comment Inconsistency** | Run the "Comment-vs-Logic Inconsistency Check" prompt. Does the LLM's report of discrepancies align with your own reading of the code? | Automated Prompt + Manual Verification | 33 |

## **V. The Academic Frontier: Novel Approaches in AI-Driven Code Analysis**

While the framework presented in this report focuses on current, practical applications of LLMs, the field of AI-driven software engineering is evolving at a rapid pace. A review of recent academic literature, particularly from preprint servers like arXiv, provides valuable insights into emerging capabilities and future directions. This research informs our understanding of both the potential and the limitations of these models.

### **5.1 Survey of Recent arXiv Papers**

Analysis of recent papers reveals several key themes relevant to the documentation and modernization of legacy systems.

* **Theme 1: LLM Comprehension and the Impact of Documentation Quality:** A recurring and critical theme is the investigation into how well LLMs truly comprehend code, especially legacy code. A foundational study, "Impact of Comments on LLM Comprehension of Legacy Code," established a direct link between the prevalence and accuracy of code comments and the model's analytical performance.33 The paper introduced Multiple-Choice Question Answering (MCQA) as an effective, objective methodology for quantitatively measuring LLM comprehension, moving beyond subjective assessments of generated text.35 The core finding—that LLMs are susceptible to being misled by inaccurate documentation—underpins the necessity of the human-in-the-loop validation process described in the previous chapter.
* **Theme 2: Expanding Applications in Modernization:** Research is rapidly moving beyond simple analysis and documentation. Multiple papers explore the use of LLMs for more advanced software engineering tasks. These include the automated translation of code from one language to another (e.g., COBOL to Java), a process often referred to as transpilation or migration.7 Other research focuses on automated code refactoring to improve the structure of legacy code and even decompilation, which aims to recover high-level source code from compiled binaries.7 The structured, validated documentation generated by the framework in this report would serve as a crucial, high-quality input for these more advanced modernization tools and processes.
* **Theme 3: Models, Datasets, and the Scarcity Challenge:** The performance of any LLM is tied to its training data. Papers in this area survey the landscape of models used for code analysis, such as OpenAI's GPT series and specialized models like CodeBERT, and the datasets they are trained on.7 A significant challenge highlighted by this research is the relative scarcity of legacy language source code (like COBOL, Fortran, and NATURAL) in the vast public datasets used for training, compared to the immense volume of modern languages like Python, Java, and C++.35 This data imbalance likely contributes to why LLMs may struggle more with legacy idioms and why highly specific, context-rich prompting is essential to guide them effectively.

### **5.2 Table: Summary of Key Academic Research**

The following table distills the primary findings from this body of research and translates them into practical implications for an enterprise legacy documentation project. It provides a quick-reference guide for technical leaders to understand the "so what" of the academic work without needing to read the full papers.

| Paper/Theme | Key Finding | Practical Implication for Legacy Documentation Project | Relevant Snippet(s) |
| --- | --- | --- | --- |
| **Impact of Comments on Comprehension** | LLM comprehension of legacy code is significantly degraded by inaccurate comments and improved by accurate, prevalent comments. LLMs tend to treat all input, including faulty comments, as truth. | LLM-generated documentation can never be fully trusted. A mandatory human validation step is required to catch errors introduced by the model being misled by poor-quality existing comments. | 33 |
| **Objective Evaluation with MCQA** | Multiple-Choice Question Answering (MCQA) is an emerging, effective, and non-subjective method for quantitatively evaluating an LLM's understanding of code. | The principles of MCQA can be adapted to create structured validation checklists for human reviewers, turning a vague "review" task into a concrete audit of the LLM's output against specific criteria. | 34 |
| **LLMs for Code Translation/Migration** | LLMs are showing increasing capability in translating code between languages (e.g., legacy to modern) and refactoring codebases. | The documentation and dependency maps generated by the framework in this report are not just an end-product but a critical prerequisite for successful, AI-assisted modernization projects. | 7 |
| **The Legacy Code Data Scarcity Problem** | Legacy languages like COBOL are underrepresented in the massive training datasets for LLMs compared to modern languages. | LLMs have less "innate" knowledge of legacy code. Therefore, success depends heavily on high-quality prompting that provides explicit context, structure, and guidance, as outlined in this report's framework. | 35 |

## **VI. Recommendations and Implementation Roadmap**

The analysis and frameworks presented in this report provide a clear path for leveraging Large Language Models to address the critical challenge of legacy system documentation. To translate this strategy into action, the following recommendations and implementation roadmap are proposed.

