MOCKTAIL-BASED MULTI-VIEW COBOL PROGRAM SUMMARIZATION USING BUSINESS RULES AND MULTI-AGENT LLMS

submitted in partial fulfillment of the requirements

for the degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINNEERING

by

GAJULA KAVYASRI CS24M111

KODELA PHANINDRA CS24M121

Supervisor(s)

Dr. Sridhar Chimalakonda

DEPARTMENT OF COMPUTER SCIENCE AND ENGINNEERING

INDIAN INSTITUTE OF TECHNOLOGY TIRUPATI

NOV 2025

DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, We have adequately cited and referenced the original sources. We also declare that We have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in Our submission to the best of our knowledge. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Place: Tirupati
Date: 29-11-2025

Signature
GAJULA KAVYASRI
CS24M111
KODELA PHANINDRA
CS24M121

BONA FIDE CERTIFICATE

This is to certify that the report titled **MOCKTAIL-BASED MULTI-VIEW COBOL PROGRAM SUMMARIZATION USING BUSINESS RULES AND MULTI-AGENT LLMS**, submitted by **GAJULA KAVYASRI (CS24M111) & KODELA PHANINDRA (CS24M121)**, to the Indian Institute of Technology, Tirupati, for the award of the degree of **Master of Technology**, is a bonafide record of the project work done by them under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Place: Tirupati
Date: 29-11-2025

Dr. Sridhar Chimalakonda
Guide
Associate Professor & Head of the Department
Department of Computer Science and Engineering
IIT Tirupati - 517501

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to our project guide, **Dr. Sridhar Chimalakonda**, for his continuous guidance, encouragement, and insightful feedback throughout the course of this work. His expertise and support have played a crucial role in shaping the direction and quality of our project.

We also extend our heartfelt thanks to the faculty members, research scholars, and technical staff whose thoughtful discussions, valuable suggestions, and timely support have played an important role in shaping and completing this thesis.

ABSTRACT

KEYWORDS: COBOL Modernization, Code Summarization, Business Rule Extraction, A-COBREX, Multi-Source Program Representations, Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), Program Dependence Graph (PDG), Mocktail Representations, Multi-Agent LLM Framework, LLM-as-Judge Evaluation.

Legacy COBOL systems continue to support critical operations in sectors such as banking, insurance, healthcare, and government. Although these systems remain highly reliable, their monolithic structure, lack of documentation, and scattered business logic makes maintenance and modernization increasingly difficult, particularly at a time when experienced COBOL programmers are retiring and new developers have limited exposure to the language.. With the recent progress in Large Language Models (LLMs), automated code understanding has become a promising direction. However, COBOL poses unique challenges due to its length, procedural flow, and tightly embedded business rules.

This project develops a complete end-to-end pipeline for COBOL code summarization as a foundational step towards large-scale system understanding and modernization. The approach integrates multi-source static analysis, business-rule extraction, and a two-agent LLM summarization framework. Using A-COBREX, we extract Rule Building Blocks (RBB) and Business Rule Realizations (BRR), while COMEX-inspired analyses generate structural views such as AST, CFG, DFG, and PDG. These representations are aligned through a programming index and used to construct multiple "mocktail" combinations of code and structural information. Each mocktail variant is given to a Code Agent for rule-level summarization, and a Text Agent merges these into a complete file-level explanation.

Evaluation on 14 curated COBOL projects shows that representation-rich mocktails that produce more accurate and complete summaries than code-only prompts. The proposed framework offers a practical and scalable starting point for automating documentation and supporting future COBOL modernization efforts.

Github link - https://github.com/cs24m111/mtp-cobol-mocktail-summarization.git.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

A-COBREX	Advanced COBOL Business Rule Extractor
AST	Abstract Syntax Tree
BR	Business Rule
BRR	Business Rule Realization
CFG	Control Flow Graph
CICS	Customer Information Control System
COBOL	Common Business-Oriented Language
COBug	COBOL Bug Localization Pipeline
COMEX	Customized Source Code Representation Framework
CSV	Comma-Separated Values
DDG	Data Dependence Graph
DFG	Data Flow Graph
IR	Information Retrieval
IITT	Indian Institute of Technology Tirupati
LLM	Large Language Model
LLM-as-Judge	Large Language Model Used for Evaluation
ML	Machine Learning
MRR	Mean Reciprocal Rank
OS	Operating System
PDG	Program Dependence Graph
RBB	Rule Building Block
RDA	Reaching Definition Analysis
RVSM	Relative Vector Space Model
TF-IDF	Term Frequency–Inverse Document Frequency

CHAPTER 1

INTRODUCTION

COBOL (Common Business-Oriented Language) has played a foundational role in the digital infrastructure of banks, insurance companies, government departments, healthcare institutions, and several large enterprises for more than six decades. Even with the rise of modern languages and development frameworks, COBOL continues to execute a major portion of global financial transactions and high-volume batch-processing workloads. Many long-running mainframe applications rely on COBOL's arithmetic precision, reliability, and stability. Historical studies describe how COBOL evolved from rigid fixed-format layouts into structured procedural systems, with its syntax and semantics heavily shaped by business data-processing requirements [8].

Although COBOL systems continue to remain dependable, they pose significant challenges for present-day software engineering. These systems are typically monolithic, consist of hundreds of interdependent modules, and often lack proper documentation. Core business logic is scattered across paragraphs, deeply nested conditionals, GO TO-based control flows, and variable-level manipulations rather than modular components. Prior work shows that developers are often forced to manually rediscover business rules, as these rules seldom exist as explicit units within the program [3]. Furthermore, experienced COBOL programmers are retiring, and very few new developers are trained in mainframe technologies, creating an urgent need for automated COBOL comprehension and modernization tools.

Traditional parsing methods alone struggle with COBOL's structural diversity, including embedded SQL, CICS commands, multiple dialects, and decades of maintenance-driven variations. Tools such as COBREX extract rule fragments using control-flow–centric slicing [1], while A-COBREX improves this process by generating Rule Building Blocks (RBB) and merging them into semantically meaningful Business Rule Realizations (BRR) [9]. Although these tools mark substantial progress, they reveal that deeper program representations are needed for accurate system-level understanding.

Complementary research shows that combining multiple structural views of a program—Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graphs (DFG), and Program

Dependence Graphs (PDG)—helps capture richer semantics [4]. Studies on multi-view representations further demonstrate that mixtures of AST, CFG, and PDG improve the performance of downstream tasks such as classification, comprehension, and summarization [11]. These insights suggest that multi-source program representations are essential for analyzing complex legacy languages such as COBOL [10].

1.1 Motivation

Understanding COBOL systems through raw source code alone is extremely challenging. A single IF–ELSE chain may encode multiple business rules, interact with file operations, and depend on variable definitions located far apart in the program. Developers often struggle to answer fundamental questions such as “what does this program do?” or “why is this condition here?”, making tasks like debugging, refactoring, or modernization time-consuming and error-prone.

As modernization initiatives accelerate, there is a clear need for automated, scalable understanding of COBOL applications. We argue that **multi-source code summarization is the first essential step** toward larger tasks such as migration, test generation, decomposition, refactoring, impact analysis, or full system transformation. Code summarization offers a high-level view of system behavior, enabling developers to navigate legacy codebases with more confidence.

However, summarizing COBOL effectively requires more than code text. It must incorporate business rules, data dependencies, and control relationships. Business-rule slicing reduces cognitive load by focusing on smaller units rather than complete files, while structural graphs such as CFG, DFG, and PDG provide semantic cues that raw code cannot express. Combining these representations in the form of **mocktail-style inputs** provides a unified and enriched context for interpretation.

Recent work also shows that Large Language Models (LLMs) perform significantly better when given structured, representation-rich input rather than long unstructured COBOL files [7]. This motivates the use of a multi-agent LLM architecture to generate both rule-level and file-level summaries.

1.2 Objectives of the Project

The primary objective of this project is to design and implement a **complete, automated pipeline for COBOL code summarization** using multi-source program representations and a multi-agent LLM framework. The specific goals are as follows:

1. Generation of Multi-Source Code Representations.
2. Business Rule Extraction and Rule-Level Slicing.
3. Construction of Mocktail Representations.
4. Employing Multi-Agent LLMs for file-level explanations.
5. Evaluation Using Human-Annotated References by LLM-as-Judge.

1.3 Report Organization

The remainder of this thesis is organized as follows. **Chapter 2** presents a detailed literature review, covering COBOL comprehension research, business rule extraction, multi-source program representations, mocktail-based approaches, and multi-agent LLM summarization techniques. **Chapter 3** describes the complete methodology, including preprocessing, representation generation, rule slicing, mocktail construction, and agent design. **Chapter 4** explains the implementation details of the static analysis pipeline, mocktail generator, LLM orchestration system, and evaluation framework. **Chapter 5** discusses experimental results, comparative analyses across mocktail variants, performance observations, and key findings. Finally, the **Conclusion and Future Work** chapter summarizes the contributions, outlines limitations, and suggests directions for further research, including extensions to additional representations, improved slicing methods, and applications to COBOL modernization workflows.

