Leveraging LLM for software modernization: COBOL Functionality Extraction Case study

Asha Rajbhoj
TCS Research
Tata Consultancy Services
Pune, INDIA
asha.rajbhoj@tcs.com

Akanksha Somase
TCS Research
Tata Consultancy Services
Pune, INDIA
akanksha.somase@tcs.com

Tanay Sant
TCS Research
Tata Consultancy Services
Pune, INDIA
tanay.sant@tcs.com

Ajim Pathan
TCS Research
Tata Consultancy Services
Pune, INDIA
ajim.pathan@tcs.com

Purvesh Doud
TCS Research
Tata Consultancy Services
Pune, INDIA
purvesh.doud@tcs.com

Vinay Kulkarni
TCS Research
Tata Consultancy Services
Pune, INDIA
vinay.vkulkarni@tcs.com

***Abstract*— Businesses are replete with applications created decades ago and are still relevant functionality wise. However, they pose significant evolution and integration challenges – the former because of paucity of skilled workforce and the latter due to high impedance mismatch with modern technology stack. The two constitute principal reasons leading to contemplation of modernization of these applications. Typical approach is to migrate existing code to the desired technology stack as a language transformation endeavor under functional equivalence. However, traditional parser-based approaches and lift-and-shift modernization methods typically add to the technology debt, thus making evolution of the modernized code even more challenging. To overcome the various lacunae in current practice in software modernization, we propose the stagewise refinement approach using LLM. In this paper, we focus on the stage of human-in-the-loop automation aided generation of functionality description specifically for Common Business-Oriented Language (COBOL) code. We illustrate the utility and efficacy of the proposed approach through validation on a small but complex business application.**

Keywords*— *Software Modernization, COBOL Functionality Extraction, Reverse Engineering, Code Summarization, Large Language Models, Model-driven Engineering.

I. INTRODUCTION

From time to time, businesses must resort to software modernization to stay competitive in the face of rapidly advancing technology. Legacy software often undergoes extensive modifications due to evolutionary maintenance to meet business needs and to keep up with hardware updates for several years by multiple developers. The lack of documentation makes it difficult for new developers to effectively understand and maintain code, resulting in code that can obscure its original code structure and intent. While a legacy application does deliver the required functionality, with ever-shrinking pool of workforce proficient in legacy technologies, evolutionary maintenance is an ever-growing challenge.

Large impedance mismatch between legacy and modern technology stacks makes software modernization process time-consuming, costly, and complex. Typical approach is to migrate existing code to the desired technology stack as a language transformation endeavor under functional equivalence. Often, the focus has been on lift-and-shift approaches, failing to leverage the benefit of modern technologies. These methods are rule-based and perform statement-wise translation, overlooking possible modularization and refactoring/restructuring to leverage the advantage of modern programming languages. Thus, traditional parser-based approach and lift-and-shift modernization methods typically add to the technology debt, making evolutionary maintenance even more challenging.

Recent advancements in artificial intelligence (AI), particularly LLMs, offer a promising alternative to automate and accelerate software modernization for a variety of reasons: (i) LLMs have inbuilt understanding of mature domains due to their training on vast data available in public domain, (ii) Mechanisms like Retrieval Augmented Generation (RAG) and "in-context learning" enable augmentation with local and/or private data, (iii) LLMs are designed for text generation. As a result, LLMs can provide a jump-start for generating functionality description in natural language text from code. However, there are several challenges in using LLMs for modernizing industry-strength software. Industry-strength programs are typically large in size e.g., several millions of lines of COBOL code. Moreover, legacy code typically tends to stress on performance way more than on understandability or maintainability thus leading to dense and tightly coupled code. This problem gets further exacerbated as programming language tends to significantly influence program structure. Also, legacy code tends to be poorly commented with comments quickly going out of sync with code. Hence, producing functionality description in natural language text from code of industry-