### **6.1 Key Recommendations**

1. **Adopt a Tiered, Structured Prompting Framework:** Resist the temptation of ad-hoc, simplistic prompting. A systematic approach is essential for consistent and high-quality results. The organization should formally adopt the tiered framework, utilizing Tier 1 prompts for initial high-level assessment and Tier 2 prompts for in-depth technical analysis. This disciplined methodology ensures that documentation efforts are both efficient and thorough.
2. **Prioritize Application Dependency Analysis:** A single program's documentation is useful, but a map of how programs interact is invaluable. Special emphasis should be placed on executing prompts designed to identify all forms of inter-program communication. This includes both explicit dependencies, such as COBOL CALLs and NATURAL CALLNATs, and, critically, the implicit dependencies created by programs sharing data through files and databases.5 Building this comprehensive dependency graph is a prerequisite for any meaningful impact analysis or modernization planning.
3. **Implement a Mandatory Human-in-the-Loop Validation Process:** Based on clear academic evidence of LLM fallibility, particularly when faced with inaccurate source comments, all LLM-generated output must be treated as a draft requiring expert validation.33 The organization must establish a formal review workflow where senior developers or system analysts use a structured checklist, such as the one provided in Section 4.3, to verify the accuracy and completeness of the generated documentation. This step is non-negotiable for creating a trusted knowledge base.
4. **Build a Centralized, Version-Controlled Knowledge Base:** The validated documentation should not reside in disparate documents or on local machines. It should be stored in a centralized, version-controlled repository like Git, integrated with a collaboration platform like Confluence.28 This creates a living, evolving "single source of truth" for the application's architecture and logic. This knowledge base becomes a priceless asset, serving as a reference for new developers, a guide for maintenance, and a foundational dataset for future AI-driven modernization initiatives.

### **6.2 Proposed Implementation Phases**

A phased approach is recommended to implement this framework, allowing the organization to build confidence, refine processes, and demonstrate value incrementally.

* **Phase 1: Pilot Project (Duration: 1-2 Sprints)**
  + **Objective:** To validate the methodology and its outputs in a controlled environment.
  + **Activities:**
    1. Select a small, well-understood but non-trivial application consisting of 3-5 programs.
    2. Systematically apply the prompts from the Master Prompt Library (Section 2.4) to each program.
    3. Assign a senior developer or analyst to act as the validator, using the Validation Checklist (Section 4.3) to review all LLM-generated output.
    4. Generate PlantUML activity diagrams and solicit feedback from the development team on their clarity and utility.
    5. Conduct a retrospective to assess the quality of the documentation, the effort required for validation, and the overall value of the process.
  + **Goal:** To prove the framework's effectiveness and build internal support for a broader rollout.
* **Phase 2: Scaled Documentation (Duration: 3-6 Months)**
  + **Objective:** To apply the validated methodology at scale to a critical business system or application suite.
  + **Activities:**
    1. Target a high-priority system identified by business and IT leadership.
    2. Develop scripts or lightweight automation to streamline the process of feeding code to the LLM API and ingesting the structured results.
    3. Establish a formal review workflow within the development team, assigning specific modules to experts for validation.
    4. Begin populating the centralized, version-controlled knowledge base with the validated documentation artifacts.
  + **Goal:** To create a comprehensive, trusted documentation repository for a significant legacy asset, delivering tangible value by reducing analysis time for maintenance and change requests.
* **Phase 3: Integration with Modernization Initiatives (Ongoing)**
  + **Objective:** To leverage the newly created knowledge base as a strategic accelerator for modernization.
  + **Activities:**
    1. Use the validated documentation and dependency maps as the primary input for strategic initiatives such as business rule extraction, identifying candidates for API enablement, or planning for automated code conversion or migration.4
    2. Continuously refine the prompt library based on feedback from the validation process and the evolving capabilities of LLMs.
    3. Investigate the use of the curated documentation to fine-tune a specialized LLM, potentially improving its accuracy on the organization's specific codebase.
  + **Goal:** To transform the documentation effort from a passive, archival activity into an active, foundational component of the enterprise's long-term technology strategy.