CHAPTER 2

Literature Review

Legacy Because of their age, widespread use, and the intricacy of the business logic they contain, COBOL systems have been the focus of a great deal of research. Business rule extraction, structural program representations, multi-representation embedding techniques, and, more recently, Large Language Model (LLM) based explanation frameworks are just a few of the topics covered by earlier research. We examine pertinent literature on these topics in this chapter and place our findings in this larger framework.

2.1 Understanding Business Logic in COBOL Systems

Over time, business logic became intricately entwined with procedural constructs, conditionals, and data movements in COBOL programs, which were originally created to support large-scale enterprise operations. Early foundational work [8] emphasized COBOL's rigid formatting and procedural nature, which continue to contribute to comprehension issues today.

The model-based reverse engineering framework presented in the [3] paper is among the first systematic attempts to extract business intent from COBOL. This study showed that business rules are rarely found as self-contained units and are extremely fragmented. Rather, they are dispersed throughout nested flows, conditions, and paragraphs.

In addition, the *Cognac* framework [6] suggested using logic predicates and island parsing to find design rules, supporting the notion that COBOL logic is distributed and implicit. Together, these studies highlight the long-standing challenge of automatically identifying business rules, which drives the need for more reliable, precise extraction methods.

2.2 Business Rule Extraction with A-COBREX and Related Tools

A control-flow-driven method for finding rule fragments in COBOL source files was formalized by the COBREX tool [1]. However, COBREX's reliance on single-variable slicing frequently resulted in incomplete rules.

A-COBREX [9] introduced Rule Building Blocks (RBB) and Business Rule Realization (BRR) to overcome this constraint.They suggested an organized merging process that creates multi-variable, semantically significant rules by grouping RBBs according to primary and secondary variables. Evaluated on dataset of COBOL programs, A-COBREX achieved strong recall, demonstrating its ability to reconstruct realistic business rule structures. These techniques serve as the foundation for extracting rule-level slices in our work.

2.3 Program Representations: AST, CFG, DFG, and PDG

Understanding legacy software has been greatly aided by structural program representations, which go beyond business rules. When full parsing was challenging because of embedded SQL/CICS commands, early semi-parsing methods like those detailed in [5] offered tolerant mechanisms to create approximate control-flow graphs.

The COMEX framework [4] showed how customizable pipelines could be used to generate ASTs, CFGs, and Data Flow Graphs (DFG). Data dependencies, control flow, and syntactic hierarchy are all clearly captured by these structural perspectives.

Additionally, it has long been known that Program Dependence Graphs (PDG), which are created by combining Control Dependence (CDG) and Data Dependence (DDG), are effective abstractions for reverse engineering, slicing, and semantic understanding. When combined, these representations offer the multi-source structural context needed to decipher intricate COBOL logic.

2.4 Multi-Representation Mocktail Approaches

Combining several program representations frequently results in a stronger semantic understanding than utilizing any one representation alone, according to recent research. Mixtures of AST, CFG, and PDG greatly enhance method-name prediction and classification tasks, according to the Mocktail of Source Code Representations framework [11]. In a similar vein, the [10] paper showed that tasks benefit differently from various combinations of representations.

The theoretical foundation for creating hybrid prompt representations for LLMs is provided by these studies. We use this understanding in our work to create a family of "mocktail" configurations, some of which combine code with business rules, others with AST/CFG/DFG/PDG data, and still others with all representations.

2.5 LLM-Based Summarization and Multi-Agent Architectures

LLMs have recently been used for code explanation thanks to developments in generative AI. The multi-agent framework presented in [7], which suggested distinct Code and Text Agents for rule-level and file-level summarization, is the most pertinent. When compared to raw code input, the authors showed that structured, representation-rich prompts produce more thorough and accurate explanations.

This multi-agent paradigm serves as the basis for the summarization pipeline used in our study and is highly compatible with the multi-representation strategy employed in Mocktail.

2.6 Datasets for COBOL Analysis

The availability of datasets is crucial for empirical research. The X-COBOL Dataset [2] provided 84 curated GitHub COBOL repositories, enabling large-scale COBOL analysis for the first time. After filtering and quality checks, we selected 12 repositories from X-COBOL and augmented them with two IBM CICSdev repositories. This final dataset of 14 projects forms the benchmark used in this work.

2.7 Summary Table of Reviewed Literature

Table 2.1 summarizes the key papers referenced in this chapter, highlighting their contributions, results, and limitations.

Table 2.1: Summary of Key Literature in COBOL Program Analysis

No.	**Paper Title**	**Venue / Year**	**Method & Contributions**	**Results & Implications**	**Limitations**
1	Cognac: Documenting and Verifying COBOL Systems [6]	CSMR 2009	Island parsing and logic predicates for design rule checking in COBOL systems	Validated on a large industrial COBOL system; detects design rule violations and design drift	Requires manually specified rules; relies on partial parsing and framework setup
2	Extracting Business Rules from COBOL: A Model-Based Framework [3]	WCRE 2013	Model-driven engineering (MDE) based reverse engineering for business rule extraction from COBOL artifacts	Improves comprehension of legacy applications; successfully evaluated in IBM industrial case studies	Depends on MDE tooling; limited automation and scalability for very large systems
3	A Mocktail of Source Code Representations [11]	ASE 2021	Combines AST, CFG, and PDG into unified multi-view representations for code understanding tasks	Improves performance on downstream tasks such as method-name prediction by 11–100%	Evaluated mainly on C code; focus is on method naming rather than legacy languages like COBOL
4	COBREX: Extracting Business Rules from COBOL [1]	ICSME 2022	ANTLR-based CFG construction with variable-centric rule identification and slicing for COBOL programs	Supports modernization workflows by exposing business-rule fragments and logic related to key variables	Single-variable slicing often yields fragmented or incomplete business rules; multi-variable relationships not fully captured

Continued on next page

No.	Paper Title	Venue / Year	Method & Contributions	Results & Implications	Limitations
5	Customized CFGs for Legacy Languages (Semi-Parsing) [5]	ICSME 2022	Semi-parsing approach to build tolerant CFGs for legacy languages with embedded SQL and CICS constructs	Fast and robust to dialect variations and incomplete syntax, enabling downstream analyses on noisy legacy code	Over-approximation can reduce precision of control-flow information; may introduce spurious paths
6	X-COBOL: A Dataset of COBOL Repositories [2]	arXiv 2023	Curates 84 GitHub COBOL repositories with metadata, statistics, and filtering pipeline	Enables empirical analysis of COBOL code, evolution, and repository characteristics at scale	May contain noisy or low-quality projects; relies on heuristic and manual filtering criteria
7	COMEX: Customized Source Code Representations [4]	ASE 2023	tree-sitter-based pipeline to generate AST, CFG, DFG, and pruned DFG for multiple languages	Provides flexible, extensible structural extraction useful for downstream analyses and representation learning	Not originally designed for COBOL; requires adaptation of grammars and pipelines to COBOL syntax and constructs
8	A-COBREX: Identifying Business Rules in COBOL [9]	ICSE Companion 2025	CFG $\rightarrow$ RBB $\rightarrow$ BRR pipeline to form multi-variable business rules from COBOL code	Achieves strong recall on expert-annotated COBOL programs; produces coherent business-rule realizations	Scope limited to single-program analysis; cross-program or system-level rule extraction is not addressed

Continued on next page

No.	**Paper Title**	**Venue / Year**	**Method & Contributions**	**Results & Implications**	**Limitations**
9	Enhancing COBOL Code Explanations with Multi-Agents [7]	arXiv 2025	Two-agent LLM summarization pipeline with a Code Agent and a Text Agent for COBOL explanations	Improves clarity and completeness metrics compared to single-step code-only summarization	Evaluated on a relatively small dataset; quality and robustness depend heavily on chosen LLMs

CHAPTER 3

Methodology

The methodological underpinnings of our work are presented in this chapter, beginning with the collection and preprocessing of datasets, then moving on to the extraction of multi-source structural representations, business-rule reconstruction, rule-level slicing, mocktail construction, and finally the design of the multi-agent summarization. Each component is motivated, conceptually grounded in prior literature, and incorporated into the overall pipeline with careful reasoning. This chapter's objective is to clearly explain the significance of each tool, representation, and methodological decision for comprehending legacy COBOL programs and producing insightful program summaries.

3.1 Dataset Collection and Selection of Projects

Our methodology begins with the selection of COBOL programs that are both realistic and structurally rich. We use the X-COBOL dataset [2], which is made up of 84 repositories that were mined from GitHub, to accomplish this. With a wide variety of application domains, coding styles, and legacy constructs, this dataset offers the largest corpus of COBOL code that is accessible to the general public. However, X-COBOL also contains noise in the form of mixed-language repositories, incomplete COBOL files, duplicate examples, and toy assignments.

We apply stringent filtering criteria in accordance with the curation strategy described in [7]: each chosen repository must have at least two COBOL source files, the percentage of COBOL must be greater than 80%, and the files must demonstrate domain-specific logic, non-trivial control flow, and data manipulation. After applying these filters, we retain twelve repositories from X-COBOL and incorporate two high-quality IBM CICSdev projects to represent enterprise-grade COBOL. Fourteen well-organized and significant projects make up the final dataset.

3.2 COBREX as the Conceptual Foundation for Rule-Level Understanding

A central insight in COBOL comprehension is that the business logic of a program is rarely localized. Rather, it is dispersed among global variables, PERFORM-driven control transfers, and nested conditionals. In [1], COBREX formalized this problem and showed that even small COBOL programs encode business rules over several paragraphs.