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00012

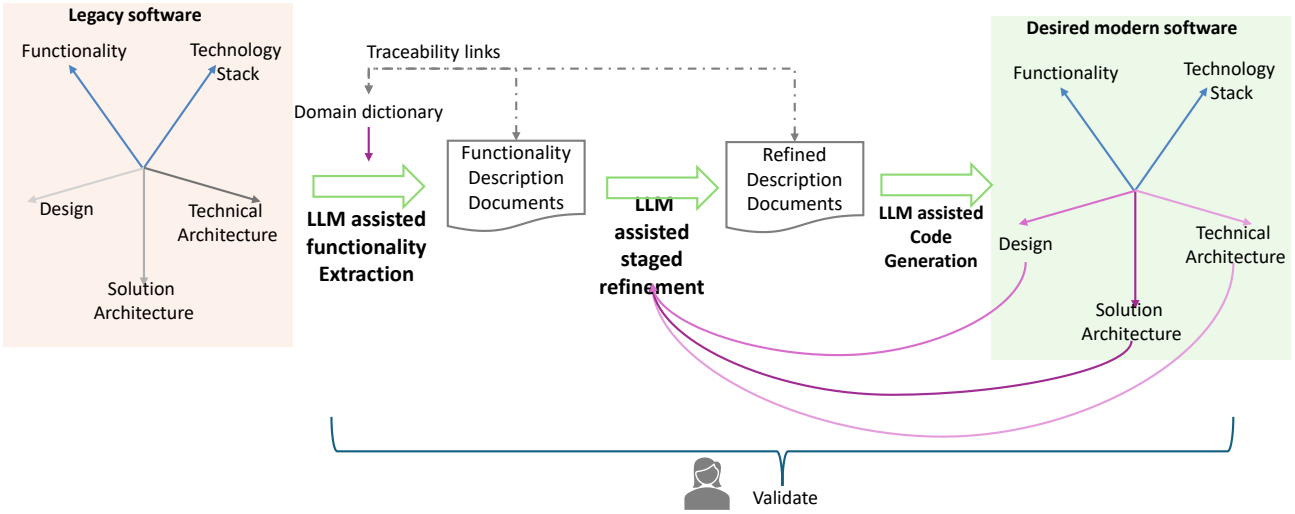

Fig. 1: LLM-assisted software modernization

strength software is a hard problem. To address these challenges we propose a systematic, human-in-the-loop approach of software modernization that combines model-driven engineering (MDE) approach with LLM to take advantages of both. This paper presents the software modernization methodology and illustrates the process and outcomes of the initial step of functionality extraction for COBOL applications and presents validation on a small yet complex business application.

The rest of the paper is organized as follows: Section II discusses the related work briefly, Section III presents the high-level generic approach for software modernization, Section IV presents LLM-assisted functionality extraction approach for COBOL code, Section V shares COBOL application functionality extraction validation using case study, Section VI discusses the threats to validity, and Section VII concludes the paper with discussion.

II. Related work

Functionality extraction from code using LLMs is explored by many researchers. Toufique Ahmed et al. introduced a few-shot prompt template that contains contextual information extracted from the source code to generate summaries for code snippets [1]. Weisong Sun et al. evaluated LLMs for code summarization, focusing on evaluation methods, prompting, model configurations, and cross-language performance [2]. Nilesh et al. proposed two-step hierarchical approach for repository-level summarization for large scale functionality understanding [3]. Fangjian Lei et al. proposed a multi-agent approach using two LLM-based agents working collaboratively to generate code explanations of functions, files, and the overall project [4]. The code-specific LLMs, such as CodeLlama [5], StarCoder [6], and DeepSeek-Coder [7], have significantly improved summarization accuracy. For code summarization, Rukmono et al. have focused on component-level summaries by leveraging abstract syntax trees and using a chain-of-thought prompting technique with LLMs [8]. S. Yun et al. created artifact-specific prompts by using project-specific few-shot {code,summary} examples with a neural prompt selector, allowing it to better capture the unique nuances of a given codebase [9].

Overall, existing approaches have primarily focused on smaller code units, such as individual functions or methods, and often struggle to scale to larger codebases. As a result, their performance diminishes when applied to larger codebases that require a broader, more comprehensive understanding.

III. Modernization Approach

Typically, software implementation uses a specific *Functionality*, *Technology Stack*, *Technical Architecture*, *Solution Architecture* and *Design*. ***Functionality*** covers the capabilities that the software provides. This may cover various functional details of software such as business processes, rules, user interactions using screen etc. ***Technology Stack*** includes tools and technologies used for building the software i.e. programming languages, frameworks, libraries, and other software components. ***Technical Architecture*** describes the overall technical structure of the system, including how different components will interact and integrate with each other. ***Solution Architecture*** focuses on the high-level design of the solution, outlining how the system will be built to achieve the desired functionality and performance. ***Design*** covers the detailed design aspects of the system, including user interface design, user experience considerations, and other design elements. Modernized system implementation would require change in one or many of these dimensions.

To handle the large impedance mismatch between legacy and modern technology, we propose a three-stage approach for modernizing software as shown in Fig. 1 which comprises the three stages: **i) LLM-assisted Functionality Extraction**: This stage involves the use of LLMs to generate functionality description of the software system, with traceability to appropriate places in the codebase. Domain dictionary is used during this phase to map terms used in code with domain term. LLMs can assist in creating comprehensive and accurate

