A Spec-Driven Approach for Source Code Summarization of COBOL Legacy Systems with LLMs

João Pedro de Camargo Vaz
Dep. de Ciência da Computação, Universidade de Brasília
Brasília, Brazil
vaz.camargo@aluno.unb.br

Vinicius Assumpção de Araújo
Dep. de Ciência da Computação, Universidade de Brasília
Brasília, Brazil
vinicius.assumpcao@aluno.unb.br

Murilo Edson de Carvalho Souza
Dep. de Ciência da Computação, Universidade de Brasília
Brasília, Brazil
edson.souza@aluno.unb.br

Michael Rodrigues da Silva
Dep. de Ciência da Computação, Universidade de Brasília
Brasília, Brazil
michael.silva@aluno.unb.br

Rodrigo Bonifácio de Almeida
Centro de Informática, Universidade Federal de Pernambuco
Recife, Brazil
rbonifacio@cin.ufpe.br

Abstract

COBOL legacy systems still support critical business processes in transaction-intensive and highly regulated organizations, yet their modernization is often constrained by outdated documentation, dispersed system knowledge, and strong dependence on specialists. Although large language models (LLMs) have shown potential for software engineering tasks, using them as direct documentation generators is risky in legacy modernization contexts, where traceability, uncertainty control, and human validation are essential. This paper presents an initial research effort toward evidence-aware, LLM-supported documentation of COBOL legacy systems. We propose a spec-driven pipeline that decomposes legacy understanding into concern-bounded stages, from system characterization to documentation assembly and explicit gap registration. The pipeline incorporates rule-governed analysis and classifies findings as Verified, Inferred, Unknown, or Confirmation Required, aiming to separate code-supported facts from hypotheses and unresolved gaps. We report preliminary results from applying the pipeline to an open-source COBOL project, used as a controlled setting to assess end-to-end executability, cross-stage coherence, traceability, and reuse of intermediate artifacts. The results provide initial feasibility evidence that the workflow can structure LLM-supported reverse engineering before more demanding industrial validation.

Keywords

COBOL legacy systems, Large language models, Software documentation, Reverse engineering, Incremental modernization, Evidence classification

1 Introduction

Legacy systems remain central to organizations that require high reliability, availability, and traceability, such as banks, insurance companies, and public institutions. In many of these environments, COBOL applications running on mainframe platforms still support critical transactional processes, while also accumulating decades of technical debt, platform dependencies, implicit business rules, and operational knowledge that is difficult to recover. As a result, modernization is not merely a technical decision about replacing an old technology stack; it is a socio-technical decision about what, when, and how to modernize without compromising business continuity [2, 12, 16].

A persistent barrier to safe legacy modernization is the lack of qualified and up-to-date documentation. In long-lived systems, available documentation often diverges from the actual behavior of the software, while relevant knowledge remains tacit, concentrated in specialists, or scattered across source code, copybooks, data layouts, execution scripts, and operational conventions. This directly affects modernization planning: before deciding whether to encapsulate, reengineer, migrate, or replace a legacy component, organizations need sufficiently trustworthy evidence about what the system does, which dependencies it has, and which business capabilities it supports [10, 11].

Large language models (LLMs) have recently been applied to software engineering tasks including code migration, documentation generation, and legacy-system analysis [4, 7, 19]. For instance, Ziftci et al. [19] demonstrated large-scale code migration at Google, while Diggs et al. [7] evaluated LLM-generated documentation for MUMPS and assembly legacy code, reporting generally useful results in controlled settings. These results create an opportunity to reduce the cost of recovering system knowledge from source code. However, direct code-to-documentation generation can still produce explanations that are fluent but insufficiently connected to explicit code evidence—for example, without traceability to specific modules, control-flow structures, or data constructs—and may blur the distinction between code-supported facts and plausible interpretations. This is particularly problematic in COBOL systems, where business logic is often embedded in procedural flows, data movement, copybooks, and platform-specific integration mechanisms. Therefore, the challenge is not only to generate documentation, but to generate documentation that is structured, evidence-aware, and useful for modernization decisions.