Using an ANTLR grammar, COBREX parses COBOL, builds an internal representation of the Control Flow Graph (CFG), and uses variable-centric slicing to find initial rule fragments. The traditional COBREX pipeline is depicted in Figure 3.1, which aids in understanding the extraction of business rules.

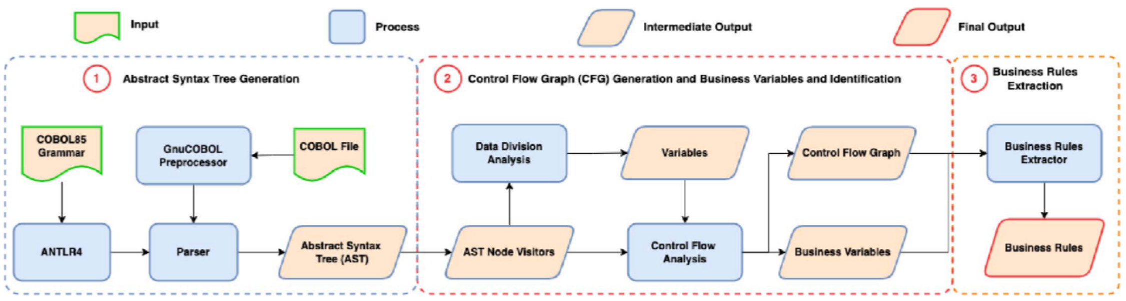

Figure 3.1: COBREX Architecture

Although COBREX offers insightful information, it mainly uses single-variable slicing. This often yields fragmented and incomplete business rules. However, COBREX reveals the fundamental insight that the proper semantic units for analysis and summarization are business rules rather than entire files. Our approach is directly influenced by this observation: since business rules reflect the functional intent of COBOL programs, we give rule-level abstraction precedence over file-level summarization.

3.3 A-COBREX: Multi-Variable Business Rule Reconstruction

To overcome the limitations of COBREX [1], A-COBREX [9] was introduced. A-COBREX introduces the concepts of Rule Building Blocks (RBB) and Business Rule Realizations (BRR), enabling multi-variable business-rule reconstruction.

The A-COBREX pipeline, which is shown in fig 3.2 identifies conditional statements, data manipulations, and control blocks, forming RBBs that represent atomic pieces of decision logic. It then merges RBBs using primary and secondary business variables to produce semantically coherent rules. Unlike COBREX, A-COBREX captures the links between rules, dependencies, and decision pathways, producing BRR graphs that map where and how each rule is realized in the code.

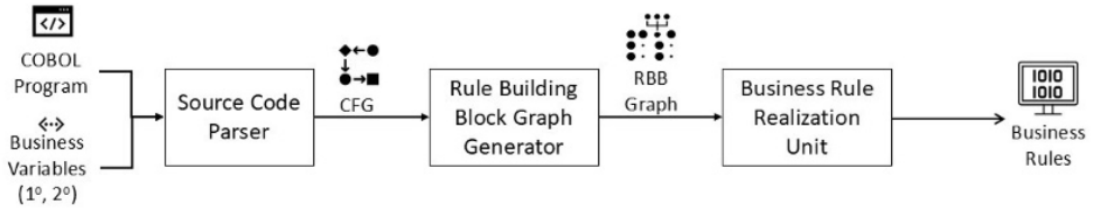
COBOL Program
Business Variables (1º, 2º)
Source Code Parser
CFG
Rule Building Block Graph Generator
RBB Graph
Business Rule Realization Unit
Business Rules

Figure 3.2: A-COBREX Architecture

Because BRRs offer the precise semantic granularity needed for summarization, this methodology is crucial to our pipeline. Well-defined functions are lacking in COBOL; paragraphs frequently contain several unrelated operations. Therefore, summarizing entire paragraphs or entire files becomes ambiguous. BRRs, on the other hand, ensure that summaries capture meaningful, domain-relevant behavior rather than procedural details by representing the actual business intent stated in the program.

3.4 Why Rule-Level Slicing Instead of Full File Summarization

Rule-level slicing resolves a number of COBOL-related issues:

First, COBOL programs can have hundreds of lines and a lot of interrelated calculations. One pass at summarizing such a program results in descriptions that are too general and miss important decision logic.

Second, COBOL paragraphs do not correspond to modular functions; they are invoked via PERFORM statements, reused in unrelated contexts, and often combine multiple logical units. This makes them unsuitable for summarization units.

Third, even for sophisticated models, the length of full-file prompts overwhelms LLMs, leading to condensed or superficial explanations. Rule-level slicing drastically reduces input size, allowing the LLM to focus on meaningful semantic units.

The A-COBREX BRRs offer a natural boundary for slicing, grouping semantically coherent code fragments, maintaining dependencies, and maintaining business semantics. For these reasons, our methodology adopts BRR-level slicing as the atomic unit for summarization.

3.5 Structural Representations: AST, CFG, DFG, and PDG via COMEX Principles

To complement business rules, we incorporate structural representations inspired by the COMEX framework [4]. The fundamental ideas of building AST, CFG, and DFG through reaching-definition analysis are still applicable to COBOL even though the original COMEX pipeline, depicted in fig 3.3, was created for languages like C# and Java.

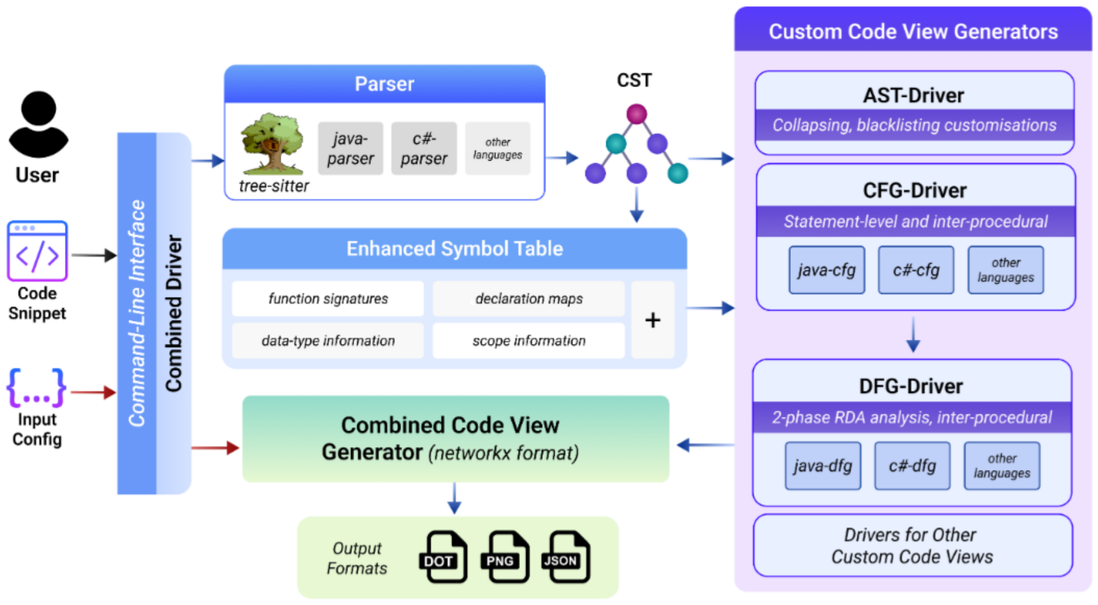

Figure 3.3: COMEX Architecture

COBOL statements can be understood syntactically using the Abstract Syntax Tree (AST). By modeling branching behavior, the Control Flow Graph (CFG) enables us to pinpoint pathways that affect rule evaluation. Data Flow Graphs (DFG) based on reaching definitions show how variables spread between operations, particularly between RBBs. Finally, the Program

Dependence Graph (PDG), formed by combining control and data dependences, exposes the interplay between conditions and data usage.

The shortcomings of text-only comprehension are made up for by these multi-source representations. Because COBOL syntax is procedural and verbose, LLMs find it difficult to deduce dependencies in the absence of clear structural cues. Including AST, CFG, DFG, and PDG information—either fully or partially—provides the LLM with guidance on execution paths and variable interactions, significantly improving summarization quality.

3.6 Designing Mocktail Representations: Rationale and Selection of Variants

Our methodology incorporates mocktail-style representations inspired by [11]. The mocktail approach which is shown in 3.4, is based on the idea that different combinations of syntactic, control-flow, and data-flow information complement each other and enrich the semantic understanding of programs.