#### Works cited

1. Hello World - Your First Program and its Structure. Simple Display Statement - CodeSignal, accessed June 22, 2025, <https://codesignal.com/learn/courses/introduction-to-cobol-programming/lessons/hello-world-your-first-program-and-its-structure-simple-display-statement>
2. Learn Adabas with online courses and programs - edX, accessed June 22, 2025, <https://www.edx.org/learn/adabas>
3. Natural and Adabas Modernization, accessed June 22, 2025, <https://modernsystems.oneadvanced.com/globalassets/modern-systems-assets/resources/ebook/natural-and-adabas-modernization-ebook.pdf>
4. The Clock is Ticking for Customers Running Natural & Adabas on z/VSE - Astadia, accessed June 22, 2025, <https://www.astadia.com/blog/the-clock-is-ticking-for-customers-running-natural-adabas-on-z-vse>
5. Top 5 Challenges of Mainframe Modernization Projects - Astadia, accessed June 22, 2025, <https://www.astadia.com/blog/top-5-challenges-of-mainframe-modernization-projects>
6. COBOL Modernization: Benefits, Challenges & 5 Critical Techniques - Swimm, accessed June 22, 2025, <https://swimm.io/learn/cobol/cobol-modernization-benefits-challenges-and-5-critical-techniques>
7. Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets - arXiv, accessed June 22, 2025, <https://arxiv.org/html/2503.17502v1>
8. CodeMEnv: Benchmarking Large Language Models on Code Migration - arXiv, accessed June 22, 2025, <https://arxiv.org/html/2506.00894v1>
9. Understanding COBOL: Divisions, Syntax, Challenges, and Modernizing Your Code, accessed June 22, 2025, <https://swimm.io/learn/cobol/understanding-cobol-divisions-syntax-challenges-and-modernizing-your-code>
10. What Is COBOL? - IBM, accessed June 22, 2025, <https://www.ibm.com/think/topics/cobol>
11. COBOL - Basic Syntax - GeeksforGeeks, accessed June 22, 2025, <https://www.geeksforgeeks.org/cobol/cobol-basic-syntax/>
12. About copybook templates - IBM, accessed June 22, 2025, <https://www.ibm.com/docs/en/file-manager-for-zos/14.1.0?topic=structure-about-copybook-templates>
13. Calling Programs, accessed June 22, 2025, <https://www.microfocus.com/documentation/server-express/sx20books/prcall.htm>
14. Getting Started with Adabas and Natural - AWS, accessed June 22, 2025, <https://cdck-file-uploads-global.s3.dualstack.us-west-2.amazonaws.com/techcommunity/original/2X/d/d0f0b927ee6024c8e6b5f5d9aa752cb651b639b5.pdf>
15. ADABAS - Wikipedia, accessed June 22, 2025, <https://en.wikipedia.org/wiki/ADABAS>
16. Natural Programming Modes - Software AG Documentation, accessed June 22, 2025, <https://documentation.softwareag.com/natural/nat841unx/pg/pg_mode.htm>
17. VS Code Extention for Adabas Natural - Stack Overflow, accessed June 22, 2025, <https://stackoverflow.com/questions/68699011/vs-code-extention-for-adabas-natural>
18. Adabas Usage - IBM, accessed June 22, 2025, <https://www.ibm.com/docs/en/addi/6.1.2?topic=programs-adabas-usage>
19. Natural Programming Basic - Software AG Learning Portal, accessed June 22, 2025, <https://learn.softwareag.com/course/info.php?id=1467>
20. Defining a COPYCODE?? - Adabas-Natural - Software AG Tech Community, accessed June 22, 2025, <https://tech.forums.softwareag.com/t/defining-a-copycode/60844>