This paper presents an initial research effort toward evidence-aware, LLM-supported documentation of COBOL legacy systems. Instead of treating documentation generation as a single prompt-to-document task, we propose a spec-driven pipeline that decomposes legacy understanding into concern-bounded stages—from system characterization through structure, data handling, integrations, and

decision flow to business capabilities, documentation assembly, and validation gaps. Each stage is governed by explicit rules and produces intermediate artifacts that can be reviewed, reused, and connected to source-code evidence. Uncertainty is treated explicitly: findings are classified according to repository support level, making uncertainty a first-class output and positioning LLMs as documentation assistants within a human-in-the-loop process. We report preliminary results from an open-source COBOL pilot designed to assess end-to-end executability, cross-stage artifact coherence, the use of the evidence model to expose uncertainty, and reuse of intermediate artifacts across stages.

The main contributions of this paper are:

- a spec-driven pipeline for decomposing COBOL legacy-system understanding into concern-bounded documentation stages;
- an evidence and uncertainty model that uses a four-class taxonomy to classify findings by the level of repository support, separating code-grounded facts from hypotheses and unresolved gaps;
- preliminary feasibility evidence from a controlled open-source COBOL pilot, together with a qualitative review by two practitioners regarding technical faithfulness, traceability, and usefulness;
- a research roadmap toward validation in a real financial-sector modernization scenario.

2 Background and Motivation

2.1 Legacy Modernization as a Decision Problem

In practice, legacy modernization is a decision-making process under uncertainty: organizations rarely replace legacy systems only because they are old, and modernization decisions involve trade-offs among operational risk, business continuity, maintenance cost, platform obsolescence, availability of expertise, and expected business value [2, 12, 16]. Previous work on legacy-system decision models emphasizes that alternatives such as maintenance, encapsulation, reengineering, migration, and replacement should be evaluated according to both technical and organizational factors [10, 11]. The quality of such decisions, however, depends on the ability to understand the existing system: with poor documentation, concentrated domain knowledge, or partially known behavior, choices are made with incomplete evidence. Recovering trustworthy knowledge about the legacy system is therefore a prerequisite for safe incremental modernization.

2.2 Documentation as Knowledge Infrastructure

In legacy systems, documentation acts as a knowledge infrastructure that supports maintenance, onboarding, auditing, impact analysis, and modernization planning; when it is outdated or incomplete, comprehension becomes dependent on manual code inspection and tacit knowledge. This problem is particularly challenging in COBOL ecosystems, where business logic is often distributed across procedural flows, data movement instructions, copybooks, file layouts, batch jobs, and platform-specific integration mechanisms. Useful documentation must therefore go beyond high-level textual summaries: it should identify structural elements, data dependencies, decision points, interaction surfaces, and business capabilities, while preserving traceability to observable evidence in the repository. In this sense, documentation for legacy modernization is closer to reverse engineering than to conventional technical writing [5, 6], which motivates organizing LLM-supported code comprehension as a structured reverse-engineering workflow rather than as a free-form text generation task.

Table 1: The adopted SDD lifecycle repurposed for reverse engineering of legacy systems.

Phase	Forward-engineering role	Reverse-engineering role in this work
spec-init	Frame a feature to be built	Frame one analytical concern of the legacy system
requirements	What the feature must do	Verifiable questions the concern must answer
design	How to implement the feature	Analytical strategy: evidence sources, discovery mode, uncertainty handling
tasks	Implementation work items	Analysis activities: collect, correlate, classify evidence
implementation	Code and tests	Documentation artifacts with evidence-classified findings

2.3 Spec-Driven Development

In this paper, we use Spec-Driven Development (SDD) to denote a structured workflow in which a feature or analytical concern is preceded by explicit specification artifacts that define requirements, design decisions, and tasks before implementation begins. In AI-assisted development, this organization replaces an unguided prompt-to-code interaction with a bounded and reviewable lifecycle. Our adopted lifecycle comprises five phases—*spec-init*, *requirements*, *design*, *tasks*, and *implementation*—each making the agent's intent, expected outputs, and intermediate decisions explicit. Recent work has begun to investigate specification-driven code generation through structured human-in-the-loop workflows, while its effects on output quality, efficiency, and developer interaction remain open empirical questions [15]. We repurpose this lifecycle for reverse engineering: each spec targets a specific analytical concern of a COBOL legacy system and produces structured documentation artifacts as its primary output. Table 1 contrasts, phase by phase, the adopted forward-engineering lifecycle with its responsibility in our reverse-engineering setting.