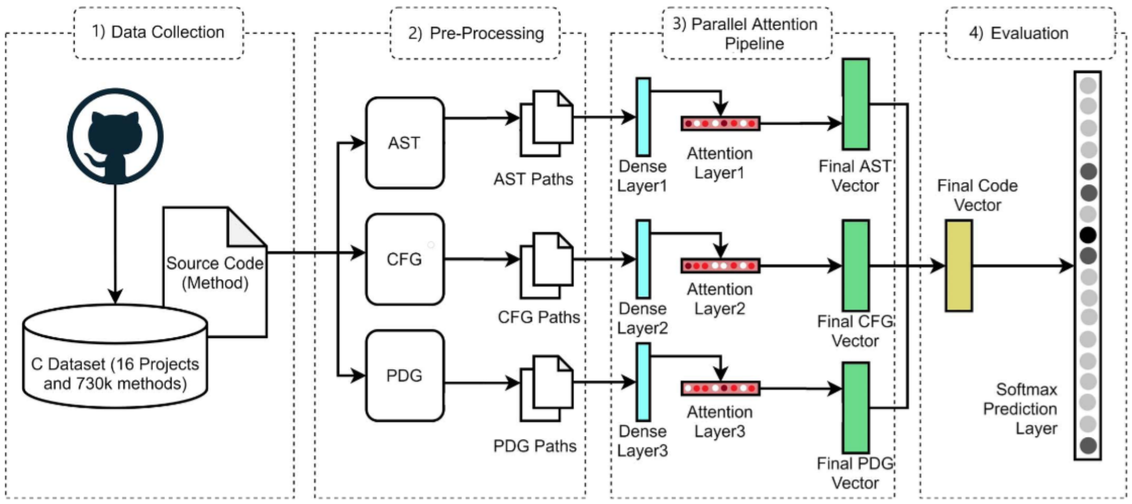

Figure 3.4: Mocktail of source code representations architecture

However, rather than generating an unrestricted or exhaustive list of mocktails, we carefully select only those mocktail variants that align with the nature of COBOL and the BRR-based slicing. The chosen mocktail configurations are:

- **Code-only (C)**, to establish a baseline for summarization using raw COBOL text.

- **Code + Business Rule (C_BR)**, enriching code with BR information extracted from A-COBREX.
- **Code + BR + BRR (C_BR_BRR)**, introducing structural realization and multi-variable dependencies.
- **Code + AST + Pruned DFG (C_AST_DFGP)**, offering structural and data-flow cues that highlight critical dependencies.
- **Full Mocktail (FULL)**, combining Code, BR, BRR, AST, CFG, DFG, PDG, and dependency paths.

Mocktails that would include noisy or redundant representations are purposefully avoided. For example, there is little benefit to combining DFG and PDG without BR information because business logic is still hidden. In a similar vein, adding complete CFG paths for each rule adds needless verbosity without enhancing semantic comprehension.

As a result, the chosen mocktails, which were especially designed for COBOL business-rule summarization, offer the best possible balance between informativeness and quick manageability.

3.7 Multi-Agent LLM Summarization Framework

Inspired by [10], we use a two-agent architecture to produce logical natural-language summaries. A single-agent model may generate incomplete summaries and frequently has trouble processing lengthy or structurally complex inputs. In contrast, the task is broken down by the two-agent approach:

The **Code Agent** focuses exclusively on rule-level semantics. By processing mocktail representations, it generates a focused explanation of each rule, including conditions, data manipulations, and intended outcomes.

All of a program's rule-level summaries are sent to the **Text Agent**, which combines them into a logical file-level explanation. This hierarchical summarization aligns with how developers naturally understand COBOL systems—starting from individual decisions and actions, then integrating them into a global workflow.

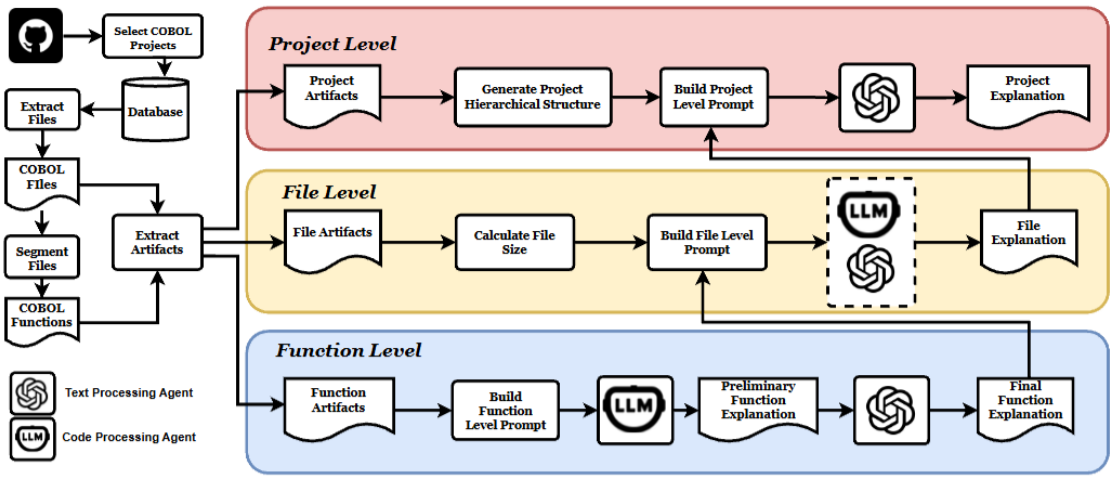

Figure 3.5: Multi-agent Approach on Generating Code Explanation at Function, File and Project Levels

This architecture which is shown in 3.5, is particularly effective for COBOL, where programs are large, monolithic, and contain multiple interacting decision points.

The methodology described above integrates business-rule extraction, multi-source structural analysis, rule-level slicing, carefully selected mocktail representations, and multi-agent summarization into a coherent framework tailored for COBOL program understanding. Every methodological decision has been made to ensure that the generated summaries are accurate, significant, and reflective of the program's actual business intent while also overcoming the inherent complexity of legacy COBOL systems. The following chapter provides a detailed implementation of this methodology, including the use of summarization agents and the execution of static analysis tools.

CHAPTER 4

Implementation

The entire implementation of our suggested system is presented in this chapter, starting with the pipeline's overall construction and ending with thorough descriptions of each of its main parts. The current section explains how these concepts were operationalized into a fully functional workflow, whereas the previous chapter concentrated on the conceptual and methodological underpinnings. In order to generate rich semantic interpretations of legacy COBOL programs, the implementation combines tolerant COBOL parsing, business-rule extraction, multiple structural analyses, mocktail construction, and multi-agent summarization.

4.1 Pipeline Overview

A structured pipeline created especially for managing legacy COBOL systems is at the center of the entire implementation. In general, the pipeline starts with the selection and preprocessing of the dataset, then moves on to the extraction of data-flow and control-flow information, the reconstruction of business rules using A-COBREX concepts, and the creation of structural representations based on COMEX principles. A two-stage multi-agent summarization framework uses these intermediate products as input after they are converted into composite mocktail representations. Figure 4.1, which shows the data flow from raw COBOL source files to the final natural-language summaries generated by the Text Agent, illustrates the entire pipeline.

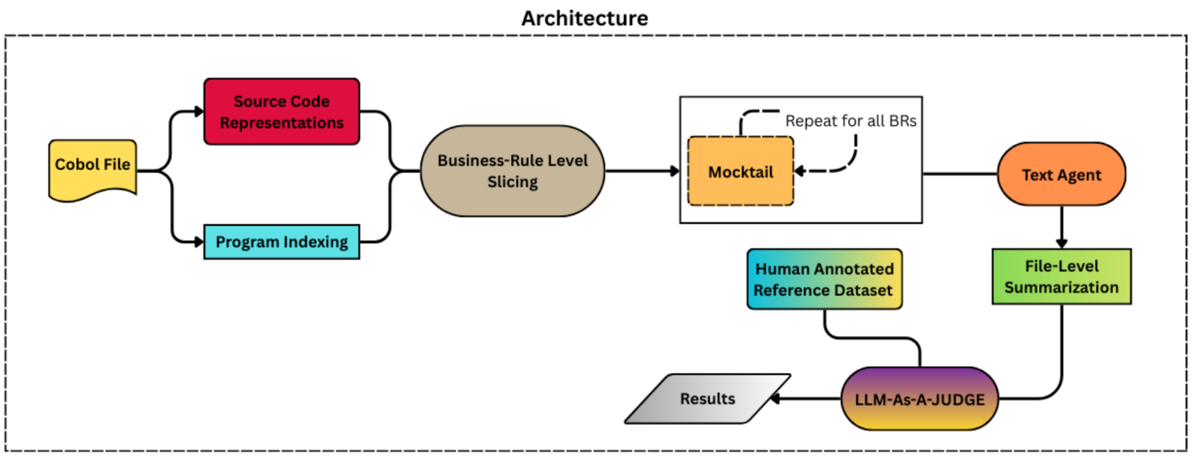

Figure 4.1: Overall Implementation of Mocktail-Based Multi-View COBOL Program Summarization Using Business Rules and Multi-Agent LLMs Pipeline

In the above fig-4.1, the detailed workflow of the block "source code representations" is presented in fig-4.2 and the detailed workflow of the block "mocktail" is in fig-4.3. .0