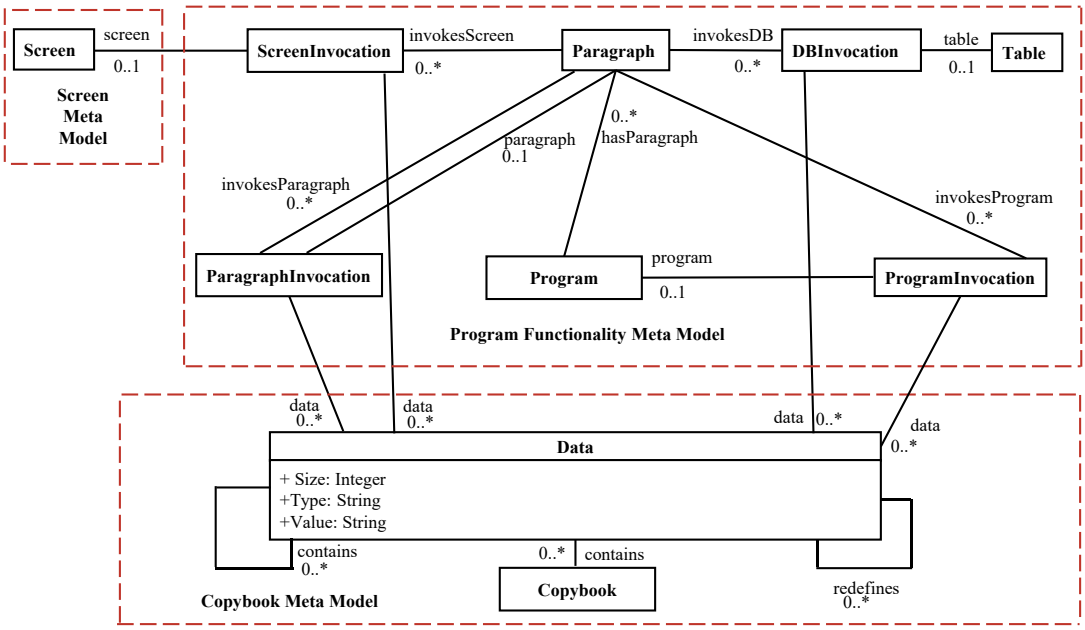

Fig. 2: COBOL Functionality Extraction Meta-Model

descriptions focused only on business functionality that can be reviewed for correctness by Subject Matter Experts (SMEs). **ii) LLM-assisted Staged Refinement**: In this stage, the generated functionality documents undergo staged refinement from textual form to a more structured specification including details of technology architecture, solution architecture, and design concerns. LLMs can assist in generating the refined specifications for these requirements. **iii) LLM-assisted Coding**: This stage involves forward-engineering the desired modern software for a specific technology stack using the refined functionality descriptions.

Overall, the approach leverages the capabilities of LLMs to aid human experts in software modernization, enhancing efficiency of the process from initial documentation to final implementation. In this paper we focus on the first stage of automation aided generation of functionality description specifically for COBOL code. The generated descriptions are manually validated for correctness.

IV. LLM-Assisted Functionality Extraction From COBOL Code Approach

A. COBOL Overview

COBOL application code typically consists of program files, copybook files. Program files consist of specific divisions for data and functionality. Copybook files are reusable COBOL code segments containing data definitions or statements, included in programs. COBOL is low resource, imperative language and provides English like syntax, whereas modern languages like Java are based on object-oriented paradigm. The syntax directed translation from COBOL to Java may produce logically correct code, but it will not be optimally modularized leveraging target language paradigm benefits. Such transformations lead to multiple maintainability challenges. Moreover, COBOL has several dialects that need creation of dialect specific transformers and increased maintenance of such translators. To address these issues there is need to extract only functionality details from COBOL code and regeneration of application using functionality.

B. COBOL Functionality Extraction Meta-Model

Fig. 2 shows the meta-model used for functionality extraction. A *Program* has multiple *Paragraph* for its functionality implementation, depicted with *hasParagraph* association. A *Paragraph* can invoke various *Program*, *Paragraph*, *Screen*, and *Table* is elaborated through invocation specific classes *ProgramInvocation*, *ParagraphInvocation*, *ScreenInvocation*, *DBInvocation* respectively. The *ProgramInvocation*, *ScreenInvocation*, and *DBInvocation* classes exchange data specified by data association with *Data* class. *Data* class represents composite structure of data elements. The redefinition of data elements is specified using *redefines* association. For traceability purpose data elements are associated with *Copybook*. *Data* has properties – *Name*, *Size*, *Type* and *Value*. *Paragraph*, *Program* and *Screen* have *Name*, *Code*, and *Description* property.

C. Functionality Extraction

Fig. 3 shows approach used for functionality extraction from COBOL application code. It is implemented in four steps: i) Preprocess ii) Extract Model iii) Validate Model iv) Generate Document.

Preprocess: Copybooks are classified into the data and statement classes by programmatically analyzing their contents. Statement copybooks are expanded within COBOL programs that reference them. Typically, code contains personal identification information of the author details that should not be sent with prompts. At times, comments written are not in sync with code. Hence, these are removed from the code in preprocess step.