2.4 LLMs for Legacy Understanding

Large language models have created new opportunities for software engineering tasks involving code understanding, summarization, migration, and documentation [8, 18, 19]. Recent work has explored LLMs for mainframe and COBOL modernization [4], for generating documentation artifacts from legacy code in industrial settings [7], and for code migration across technology stacks [1, 13].

However, generated explanations may be plausible but unsupported by the source code, may omit relevant dependencies, or may blur the distinction between facts and hypotheses—limitations that are especially problematic when documentation is expected to support modernization decisions. Recent discussions on evaluating

LLMs in software engineering reinforce the need for task-specific criteria, realistic settings, and stronger attention to correctness, coverage, and validity [3, 9, 14]. For legacy documentation, evaluation should consider whether artifacts remain connected to code evidence, whether they expose uncertainty, and whether they support modernization reasoning—not only textual readability.

What is new. The individual ingredients of this work—staged analysis, evidence tagging, and rule-constrained agents—appear in prior work; the contribution is their combination and repurposing for COBOL reverse engineering: (i) the adopted SDD lifecycle is inverted so that each spec documents an existing concern instead of specifying a new feature (Table 1); (ii) uncertainty becomes a first-class pipeline output through a four-class evidence taxonomy; and (iii) agent behavior is constrained by an explicit, versioned rule layer. The approach differs from Diggs et al. [7] in three dimensions: the unit of documentation (concern-bounded system-level specs and reusable artifacts vs. line-wise comments associated with specific code locations), the workflow (a staged reverse-engineering lifecycle with artifact reuse and human checkpoints vs. chunk-based comment generation using placeholders and structured prompts), and the output contract (evidence-classified findings with traceable sources and explicit unresolved gaps vs. line-wise comments returned in a JSON-structured format).

3 The Proposed Spec-Driven Approach

The approach was designed from the premise that legacy understanding should not be produced as a single free-form summary, but as a sequence of bounded analytical steps, each focused on a specific concern and governed by explicit rules—transforming LLM-supported code comprehension into a controlled reverse-engineering workflow in which intermediate findings are traceable, reusable, and explicit about uncertainty.

Applying LLMs directly to COBOL documentation without such structure faces three recurring challenges: (i) *grounding failure*, where plausible explanations lack code evidence, blurring verified facts and interpretations; (ii) *artifact absence*, where single-prompt workflows lack reusable intermediate outputs, preventing structural findings from informing higher-level interpretations; and (iii) *scope drift*, where unconstrained agents produce general summaries rather than isolating specific concerns. The spec-driven approach is designed to address these challenges through an explicit evidence taxonomy, a concern-bounded lifecycle with artifact reuse, and the one-spec-per-concern principle, respectively.

The approach is composed of three elements: (i) a COBOL pilot repository used as a controlled target system; (ii) a set of SDD commands and artifacts that define and chain the concern-bounded specs; and (iii) a rule layer that constrains how the LLM-based agent performs discovery, design, task generation, and documentation assembly.

3.1 Concern-Bounded Understanding

The approach decomposes legacy understanding into nine sequenced concerns (Figure 1), each represented as a separate *spec* responsible for one main question. The concerns progress from structural layers—system characterization, interaction surfaces, application structure, state and data handling, and external/internal integrations—to interpretive stages: processing and decision flow, business capabilities, documentation assembly, and validation and gaps.

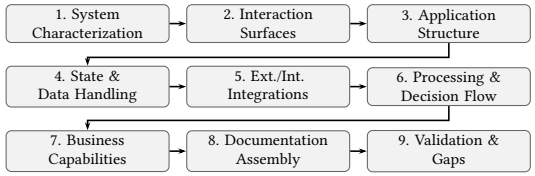

Figure 1: The nine concern-bounded stages of the spec-driven pipeline (arrows: evidence-accumulation flow).