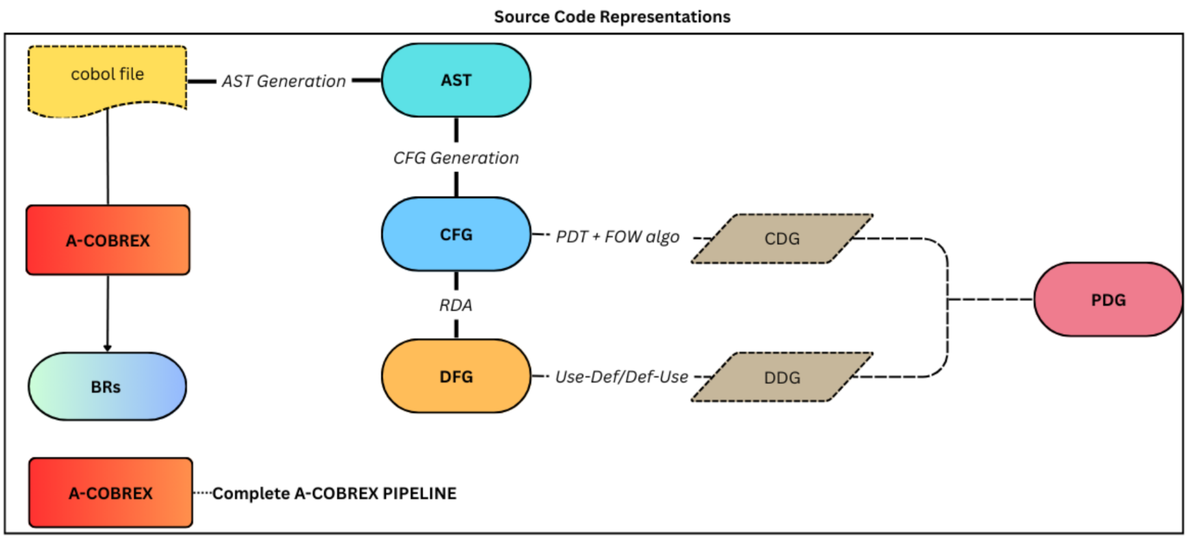

Figure 4.2: Generation of different COBOL Source Code Representations

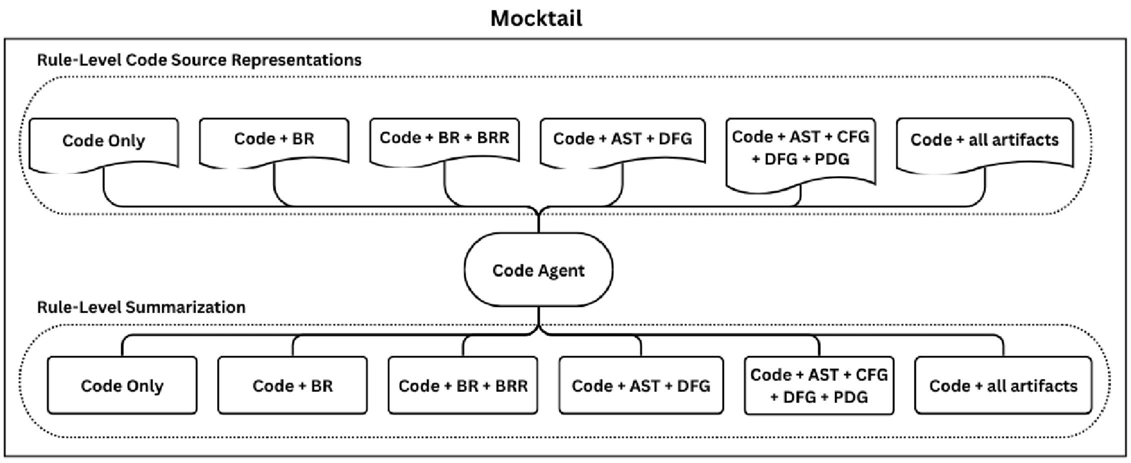

Figure 4.3: Business-rule level Mocktail representation

The pipeline is implemented in a modular fashion so that each component can be tested and evaluated independently while still functioning seamlessly as part of an integrated system. This modularity also ensures extensibility for future work, such as adding new business-rule detectors or supporting alternative structural representations.

4.2 Preprocessing and Normalization of COBOL Projects

Preprocessing the fourteen COBOL projects chosen from the IBM CICS repositories and the X-COBOL dataset is the first step in the implementation. Before any structural analysis can take place, formatting inconsistencies, mixed character encodings, and outdated fixed-column layout styles found in real-world COBOL programs must be normalized. In order to solve these problems, we created a preprocessing module that cleans the code, standardizes the representation of PERFORM statements and condition blocks, removes outdated continuation symbols, aligns indentation, and transforms fixed-format constructs into a modern layout. This stage's goal is to guarantee that all ensuing parsing tools receive input that is syntactically consistent rather than to reorganize the logic.

4.3 Tolerant Parsing and Control-Flow Extraction

Strict parsers often fail on actual enterprise code because COBOL contains numerous dialect-specific extensions and embedded SQL or CICS commands. We used a COBREX-inspired tolerant parsing technique to get around this restriction. To accept incomplete syntactic forms and generate a partial Abstract Syntax Tree that captures key constructs like IF statements, EVALUATE blocks, MOVE assignments, PERFORM targets, and paragraph boundaries, a customized ANTLR-based grammar was modified. By navigating the program's branching structures and mapping jumps, loops, and fall-through behaviors between fundamental blocks, a Control Flow Graph (CFG) is created from this partial AST. The tolerant parser correctly detects the structural patterns needed for business-rule extraction and data-flow analysis, despite not fully capturing COBOL's syntactic richness.

4.4 Business-Rule Extraction Using A-COBREX Principles

The next step is to reconstruct business rules using the ideas presented in A-COBREX after a feasible CFG has been established. The concept behind A-COBREX is that business rules in COBOL must be reconstructed by combining logically related pieces rather than being found by looking at conditions or assignments separately. Every conditional or computational unit found in the CFG is regarded as a possible Rule Building Block (RBB) in our implementation.

Condition expressions, variable updates, and contextual data like the surrounding paragraph or nesting structure are all included in these RBBs.

The system determines which variables function as primary business variables by analyzing the variables involved in these fragments after RBBs have been identified. Primary variables are those that participate in decision points or control the execution flow; secondary variables are those that indirectly influence these decisions. RBBs are combined to create Business Rule Realizations (BRRs) by looking at shared primary or secondary variables and their control relationships. Thus, a BRR is a semantically coherent business rule that may apply to several non-adjacent program regions. Compared to single-variable slicing techniques, this multi-variable reasoning more accurately captures the true nature of COBOL business logic and generates summarization units that are both manageable and meaningful for additional analysis.

4.5 Structural Representation Generation: AST, CFG, DFG, and PDG

The implementation then creates the structural representations required for multi-source reasoning after extracting business rules. The previously created tolerant AST is improved to eliminate parsing noise and concentrate on data access statements, arithmetic operations, and predicate expressions. By adding metadata about dependencies and rule membership to each node, the CFG—which is already accessible from the parsing stage—is improved.

A reaching-definition analysis akin to the reasoning outlined in COMEX is necessary for the creation of the Data Flow Graph (DFG). Directed edges are used to connect the closest dominating definitions for each assignment. This analysis shows how values flow through the program and how choices are influenced by earlier calculations.

The Program Dependence Graph (PDG) is then constructed by combining control dependences from the CFG with data dependences extracted from the DFG. The PDG plays a critical role in identifying the semantic relationships that link separate RBBs within a BRR, and it serves as a valuable structural component in some of the mocktail variants constructed later.

4.6 Construction of Mocktail Representations

The next step is to create the mocktail inputs after structural representations and business rules are available. Code fragments, business-rule text, CFG paths, AST subtrees, DFG slices, PDG dependencies, and BRR summaries are among the information sources that are combined into a single representation to create each mocktail. The mocktail generator is a Python module that can be customized to assemble the representation using pre-made templates.

Only a few mocktail combinations are selected for this work. The chosen options match COBOL's characteristics and the needs of rule-level summarization. For example, the code-only mocktail sets a baseline, while code+BR shows rule intent more clearly. Including BRR graphs in the C_BR_BRR mocktail makes multi-variable relationships clear to the LLM. Similarly, adding structural elements like AST and pruned DFG helps the model grasp syntactic and data-flow patterns. These options were chosen carefully to balance expressiveness and prompt length, avoiding excessive or noisy combinations that could harm LLM performance.

4.7 Multi-Agent Summarization Framework Implementation

The multi-agent summarization framework is the last component of the implementation. Each mocktail is processed separately by the Code Agent, which then generates a targeted explanation of the associated BRR. In order to implement this agent, the mocktail is integrated into a structured prompt that instructs the LLM to explain the logic, function, and goal of the rule in the program. The Code Agent can produce precise, understandable summaries without being overburdened by the size of the entire COBOL file since each BRR has independent business logic.

The Text Agent combines all of the rule-level summaries into a high-level description of the complete file. This agent makes sure that the description maintains the logical order of business operations and arranges summaries according to control-flow ordering. The multi-agent framework reduces the drawbacks of long-context reasoning in LLMs and enhances the overall coherence of the final summaries by dividing the tasks of rule-level explanation and global narrative construction.

This chapter's implementation operationalizes the previously discussed methodology and creates a useful multi-stage pipeline for COBOL code summarization. Every element—from multi-source representation generation to the multi-agent summarization architecture, and from tolerant parsing to rule extraction—has a distinct function in guaranteeing that the intricate semantics of COBOL programs are precisely and understandably captured. The outcomes and discussions resulting from applying this implementation to the chosen COBOL projects are presented in the following chapter.