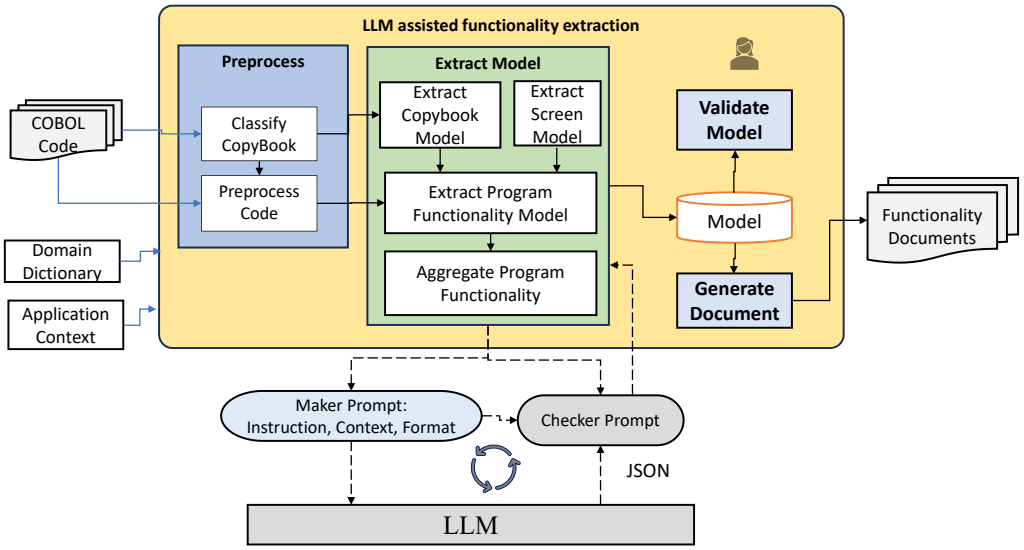

Fig. 3: LLM-assisted Functionality Extraction

Extract Model: To extract large size functionality model from large size COBOL code there is need for systematic interaction with LLM. We have leveraged our prior work on LLM4Model [10] to specify mapping of meta-model elements with prompt and instantiation of model using LLM generated JSON response and passing the instantiated model information to prompts. Extract step uses maker and checker prompts for interacting with LLM. For all classes, associations shown in the meta-model, purposive maker and checker prompts are defined. Extract model step broadly has following sub-steps: i) *Extract copybook Data Model* ii) *Extract Screen Model* iii) *Extract Program Functionality Model* iv) *Aggregate Program Functionality*.

Extract Copybook Model: In this step first copybook, Data objects and all associations links among these are extracted using purposive maker and checker prompt. These prompts maps to various COBOL datatype to common datatypes - *Alphabetic*, *Alphanumeric*, *Numeric*, *Floating Point*, *Packed Decimal*, *Binary*, *Boolean* etc. Unique terms are generated in this step and domain dictionary is prepared using these terms to use in subsequent extraction.

Extract Screen Model: Basic Mapping Support (BMS) files define the layout for user interfaces, mapping data fields to screen layouts for user interaction. In this step, functional information is extracted from Screen specific BMS files, using purposive maker and checker prompt.

Screen Description Prompt:
Given a <**ApplicationContext**>, provide a description of the purpose, behavior, detailed layout of the screen in form of visual of the BMS screen <**Screen.Code**>.

***Extract Program Functionality Model*:** The length of COBOL programs may exceed the input token limit of LLMs, making it impossible to send an entire program at once. To comprehend the functionality using LLM effectively divide-n-conquer strategy is adopted. COBOL program code is split into paragraphs and then program functionality model is extracted using paragraph code. Domain dictionary is used during this phase to map terms used in code with domain term. Program Functionality summary is extracted using paragraph summary and paragraph model. Model extraction is executed in the following sequence.

Program -> Paragraphs <Name, Code, Description>,

Paragraph->data->Data <Name>,

Paragraph -> invokesParagraph->ParagraphInvocation <Name>,

ParagraphInvocation -> paragraph->Paragraph <Name>,

Paragraph->invokesScreen->ScreenInvocation <Name>,

ScreenInvocation ->screen->Screen <Name, Code, Description>,

Screen->data->Data <Name>,

Paragraph ->invokesProgram->ProgramInvocation <Name>,

ProgramInvocation ->program->Program <Name>,

Program ->data->Data <Name>,

Paragraph ->invokesDB->DBInvocation <Name>,

DBInvocation ->table->Table <Name>,

Table ->data->Data <Name>

Paragraph description extraction prompt takes ApplicationContext and paragraph code as input and instructs to generate the high level summary excluding the programming specific details. Paragraph invocation extraction prompt takes list of paragraphs as input to identify which paragraphs are invoked with the given paragraph. Screen invocation, Program invocation, DB invocation prompts are given as input with a few shot examples as syntax of invocation varies across different variants of COBOL. The Checker prompts validate the response

Paragraph Description Prompt:
Given the paragraph code of the <**ApplicationContext**>. Given referred dictionary terms <**Subset Domain Dictionary**> , Given invoked screen title <**Screen title**> . ,The following is a list of paragraphs invoked -<**Para Invocations**>. The following is a list of programs invoked in the given paragraph: ,<**Program invocations**> Generate a very concise, clear functionality description. It should cover the business intent, business logic, and functionality specific details. Do not include code specific details like variable names or cobol statements inside the description. Make sure to include system messages, validation criteria or dependency on other program/paragraph if present in code. No bullets . Also, do not mention the domain name in your response. The paragraph code is as follows:<**Paragraph.Code**>

received from maker prompt for the semantic as well as syntactic check. To give an overall idea of prompts used for model extraction, few representative prompts text are shown below:

Screen Invocation Prompt:
In a COBOL program, screens are invoked using **only** the MAPSET() command.
The parameter passed to MAPSET() indicates the screen being invoked.
<Few Shot Examples>.
Screen Identification:
- Extract all screen names from the provided COBOL code snippet by identifying the value passed to the ONLY **MAPSET()** command.
- Compare the extracted list of screen names with the "Name" values in the JSON response.
TASK:
Given the COBOL code provided, <**Paragraph.Code**>.
Find all the screens that are invoked. Return an empty list [] if no screens are invoked.
Do not output anything else. No verbose ONLY JSON.

Program Invocation Prompt:
In COBOL programs, program are invoked as follows: **<Few Shot Examples>**

ProgramInvocation Identification
- Extract all program names from the provided COBOL code snippet by identifying the value passed to ONLY **PROGRAM()** or **CALL** command.
- Return the extracted program names in expected format. Ensure there are no duplicates in the output.
- Compare the extracted list of program names with the "Name" values in the JSON response.
TASK: Given the COBOL code of **<Paragraph.Name> :<Paragraph.Code>.** Identify the names of the program invoked . Return [] if you don't find anything.
No verbose ONLY JSON. pls remove leading and trailing spaces from Name value.