21. COBOL - Copy Statement - GeeksforGeeks, accessed June 22, 2025, <https://www.geeksforgeeks.org/cobol/cobol-copy-statement/>
22. Copycode - Software AG Documentation, accessed June 22, 2025, <https://documentation.softwareag.com/naturalONE/natONE912/natmf/pg/pg_obj_copycode.htm>
23. LLM Prompting Techniques for Developers - Pedro Alonso, accessed June 22, 2025, <https://www.pedroalonso.net/blog/llm-prompting-techniques-developers/>
24. What is a Copybook in COBOL? - Northern Virginia SEO and Digital Marketing, accessed June 22, 2025, <https://thospfuller.com/2023/11/08/what-is-a-copybook-in-cobol/>
25. NATURAL Programming | PDF | Subroutine | Control Flow - Scribd, accessed June 22, 2025, <https://www.scribd.com/document/194083913/NATURAL-Programming>
26. INCLUDE - Software AG Documentation, accessed June 22, 2025, <https://documentation.softwareag.com/natural/nat911unx/sm/include.htm>
27. UML Made Easy with PlantUML & VS Code - CodeProject, accessed June 22, 2025, <https://www.codeproject.com/Articles/1278703/UML-Made-Easy-with-PlantUML-VS-Code>
28. Optimizing documentation with PlantUML: Integration, Efficiency, and Best Practices, accessed June 22, 2025, <https://community.atlassian.com/forums/App-Central-articles/Optimizing-documentation-with-PlantUML-Integration-Efficiency/ba-p/2848741>
29. Quick Guide to PlantUML: Diagrams, Syntax & Best Practices | Miro, accessed June 22, 2025, <https://miro.com/diagramming/what-is-plantuml/>
30. Activity Diagram (legacy) syntax - PlantUML, accessed June 22, 2025, <https://plantuml.com/activity-diagram-legacy>
31. Activity Diagram (New Syntax) - PlantUML, accessed June 22, 2025, <https://plantuml.com/activity-diagram-beta>
32. Sequence Diagram - PlantUML Web Server, accessed June 22, 2025, <https://plantuml.com/sequence-diagram>
33. Impact of Comments on LLM Comprehension of Legacy Code - arXiv, accessed June 22, 2025, <https://www.arxiv.org/pdf/2506.11007>
34. [2506.11007] Impact of Comments on LLM Comprehension of Legacy Code - arXiv, accessed June 22, 2025, <https://arxiv.org/abs/2506.11007>
35. Impact of Comments on LLM Comprehension of Legacy Code - arXiv, accessed June 22, 2025, <https://arxiv.org/pdf/2506.11007>
36. LLM4Decompile: Decompiling Binary Code with Large Language Models | Request PDF, accessed June 22, 2025, <https://www.researchgate.net/publication/386193431_LLM4Decompile_Decompiling_Binary_Code_with_Large_Language_Models>
37. [2503.17502] Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets - arXiv, accessed June 22, 2025, <https://arxiv.org/abs/2503.17502>
38. Large Language Models (LLMs) for Source Code Analysis: applications, models and datasets - arXiv, accessed June 22, 2025, [https://arxiv.org/pdf/2503.17502?](https://arxiv.org/pdf/2503.17502)
39. Bridging Gaps in LLM Code Translation: Reducing Errors with Call Graphs and Bridged Debuggers - ResearchGate, accessed June 22, 2025, <https://www.researchgate.net/publication/385287137_Bridging_Gaps_in_LLM_Code_Translation_Reducing_Errors_with_Call_Graphs_and_Bridged_Debuggers>