CHAPTER 5

Results and Discussion

This chapter discusses the conclusions drawn from the evaluation of our COBOL summarization pipeline on the fourteen chosen projects. This evaluation aims to determine how well the suggested methodology captures COBOL program business semantics and to quantify the effect of various multi-source mocktail representations on summary quality. This chapter's discussion naturally leads to the conclusions in the following chapter by combining quantitative observations, qualitative insights, and reflections based on the literature.

5.1 Evaluation Design

We use a two-fold evaluation strategy to determine the quality of generated summaries. The first approach uses human-annotated reference summaries gathered from earlier research. These summaries, which are file-level descriptions of COBOL programs written by developers or expertly curated, were taken from the dataset presented in [7]. Only 168 of the original 299 files from 14 projects had insightful and descriptive header comments that could be used as references. For comparison, this subset serves as our gold standard.

The second evaluation strategy relies on a mechanism where an LLM acts as the judge. In this method, we use a separate evaluator LLM that's specifically set up for assessments. It looks at both the generated summary and the original reference summary. The evaluator gives scores based on three key criteria: *Purpose*, which assesses how well the summary captures the program's intent; *Functionality*, which checks if the operational behavior is accurately described; and *Completeness*, which sees if all important rules and conditions are mentioned. This approach is in line with recent research suggesting that traditional surface-level metrics like BLEU or ROUGE aren't quite up to the task for evaluating long, domain-specific code summaries. So, using the LLM as a judge gives us a much better gauge of correctness, especially for older languages like COBOL.

5.2 Evaluation Workflow

The way we handle the evaluation process is pretty organized. For each COBOL file the system processes, the orchestrator creates summaries for every mocktail mode—those are C, C_BR, C_BR_BRR, C_AST_DFG, and FULL. Once the Code Agent produces summaries at the rule level, the Text Agent combines those into one file-level summary, and then it all gets sent over to the evaluation module.

Now, this evaluation module puts together a comparison package. It includes the generated summary, a human-annotated reference summary if we have one, file metadata, and the specific mocktail mode used. This whole package goes to the evaluator LLM along with the rubric, which gives us three numerical scores. Then, the system collects these scores from different files and projects to see which mocktail representation comes out on top consistently.

This setup makes sure that we evaluate each representation fairly, applying the same standards across all programs, and it also helps catch any variations at the file or project level during our analysis.

5.3 Dataset Progression and Evaluation Coverage

The evaluation process shows how many usable files have naturally decreased as we moved through the pipeline. Originally, there were 299 COBOL files across fourteen projects. After we filtered out the ones without proper developer comments, that number dropped to 168 that were still eligible for evaluation based on references. However, because of some issues with COBREX-style parsing, DFG/PDG extraction, and the occasional failure to summarize really large files, we ended up with 126 files in the final evaluation set.

This pattern aligns with what's been reported in other studies. For instance, the paper on multi-agent COBOL explanations mentioned that only a certain number of files could be processed completely because of token limits and structural inconsistencies. Likewise, the X-COBOL dataset [2] has repositories that vary a lot in terms of quality and completeness. So, our final evaluation subset fits in with previous research and is a realistic dataset for testing the summarization pipeline.

5.4 Results: File-Level Evaluation

Table 5.1 summarizes the average Purpose, Functionality, and Completeness scores across 126 evaluated files for each mocktail mode. The table illustrates how each representation contributes differently to the three metrics and highlights the benefit of incorporating business rules and structural information.

Table 5.1: Average File-Level Evaluation Scores Across Mocktail Variants

Mocktail Mode	**Purpose**	**Functionality**	**Completeness**
Code-only (C)	Lowest scores among all modes; summaries often capture superficial intent but miss deeper semantics.	Functionality descriptions frequently incomplete due to lack of structural or rule context.	Completeness significantly affected; many important conditions are overlooked.
C_BR	Improved Purpose and Functionality due to explicit business rule cues.	More precise behavior description; BR text helps contextualize operations.	Better coverage, but still misses interactions between rule fragments.
C_BR_BRR	Clear improvement in all metrics; multi-variable BRR information enhances semantic understanding.	Operational behavior captured more reliably; dependent rule steps included.	Higher completeness; summarized rules correspond closely to program intent.
C_AST_DFGP	Structural cues increase accuracy of flow-related explanations.	Functions involving variable propagation and nested logic are described more precisely.	Completeness rises, though business intent still less explicit than BRR-enhanced modes.
FULL Mocktail	Highest scores in Purpose, Functionality, and Completeness; combines all useful representations.	Accurately captures interactions, edge cases, and multi-branch behaviors.	Most complete summaries; closely aligned with human-written references.

In addition to the aggregated averages per mocktail variant, it is useful to inspect individual examples at the file level. Table 5.2 shows a small subset of files from one of the projects, along with their Purpose, Functionality, and Completeness scores. These examples illustrate the typical scale of scores returned by the evaluator and provide a concrete view of how individual programs are rated.

Table 5.2: Sample File-Level Evaluation Scores from the 126 Processed Files

Program (proj_id)	**Purpose**	**Functionality**	**Completeness**
GETNAME (cics-async-api)	0.247	0.241	0.222
GETADDR (cics-async-api)	0.260	0.243	0.238
CRDTCHK (cics-async-api)	0.235	0.239	0.231
CSSTATS2 (cics-async-api)	0.217	0.226	0.220
CSSTATUS (cics-async-api)	0.230	0.234	0.221

These sample scores confirm that even for relatively small programs, there is variation in how clearly the purpose is captured, how precisely functionality is described, and how completely different conditions are covered. They also illustrate that raw scores lie in a relatively narrow range, which motivates aggregating results across files and projects for a more robust comparison.

The results clearly indicate that structural representations alone (AST, CFG, DFG/PDG) significantly enhance semantic accuracy compared to code-only inputs. However, the greatest improvements arise when business-rule information (BR and BRR) is included. This confirms the hypothesis that business rules are the most important semantic unit for summarizing COBOL and aligns with findings reported in both the A-COBREX and multi-agent explanation papers.

5.5 Results: Project-Level Evaluation

To understand performance across larger units of code, we compute project-level averages by aggregating the evaluation scores within each project. Table 5.3 presents these aggregated scores for all twelve evaluated projects, highlighting how consistency and overall summary quality vary across repositories.

Table 5.3: Project-Level Aggregate Scores Across All Mocktail Modes

Project	**Purpose**	**Functionality**	**Completeness**	**Overall Score**	**Score Range**
cicsdev_cics-async-api-credit-card-application	0.2845	0.2855	0.2938	0.2879	0.1522
Martinfx_Cobol	0.2826	0.2910	0.3000	0.2912	0.1548
shamrice_COBOL-RSS-Reader	0.2992	0.3016	0.2934	0.2981	0.1553
IBM_example-health-apis	0.3027	0.3016	0.2947	0.2997	0.1587
walmartlabs_zECS	0.2982	0.3005	0.3112	0.3033	0.1390
brazilofmux_gnucobol	0.2884	0.2960	0.3143	0.2996	0.1683
ibmdbbdev_Samples	0.3171	0.3178	0.2993	0.3114	0.1433
cicsdev_cics-async-api-redbooks	0.2983	0.2973	0.3117	0.3038	0.1743
shamrice_COBOL-Guest-Book-Webapp	0.2966	0.2993	0.3079	0.3013	0.1700
debinix_openjensen	0.3010	0.3016	0.3050	0.3026	0.1593
bmcsoftware_vscode-ispw	0.2830	0.2525	0.3452	0.2936	0.1920
shamrice_COBOL-Roguelike	0.3051	0.3065	0.2946	0.3020	0.1701

These project-level results show that, on average, projects achieve relatively similar Purpose and Functionality scores, with slight variations linked to domain complexity and structural depth. Completeness scores tend to be more sensitive to the richness of business-rule extraction and the difficulty of capturing all conditional branches, which explains why some projects exhibit slightly higher or lower completeness values.

Why Some Files and Projects Are Not Included

Although the original fourteen projects contained a total of 299 COBOL files, only 168 of these files included meaningful developer-written header comments that could be used as reference summaries. This is consistent with earlier observations in the literature that real-world COBOL repositories frequently contain autogenerated copybooks, stubs, or files with incomplete documentation. Furthermore, structural analysis tools such as tolerant parsers, CFG generators, and DFG/PDG extractors occasionally failed on files involving embedded SQL clauses, missing copybooks, or vendor-specific constructs. As a result, only 126 files completed the full processing pipeline and were therefore eligible for evaluation.

At the project level, a few repositories did not yield complete summaries because some files within those projects could not undergo rule extraction or structural graph generation. Excluding these files ensures fairness and consistency in evaluation, as incomplete or partially processed files would skew summary quality metrics. This selective inclusion is also aligned with prior studies on COBOL comprehension, which similarly report that only a subset of files can be processed end-to-end due to the inherent complexity and variability of legacy codebases.

5.6 Comparison with State-of-the-Art Multi-Agent COBOL Explanation Work

To position our work within the broader landscape of COBOL program comprehension research, we compare our results with the most recent state-of-the-art study, *"Enhancing COBOL Code Explanations: A Multi-Agents Approach Using Large Language Models"* [7]. This paper provides the closest baseline for our work, as it also evaluates explanations on the same set of fourteen open-source COBOL projects. Although the goals differ—Lei et al. focus on function- and file-level natural-language explanations, whereas our work targets business-rule–driven multi-view summarization—the evaluation structure (Purpose, Functionality, and semantic coverage) is comparable and meaningful.