DB Invocation Prompt:
In COBOL paragraph code, database files are invoked with the following commands.
<Few Shot Examples>
** DBInvocation Table Identification**
- Extract all table names from the provided COBOL code snippet by identifying the value passed to the ONLY **DATASET()** command that deals with database file operations (e.g., READ, WRITE, DELETE).
- - Return the extracted table names in expected format. Ensure there are no duplicates in the output.
- Compare the extracted list of table names with the "Name" values in the JSON response.
- TASK:
In the <**Paragraph.Name**> COBOL paragraph code, find only the database files that are beings invoked. Return [] if no database files are invoked.<**Paragraph.Code**>.

DB Invocation Data Prompt:
In the COBOL program, database files are invoked as follows. <**Few Shot Examples>**
DBInvocation Data Identification
- Extract all data names from the provided COBOL code snippet by identifying the value passed to the ONLY **FROM()/INTO() RESP and RESP2** command.
- Confirm that the response only contains the data belonging to the specified database invocation.
- Return the extracted data names in expected format. Ensure there are no duplicates in the output.
- Compare the extracted list of data names with the "Name" values in the JSON response.....
TASK: Find the data of <**DBInvocation.Name**> database file. <**Paragraph.Code**>.Do not output any other data that doesn't belongs to <**DBInvocation.Name**>. <**Format Instruction>**

DB Invocation Data Checker Prompt:
You are a COBOL expert. Your task is to analyze the provided prompt and response. Check If the JSON response has missing double quotes around keys and string values. Please fix the JSON by adding the necessary quotes. If the provided response is not following what is expected in prompt., Then only correct the response. For analysis check following guidelines.
DB invocation specific Data Identification:
.....
Validation Criteria:
....
JSON validation criteria:
....
Expected Output Format:
1. If the JSON structure is valid and it correctly lists the data associated with the specified db invocation:
```json
{
"Data": [{ "Name": "DataName1", "Name": "DataName2" }]
}
```
Given the JSON response and the COBOL code snippet, carry out the validation checks and return your output in the required format based on these criteria.

Aggregate Program Functionality In this step, extracted descriptions and model from individual program paragraphs is used for generating an aggregated functionality description of the program using an LLM. The prompt instructs LLM to create a coherent, step-by-step narrative of the process flow to unfold it from the top-level paragraph details.

Program Description Prompt:
You are an expert in reverse engineering COBOL systems. You will be provided with a description of a program paragraph that details the interactions it has with other paragraphs, programs, screens, and databases. Using this information, create a coherent, step-by-step narrative of the process flow that unfolds from the top-level paragraph. Focus on the specific implementation details; instead, it should express the workflow , highlighting the business logical and connections between the various components. The flow should illustrate the overall functionality and sequence of events that occur within the program so that it can later be used modernized the application. Instruction to follow 1. Focus only on logical flow and functional behavior 2. Do not include any programming suggestion. 3. Mention screen/database access where relevant.4. Format output as a number or bullet point list of logical steps in the process. Description: <**Combined Paragraphs Descriptions of Program**>
Please provide a detailed project-level summary for <**Program Name**>.

Validate Model: As the digitalized model is large, it is a challenge to validate the model manually. To address this issue, a model sanity checker is developed to check various model validation rules on the model. Issues such as *duplicate copybook/programs/data, copybook without data, Non-group data having data, Invalid data types, Data without type, Program without paragraph, Program with duplicate paragraphs, Program data names same as copybook data name, Invocations without associated data, Invocations without target program/screen/table, Isolated paragraphs/data* are reported in sanity report. This model sanity report helps in manually correcting the model if there are any errors skipped through checker prompt.

Generate Document: Using extracted model two functionality documents are generated: i) Program specific functionality document. ii) Model traceability document.

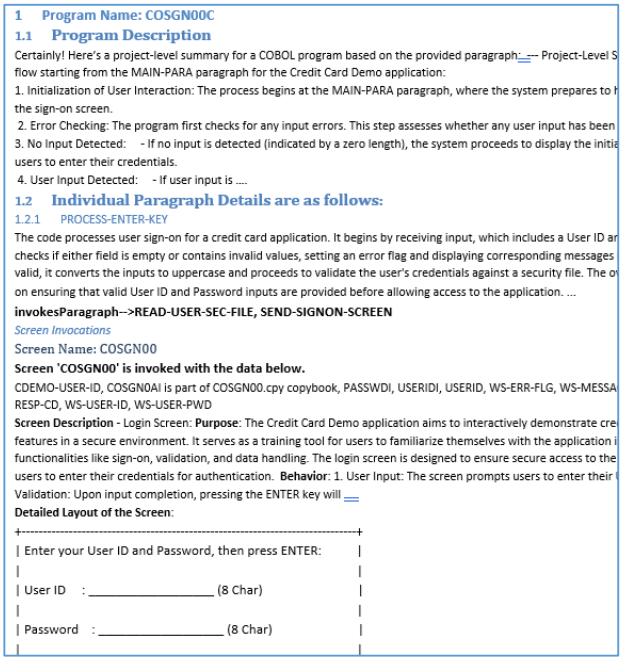
1 Program Name: COSGN00C
1.1 Program Description
Certainly! Here's a project-level summary for a COBOL program based on the provided paragraph: -- Project-Level S
flow starting from the MAIN-PARA paragraph for the Credit Card Demo application:
1. Initialization of User Interaction: The process begins at the MAIN-PARA paragraph, where the system prepares to h
the sign-on screen.
2. Error Checking: The program first checks for any input errors. This step assesses whether any user input has been
3. No Input Detected: - If no input is detected (indicated by a zero length), the system proceeds to display the initia
users to enter their credentials.
4. User Input Detected: - If user input is ....
1.2 Individual Paragraph Details are as follows:
1.2.1 PROCESS-ENTER-KEY
The code processes user sign-on for a credit card application. It begins by receiving input, which includes a User ID ar
checks if either field is empty or contains invalid values, setting an error flag and displaying corresponding messages
valid, it converts the inputs to uppercase and proceeds to validate the user's credentials against a security file. The o
on ensuring that valid User ID and Password inputs are provided before allowing access to the application. ...
invokesParagraph-->READ-USER-SEC-FILE, SEND-SIGNON-SCREEN
Screen Invocations
Screen Name: COSGN00
Screen 'COSGN00' is invoked with the data below.
CDEMO-USER-ID, COSGN0AI is part of COSGN00.cpy copybook, PASSWDI, USERIDI, USERID, WS-ERR-FLG, WS-MESSA
RESP-CD, WS-USER-ID, WS-USER-PWD
Screen Description - Login Screen: Purpose: The Credit Card Demo application aims to interactively demonstrate cre
features in a secure environment. It serves as a training tool for users to familiarize themselves with the application i
functionalities like sign-on, validation, and data handling. The login screen is designed to ensure secure access to the
users to enter their credentials for authentication. Behavior: 1. User Input: The screen prompts users to enter their
Validation: Upon input completion, pressing the ENTER key will
Detailed Layout of the Screen:
+--+
| Enter your User ID and Password, then press ENTER: |
| |
| User ID : _____ (8 Char) |
| |
| Password : _____ (8 Char) |
| |

Fig. 4: Sample Generated Document for a Program

Fig. 4 shows a sample generated document for a program. Program specific functionality documents contain program summary, paragraph specific heading containing paragraph descriptions, paragraph/program/screen/DB invocations and data associated with each invocation. Generated model traceability document provides complete traceability for each paragraph along with details such as – program name, paragraph name, screen invocation, program invocation, paragraph invocation, DB invocations along with data details. The generated documents are used for code generation. Due to space limit, it is not covered in this paper.

V. COBOL Case Study

A. COBOL Functionality Extraction Case Study

Functionality extraction approach is evaluated on Credit Card application written using IBM COBOL available in public domain [11]. It is a small yet complex enough application for validation. Table I summarizes the COBOL application file set. This application covers five components: i) Admin functionality covering user management ii) Account Management that enables users to view and update their account information iii) Card management functionality allow users to view card details, update card details, and see the list of cards iv) Transaction management functionality allows user to view transactions, add transactions, report transactions, see the list of transactions v) Bill Payment functionality enables users to pay their bills.

TABLE I. Input COBOL Files Summary

	Count
Number of Copybook Files	45
Number of BMS Files	17
Number of COBOL Files	17
Smallest Copybook File Lines of Code	10
Smallest COBOL File Lines of Code	261
Largest Copybook File Lines of Code	1317
Largest COBOL File Lines of Code	4237

Various screen types are used in application functionality, each reflecting different aspects of complexity. A menu screen displays a set of options for user selection, such as admin menu screens or main menu screens. Form-based screens enable users to enter data into specified fields, including tasks like updating accounts, adding users, or modifying user details. View screens organize information or content for users to read or analyze systematically. For example, account viewer screen, transaction view screen. Pagination type of screen divides content into separate pages, allowing users to navigate through large data. This format is often seen in lists. For example, list users, list transaction etc. Credit Card application covers these complex screens.

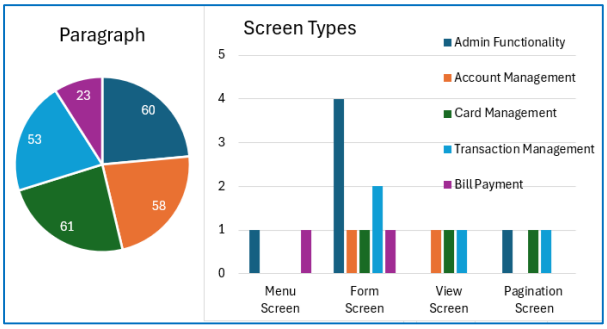

Fig. 5: Credit Card Application Component Summary

Fig. 5 shows a pie chart indicating the component-wise paragraph count and bar chart shows the different types of screens involved in functionality implementation to indicate the complexity of the application.