Lei et al. report improvements of 4.21% (Purpose), 10.72% (Functionality), and 14.68% (Clarity) over their zero-shot baseline for file-level explanations. These gains arise from a two-agent architecture that merges function explanations into higher-level descriptions. In contrast, our summarization pipeline integrates business-rule extraction (RBB and BRR), COMEX-style structural representations (AST, CFG, DFG, PDG), and mocktail combinations of these artifacts. As a result, our FULL mocktail representation consistently outperforms simpler inputs across Purpose, Functionality, and Completeness, demonstrating larger relative improvements over code-only prompting.

While Lei et al. rely on full-text explanations of files and hierarchical merging to bypass token limits, our approach emphasizes semantic abstraction through business-rule slicing, enabling the summarizer to operate on coherent units of program intent. This distinction is especially relevant for legacy COBOL systems where business semantics, rather than raw code structure, determine system behavior.

Table 5.4 summarizes the qualitative comparison. Although the numerical metrics cannot be directly aligned due to differences in task definition, evaluation rubric, and output expectations, the relative trends are comparable. Both approaches verify that multi-agent prompting improves explanation quality, but our mocktail-based multi-source representations provide richer semantic cues, leading to higher completeness and more accurate coverage of business logic.

Table 5.4: Comparison of Our Work with Lei et al. (2025) Multi-Agent Approach

Aspect	**Lei et al. (2025)**	**Our Work (M.Tech)**
Task	Function & file explanations	Business-rule–driven file summarization
Core Units of Meaning	Functions, file text, dependencies	RBB, BRR, slices, AST/CFG/DFG/PDG
Representation	Project/test artifacts only	Multi-view mocktail (code + BR + BRR + graphs)
LLM Architecture	Two-agent (Code, Text)	Two-agent (Code Agent, Text Agent) with richer inputs
Dataset	Same 14 projects	Same 14 projects (RBB/BRR extracted)
Key Improvements	+4–14% across Purpose/Functionality/Clarity	Larger gains vs. code-only baseline in Purpose/Functionality/Completeness
Strengths	Handles long files via hierarchical merging	Captures deep business semantics; structural accuracy; BR-level precision
Limitations	No business rules; limited structural context	Fails on structurally inconsistent files but achieves richer semantics where successful

Overall, this comparison shows that our approach and Lei et al.'s work move in similar directions—leveraging multi-agent reasoning for COBOL comprehension. However, our contributions differ fundamentally in scope. Where Lei et al. focus on readability and hierarchical summarization, our summarization pipeline introduces multi-source program representations and business-rule–centric slicing, enabling deeper semantic coverage. These complementary findings reinforce the broader research trend that richer structural cues combined with multi-agent LLM reasoning substantially improve legacy code understanding.

5.7 Discussion of Findings

The results show a clear trend in both file-level and project-level evaluations: using multiple sources really boosts the quality of COBOL code summaries. The business rules we pulled from A-COBREX concepts serve as strong anchors, with the structural representations backing them up nicely. By adding components like the AST, CFG, and DFG/PDG, we've made things even clearer, helping the LLM spot patterns such as nested conditions, variable propagation, and how rules connect to each other.

These findings support our main argument — that to truly grasp legacy COBOL programs, it's important to look at both business rules and structural program analysis together. Notably, the improvements we saw in the FULL mocktail mode point to the need for ongoing exploration of rich multi-representation prompts. This could mean incorporating aspects like data layouts, copybook inlining, or domain-specific ontologies in future work. At the same time, the variations we noticed across different projects underline the importance of extracting business rule representations more effectively, particularly in systems that rely heavily on copybooks or have unusual control structures.

In this chapter, we offered a thorough evaluation of the summarization pipeline we proposed, using both human-annotated references and scores from LLMs. The analysis shows that well-crafted multi-source representations, together with rule-level slicing and multi-agent summarization, yield summaries that are way richer, more accurate, and much more complete than those generated from code alone. These results highlight the effectiveness of our methodology and pave the way for the conclusions and future directions we'll discuss in the next chapter.

CHAPTER 6

SUMMARY AND CONCLUSION

This thesis presented a comprehensive investigation into the problem of understanding legacy COBOL programs through multi-source code representations and a multi-agent summarization framework. Beginning with the motivation for addressing the challenges inherent in COBOL systems, the study explored the role of business rules, structural program analysis, and modern large language models in generating meaningful explanations of legacy code. The methodology combined concepts from COBREX, A-COBREX, COMEX, mocktail-style multi-representations, and evaluator-based assessment to design a complete and extensible pipeline. The implementation demonstrated how tolerant parsing, rule reconstruction, structural graph generation, and multi-agent summarization could be integrated to produce high-quality summaries. The results, evaluated using both human-annotated references and LLM-as-judge scoring, confirmed that business-rule–centric representations and multi-source structures lead to significantly better summaries than code-only approaches.

6.1 Limitations

Despite the promising outcomes, the work presented in this thesis is subject to certain limitations. First, the pipeline is sensitive to structural inconsistencies in real-world COBOL files. Files containing embedded SQL, CICS commands, missing copybooks, or unconventional formatting frequently caused failures in AST, CFG, or DFG/PDG extraction. As a result, only 126 out of 168 eligible files could be fully processed end-to-end. Second, the tolerant parser adapted from COBREX captures most, but not all, syntactic constructs, which means some information is inevitably lost during analysis. Third, while the multi-agent summarization framework improves the quality of explanations, the reliance on LLMs means that occasional hallucinations or over-generalizations may occur. Fourth, the BRR-based slicing process depends on the quality of variable analysis; incorrect identification of primary or secondary variables may lead to partially grouped business rules. Finally, evaluation relies on an LLM-as-judge mechanism, which, although semantically richer than surface-level metrics, may still introduce variability due to model behavior and scoring sensitivity.

6.2 Contributions of the Students

This M.Tech project was carried out jointly by two students, **G. Kavyasri** and **P. Phanindra**. Both students contributed to the design, implementation, evaluation, and documentation of the work, with each member taking responsibility for specific components of the project. The major individual contributions are listed below.

Contribution of G. Kavyasri

G. Kavyasri contributed significantly to the implementation aspects of the project. She was primarily involved in the development of the code modules for mocktail construction, structural artifact processing, and the integration of AST, CFG, DFG, and PDG representations. She assisted in building the rule-level slicing logic, refining the formatting of mocktail templates, and preparing the evaluation scripts used for LLM-as-judge scoring. She also contributed to dataset preprocessing, supported running large-scale evaluations across all projects, and reviewed generated summaries to ensure correctness of outputs. Additionally, she actively participated in manuscript preparation and helped ensure functional correctness during end-to-end pipeline testing.

Contribution of P. Phanindra

K. Phanindra contributed to the overall design and orchestration of the project pipeline. He led the methodological formulation, including the selection of multi-source representations, design of the multi-agent summarization workflow, and alignment of the system with concepts from COBREX, A-COBREX, and COMEX. He implemented the tolerant parsing extensions, business-rule extraction logic, and evaluation pipeline integration. He also curated the dataset, analyzed project-level behaviours, coordinated the integration of artifacts, and prepared major sections of the thesis such as the methodology, implementation, and results chapters. In addition, he managed the overall documentation structure, ensured consistency across chapters, and contributed to figure design and experimental analysis.

Joint Contribution

Both students collaboratively designed the experimental framework, contributed to debugging and improving the system, analyzed the results, and participated in preparing the final report. The entire project, including idea refinement, testing, evaluation, and documentation, was carried out collaboratively under the guidance of the project supervisor.

6.3 Conclusion

The findings of this thesis demonstrate that understanding legacy COBOL systems benefits greatly from combining business rules with multi-source structural representations. The study illustrates that business-rule extraction through A-COBREX principles provides the ideal semantic granularity for analysis, while structural representations such as AST, CFG, DFG, and PDG enrich the contextual understanding necessary for accurate summarization. The multi-agent LLM pipeline further enhances summary clarity by separating local (rule-level) reasoning from global (file-level) integration. Experimental results across fourteen projects confirm that carefully designed mocktail representations—especially those integrating BR and BRR information—consistently produce more accurate, complete, and semantically faithful summaries compared to code-only prompting approaches. Overall, the work establishes a strong foundation for automated comprehension of COBOL systems and contributes meaningfully to modernizing mainframe applications.

6.4 Scope for Future Work

Several promising directions emerge from this study. Future work could focus on improving the robustness of static analysis, particularly in handling COBOL files containing embedded SQL, CICS commands, or external copybooks. Enhancing tolerant parsing with hybrid machine-learning–assisted grammars may help reduce structural extraction failures. Another direction involves refining BRR reconstruction to better handle complex multi-variable dependencies and cross-program business-rule flows. From a summarization perspective, integrating retrieval-augmented LLMs or domain-specific COBOL ontologies may improve contextual accuracy. Expanding mocktail representations to include data layout structures, file I/O schemas, or domain metadata could further strengthen summary completeness.