B. Evaluation

Domain terms are generated using copybook model and domain dictionary is manually prepared. Domain dictionary helps mapping the code elements with the real world domain terminology. For example, 'FLDT' maps to 'First Lien Deed of Trust', 'TRAN-CAT-BAL' maps to 'Transaction category balance' and so on. We used GPT-4o mini for the functionality extraction [12]. Large size model is automatically authored using automated LLM-assisted functionality extraction. Models authored are validated against well-formedness constraints using model sanity checking. For validating extraction further, generated traceability sheet is used. Authored functionality model is validated manually by comparing data in traceability sheet with the original code. Validation scores are assigned manually for every paragraph and its corresponding extracted associations (paragraph / program/ screen / DB invocations). A score of 1 is given for accurate extraction, 0.5 is given for partial accurate and 0 if given for inaccurate or missing data. Using the scores the accuracy is computed using the formula given below.

$$Accuracy = \frac{\sum(Validation\ Score)}{\sum(Paragraphs\ Validated)} * 100$$

TABLE II. Validation Score Results

Functionality	**Paragraph Description Accuracy (%)**	**Paragraph Invocation Accuracy (%)**	**Program Invocation Accuracy (%)**	**Screen Invocation Accuracy (%)**	**DB Invocation Accuracy (%)**	**Mean Accuracy (%)**
Admin	96.6	95	94.1	98.3	96.6	96.2
Account Management	87	83.6	100	96.5	100	93.4
Card Management	91.8	97.5	95.9	88.5	90.1	92.8
Transaction management	84.9	97.1	85.8	92.4	81.1	88.3
Bill Payments	91.3	93.4	89.1	86.9	65.2	85.2

Table II shows the accuracy details for all five components. The mean accuracy of ~91% is observed. The inaccuracies were observed in the generated outputs specifically related to missing paragraph invocations, dynamic program invocations, incomplete paragraph summaries, and incorrect data identification for invocations. For instance, the LLM identified "Varying" and "UNTIL" keywords as paragraph names for paragraph invocation as they appeared following the PERFORM statement. This specific syntax was not part of the few shot examples in the prompt. There were instances for dynamic data flow determining the names of program and data going beyond the paragraph code. Such cases resulted in incorrect invocation model creation. Additionally, instances of incomplete paragraph summaries were noted. For example, in the Transaction Management functionality, the generated summary of 'MAIN_PARA' paragraph of 'COTRN02C' program is incomplete " *The system evaluates user actions such as entering data, pressing function keys, or making selections. Depending on the users input, it processes transactions, clears screens, or copies last transaction data, and handles invalid key entries with appropriate messages*…". The summary generated did not cover the details of user input and actions. There were many instances of incorrect data attached to paragraphs. For instance, 'COACTUPC' program of Account Management functionality, the LLM incorrectly identified a paragraph name '1000-PROCESS-INPUTS' as a data element. This has happened as copybook data is not given as input to the prompt instead theses were resolved after receiving response from LLM. There were few cases in which the model could not identify paragraph invocations specified after GOTO statement.

LLMs effectively identified the exact layout, including all the labels, fields, and other elements of the screen as defined in the BMS file code. Few inaccuracies have been observed in the generated screen descriptions, specifically regarding whether the data is input by the user or auto-filled. For instance, in the Card Management functionality, the generated description states that "*the screen provides fields where users can enter their account number or card number to perform a search for specific card details.*" However, these inputs will not be entered by the user; they will be auto-filled. Similarly, in the Admin Functionality screen description, it mentions that "*several input fields are used to receive user input, such as transaction name, current date, current time, and program name.*" Again, these inputs will not be entered by the user, as they will also be auto-filled.

Many of the issues could potentially be fixed through prompt enhancements and additional few shot examples. However, some problems related to LLM's hallucinations, required human correction. A team of four completed the manual validation in three days, working nine hours per day, totaling 12 person-days of effort.

VI. Threats to Validity

A high-level, generic approach for software modernization is proposed, in which functionality description are extracted from code to facilitate further code generation. This method helps in eliminating the source language specific code structures in target language application. The functionality extraction part is specifically grounded for the COBOL language but is adaptable to any variant of COBOL. LLMs are made cognizant of COBOL language through few shot examples as needed, These examples may vary for different COBOL variants and are therefore externalized. The approach is designed to be generic, so as to work it across various application domains.

Validation has been carried out on a application that covers various complexities – different types of screen, screen interactions, multiple stakeholders. Code also covered syntactic complexity such as multiple usage of *GOTO* statements, dynamic function invocations, Customer Information Control System (*CICS)* invocation etc. Given the complexity and diversity of the sample COBOL programs we believe that approach can be used for other applications as well. The results for other applications may vary depending on how well the application code is structured. As work is yet to be deployed in production, usefulness results are not yet available.

The approach used the same LLM for maker and checker prompt that may have created bias towards the case study domain.

VII. Discussion

Legacy business applications, often built with languages like COBOL, present significant modernization challenges due to their limited functionality, lack of documentation, and shrinking pool of knowledgeable developers. Traditional parser-based and lift-and-shift modernization methods are often resource-intensive and fail to leverage the advantages of modern technologies. Recent advances in LLMs provide a promising pathway for automating and accelerating software modernization, as these models can analyze, summarize, and extract functionality from legacy code. However, challenges remain, including handling large and complex codebases, dealing with tightly coupled programs, and adapting to language-specific structures.

In this paper, a systematic, human-in-the-loop approach is proposed to address these issues. We explored LLM-assisted methodology for extracting functionality from COBOL code and validated on small yet complex business applications. Our approach automates a traditionally time-consuming and intellectually demanding extraction process. We combine MDE approach with LLM to take advantages of both. For model generation, we leveraged our prior work on LLM4Model to automate the process of generation of large-sized models. The efficacy of our approach was validated on small yet sufficiently complex business applications. Several challenges and insights emerged during the evaluation of our approach.