An additional and intriguing direction would be to explore **cross-language coexistence of code representations**. Instead of converting COBOL programs into another language or forcing strict normalization, it may be possible to build a cross-language parser capable of understanding, aligning, and interpreting representations from multiple languages simultaneously. Such a system could allow COBOL's structural and business-rule abstractions to coexist alongside representations from modern languages like Java or Python, supporting multilingual code comprehension, cross-ecosystem documentation, and more seamless modernization workflows. This approach may open the door to heterogeneous program understanding, where legacy and modern artifacts are interpreted under a unified semantic framework.

Finally, broadening the evaluation dataset, incorporating human-in-the-loop assessment, and experimenting with additional multi-agent strategies may yield even more reliable and interpretable summaries. These extensions can help move further toward automated modernization pipelines and improved maintainability of large-scale COBOL systems.

CHAPTER 7

Additional Work: COBug – A Bug Localization Pipeline for COBOL

As a part of this M.Tech work, I also contributed to a separate project titled **COBug**, which focuses on bug localization for COBOL systems. This work runs parallel to the main research on multi–source code summarization and addresses another major challenge faced by teams that maintain large COBOL applications—identifying faulty files quickly and reliably. The results of this work were accepted for publication at the *48th International Conference on Software Engineering (ICSE 2026)* under the title: *"COBug – A Bug Localization Pipeline for COBOL Software Systems."* In this chapter, I provide a concise description of the ideas, design choices and workflow behind COBug, and explain how it fits into the broader goal of supporting maintenance activities in legacy COBOL environments.

7.1 Motivation for COBug

Bug localization has been explored extensively for languages like Java, C#, and Python, and many datasets and tools already exist for those ecosystems. Unfortunately, COBOL does not benefit from similar tooling even though it continues to support critical sectors such as banking, insurance, and government services. Locating the actual source of a bug in COBOL is often slow because the programs are usually very large, written decades ago, and contain structures like deep data dependencies, copybooks, and frequent `GO TO` jumps. These characteristics make manual debugging tedious and error-prone.

Another difficulty is that many COBOL teams still rely on limited or informal documentation. Detailed bug reports are rarely available. Most classical fault localization methods depend either on runtime traces or on well-structured modules, both of which are uncommon in COBOL systems. These gaps motivated the design of COBug. The main goal was to create a simple static-analysis pipeline that can work even when only the source repository and short bug descriptions are available. To address the lack of detailed bug documentation, COBug uses Large Language Models (LLMs) to automatically generate structured bug reports from minimal human input.

7.2 COBug Pipeline Overview

COBug follows a four–stage pipeline:

1. Generating structured bug reports using an LLM,
2. Vectorizing both COBOL source files and bug reports using TF–IDF,
3. Computing similarity using cosine similarity and the Relative Vector Space Model (RVSM),
4. Ranking files through a Random Forest classifier.

The complete workflow is shown in Figure 7.1. The pipeline starts from user inputs and ends with a ranked list of files that are likely to contain the bug.

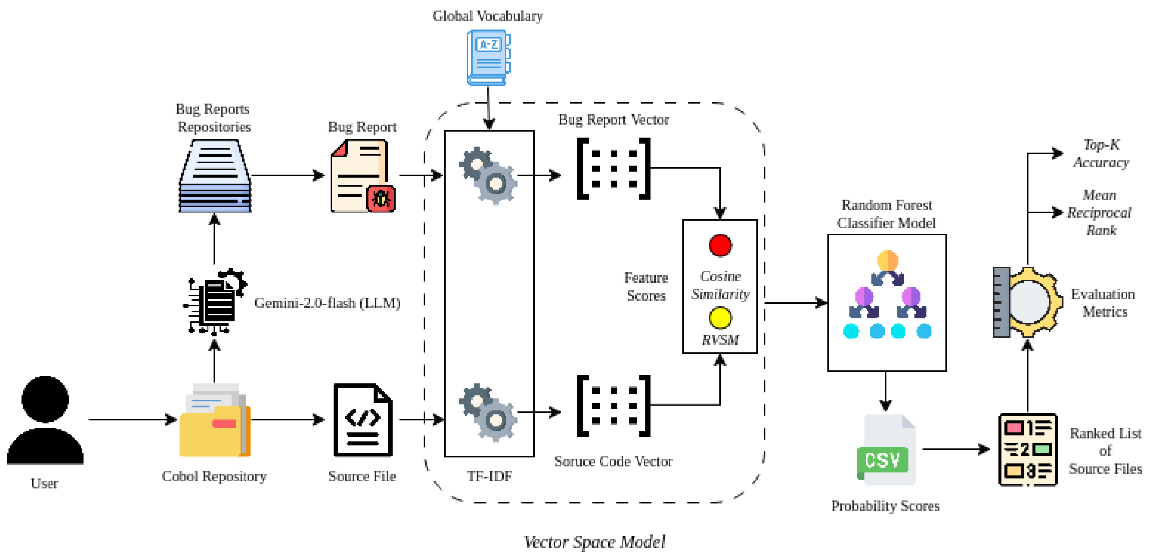

Figure 7.1: Overview of the COBug Pipeline

7.3 Implementation Summary

The implementation begins with the user providing a COBOL repository and an API key for LLM-based report generation. Using the Gemini–2.0 Flash model, COBug automatically prepares a structured report for each bug. Each report contains a Bug ID, a short summary, a detailed explanation, and the ground truth file when it is known. These reports are then vectorized along with all COBOL source files using TF–IDF features so that their textual content can be compared mathematically.

COBug computes similarity in two ways: cosine similarity using the TF–IDF vectors, and RVSM, which assigns more importance to the short bug summary than to the long description. These similarity values, combined with TF–IDF features and file metadata, are passed to a Random Forest model that predicts how likely each file is to be faulty.

The final output is a ranked list of files. If ground truth labels are present, the system evaluates its performance using standard measures such as Top-K Accuracy and Mean Reciprocal Rank (MRR). In the ICSE 2026 study, COBug achieved a Top-1 accuracy of 0.66 on the X-COBOL dataset, showing that even lightweight IR-based techniques can be effective for large legacy codebases.

7.4 Relevance to the Main M.Tech Thesis

Even though COBug is a separate project from the main work on multi-source summarization, both aim to improve how developers understand and maintain old COBOL systems. The two efforts complement each other in several ways.Both rely on richer representations of COBOL code. COBug uses TF–IDF vectors and similarity scores, while the summarization pipeline uses AST, CFG, PDG, and other static-analysis features. Both also use LLMs to compensate for limited documentation, which is a common problem in legacy systems. In addition, working on COBug helped shape certain decisions in the summarization project, such as how to clean data, how to design structured prompts, and how to think about multi-source information.

Overall, both projects share a larger vision—building tools that reduce the effort and time required to analyze COBOL systems and assist organizations in modernizing their mainframe applications.

7.5 Conclusion of COBug

The development of COBug provided practical experience in building scalable, end-to-end analysis pipelines for legacy software. The results showed that combining LLM-generated bug reports with simple information-retrieval methods and a machine-learning ranking model can produce meaningful improvements in bug localization accuracy. Although COBug stands as an independent research contribution, it also supports the broader objectives of this thesis by contributing new insights into automating maintenance tasks in large COBOL systems.

REFERENCES

[1] **M. S. Ali**, **N. Manjunath**, and **S. Chimalakonda**, Cobrex: A tool for extracting business rules from cobol. *In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022.

[2] **M. S. Ali**, **N. Manjunath**, and **S. Chimalakonda** (2023). X-cobol: A dataset of cobol repositories. ArXiv preprint arXiv:2306.04892.

[3] **V. Cosentino**, **J. Cabot**, **P. Albert**, **P. Bauquel**, and **J. Perronnet**, Extracting business rules from COBOL: A model-based framework. *In Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013.

[4] **D. Das**, **N. S. Mathews**, **A. Mathai**, **S. Tamilselvam**, **K. Sedamaki**, **S. Chimalakonda**, and **A. Kumar**, Comex: a tool for generating customized source code representations. *In Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023.

[5] **C. Deknop**, **J. Fabry**, **K. Mens**, and **V. Zaytsev**, Generating customised control flow graphs for legacy languages with semi-parsing. *In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022.

[6] **A. Kellens**, **K. D. Schutter**, **T. D'Hondt**, **L. Jorissen**, and **B. V. Passel**, Cognac: A framework for documenting and verifying the design of COBOL systems. *In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009.

[7] **F. Lei**, **J. Liu**, **S. Noei**, **Y. Zou**, **D. Truong**, and **W. Alexander** (2025). Enhancing cobol code explanations: A multi-agents approach using large language models. ArXiv preprint arXiv:2507.02182.

[8] **J. E. Sammet**, The early history of COBOL. *In History of Programming Languages*. 1978, 199–243.

[9] **S. Shah**, **S. Agarwal**, **S. Krishnan**, **V. Kanvar**, and **S. Chimalakonda**, A-cobrex: A tool for identifying business rules in COBOL programs. *In Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2025.

[10] **K. C. Swarna**, **N. S. Mathews**, **D. Vagavolu**, and **S. Chimalakonda** (2024). On the impact of multiple source code representations on software engineering tasks—an empirical study. *Journal of Systems and Software*, **210**, 111941.

[11] **D. Vagavolu**, **K. C. Swarna**, and **S. Chimalakonda**, A mocktail of source code representations. *In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.