Considering the large size of COBOL program it was required to adopt divide-n-conquer approach. Program decomposition was done for all the program although few program size was less than the token length limit of LLM as we observed that when the complete program was given the LLM missed out important functionality details or only produced high-level description.

There were many instances for dynamic data flow going beyond the paragraph code. Such cases resulted in incorrect model creation. It was also observed that at time partial or abstract paragraph summaries were generated by the LLM. Errors could be minimized using maker and checker prompting

strategy, by incorporating application context. Few cases of inaccuracy were observed as LLM was not able to comprehend COBOL syntax. This could be related to unseen code pattern in LLM training and absence in the few-shot examples. There were some specific scenarios where data values are modified in different sections of the program. These cases could be identified by validate model phase and were manually corrected. For large sized copybook files, LLM was unable to process the complete file due to token length limit. To address this we had to split the files in logical part and was combined automatically during extraction process. The generated screen descriptions accurately reflected the layout, detailing all labels, fields, and other elements as specified in the BMS file code. In some cases, it was unclear whether particular data would be entered by the user or automatically populated. Overall it is observed that many of the issues could potentially be fixed through prompt enhancements and additional few shot examples. However, some problems related to LLM's hallucinations, required human correction. Domain dictionary helped mapping the code elements and declaration of variable names with the real world domain terminology. Generated document and traceability sheet helped in establishing the links with application code that not only helped in validation as well as served as input to subsequent steps of software modernization.

Functionality extraction is executed at paragraph level using screen/program/db invocation details. The approach worked very well for Card demo application due to meaningful naming convention for variables and paragraphs. If this is not the case for a COBOL application then either the mapping of section paragraph variable names to domain specific terminology will be required or concise description of called paragraph may help to provide relevant context information.

Different COBOL dialects exhibit multiple variations, which also depend on the technology stack used. These syntax differences present challenges for model extraction. The few-shot examples in prompts should be tailored to the specific technology stack and COBOL dialect being used. Using LLMs to generate prompts tailored to a COBOL dialect, instead of creating them manually, is an area that requires further research. We are exploring on combining parser based , MDE based and LLM based approach together for further accuracy improvement.

The effort required to validate functionality descriptions is minimized through the use of checker prompts and model sanity checks. Nevertheless, additional development is necessary to ensure the correctness, completeness, consistency, and relevance of functionality extraction outputs. The current approach uses GPT-4o-Mini for both generation and checking. In the future, we plan to use a LLM with better capability for the checker to improve validation process. In this paper we have only reported findings on functionality extraction that can be leveraged for the code generation.

In conclusion, we would like to say, large size software modernization is a time-, effort- and intellect-intensive activity. LLMs can be effective in reducing this burden. Validating on a small yet complex application with good results gives us confidence that LLM based approach can be applied widely across different applications and domains. Suitable domain contextualization, handling the size, handling the data-flows and automating functionality validation for correctness, completeness, consistency and relevance are some of future work.

REFERENCES

[1] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, “Automatic semantic augmentation of language model prompts (for code summarization),” in proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–13.

[2] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, “Source code summarization in the era of large language models,” arXiv preprint arXiv:2407.07959, 2024.

[3] N. Dhulshette, S. Shah, and V. Kulkarni, “Hierarchical repository-level code summarization for business applications using local llms,” in 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code). IEEE, 2025, pp. 145–152.

[4] F. Lei, J. Liu, S. Noei, Y. Zou, D. Truong, and W. Alexander, “Enhancing cobol code explanations: A multi-agents approach using large languagemodels,”arXiv preprint arXiv:2507.02182, 2025.

[5] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi,J. Liu, R. Sauvestre, T. Remezet al., “Code llama: Open foundation models for code,”arXiv preprint arXiv:2308.12950, 2023.

[6] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone,C. Akiki, J. Li, J. Chimet al., “Starcoder: may the source be with you!”arXiv preprint arXiv:2305.06161, 2023.

[7] Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang, R. Xu, Y. Wu, Y. Li, H. Gao, S. Ma et al., “Deepseek-coder-v2: Breaking the barrier of closed source models in code intelligence,”arXiv preprint arXiv:2406.11931, 2024.

[8] S. A. Rukmono, L. Ochoa, and M. R. Chaudron, “Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis,” in 2023 IEEE International Conference on Data and Software Engineering (ICoDSE). IEEE, 2023, pp. 7–12.

[9] S. Yun, S. Lin, X. Gu, and B. Shen, “Project-specific code summarization with in-context learning,”Journal of Systems and Software, vol. 216, p.112149, 2024.

[10] A. Rajbhoj, A. Somase, T. Sant, S. Vale, and V. Kulkarni, “LLM4model: Automated requirements specification model authoring,” in International Conference on Advanced Information Systems Engineering. Springer,2025, pp. 128–136.

[11] “Carddemo - mainframe credit card management application,” accessed:Apr. 10, 2025. [Online]. Available: https://github.com/aws-samples/aws-mainframe-modernization-carddemo.

[12] “Gpt-4omini:advancingcost-efficientintelligence.”[Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.