To illustrate, the *system-characterization* spec defines requirements such as *REQ-SC-01: identify the execution model from available COBOL artifacts* and *REQ-SC-02: enumerate primary program units and entry points*. Its design phase specifies the evidence sources to inspect (COBOL source files, Makefile, README), the discovery mode (light or full), and known uncertainty points; its implementation produces a structured artifact listing the module inventory, execution-model classification, and evidence class per finding—consumed as inputs by the subsequent *application-structure* and *processing-and-decision-flow* specs.

This ordering is intentional: it prevents the LLM from jumping to business-level explanations before sufficient structural evidence has been collected—a business capability should not be inferred from a file name or a fluent interpretation of a paragraph, but grounded in previous outputs such as module inventories, data-handling summaries, integration maps, and decision-flow descriptions. The design also reduces context explosion: each spec remains small enough to be reviewed independently, while its outputs become inputs for later stages.

3.2 Spec Lifecycle

Each concern follows the SDD lifecycle with the phase responsibilities summarized in Table 1. The lifecycle is deliberately repetitive: instead of one large specification for the entire system, a chain of smaller specs keeps each step inspectable, prevents later stages from silently absorbing unresolved assumptions, and gives human reviewers well-defined checkpoints before findings are promoted into consolidated documentation.

A key distinction in the lifecycle is the separation between exploratory research and design: the *research* artifact holds intermediate investigation, conflicting signals, candidate interpretations, and evidence notes, while the *design* artifact records the scoped analytical plan—concern boundaries, inputs, outputs, dependencies, evidence sources, and uncertainty-handling decisions. This prevents raw exploration from being mistaken for validated documentation and keeps the design layer reviewable.

Transitions between stages are human-gated rather than autonomous: a spec's artifacts are promoted to inputs of downstream specs only after passing the design-review checkpoint (Section 3.3) and a findings review at the end of implementation. When the agent misclassifies a finding—e.g., labeling as *Verified* a claim that is only indirectly supported—the reviewer demotes it at the checkpoint,

and the corrected artifact is what subsequent specs consume, preventing the propagation of inflated evidence. Conflicts between a new finding and a previously accepted artifact are recorded in the research artifact and must be resolved—by revising the earlier artifact or reclassifying the finding—before documentation assembly.

3.3 Rule-Governed Agent Behavior

The approach relies on a rule layer to adapt a generic spec-driven development workflow to legacy-system documentation. These rules act as behavioral constraints for the LLM-based agent: rather than formatting preferences, they enforce methodological properties that address the three challenges described at the start of this section.

The solution is organized in three architectural layers. The *spec layer* stores and chains concern-bounded specification artifacts, one per concern. The *rule layer* constrains agent behavior through explicit governance instructions loaded at the start of each analytical session. The *artifact layer* accumulates intermediate and consolidated documentation outputs consumed by subsequent specs, thereby supporting artifact reuse across the workflow.

Operationally, the governance layer is implemented as plain-text instruction files versioned with the repository and loaded into each analytical session. The implementation comprises nine rule files, with 2,036 lines of rule definitions. Rather than loading the entire corpus for every interaction, each lifecycle command selects only the subset relevant to its phase, reducing the fixed instruction footprint and preserving context capacity for repository evidence and artifacts produced by previous stages. The current implementation uses a single-agent setup; multi-agent orchestration and parser-based augmentation are intentionally postponed until the base workflow receives broader validation (Section 5).

The first group of rules defines global steering principles—repository-first reasoning, discovery before interpretation, one concern per spec, evidence before inference, and avoidance of repository-specific shortcuts—which prevent the agent from behaving as a generic text generator and encourage it to reason as a reverse-engineering assistant. A second group governs design production: each design must describe an analytical strategy rather than implementation details, name evidence sources, define expected outputs, expose dependencies, and include uncertainty handling as part of the design itself.

The approach also distinguishes between light and full discovery. Light discovery frames a concern through a limited repository scan, supporting early classification of findings and the decision of whether deeper inspection is necessary; full discovery is used when the concern requires correlation across multiple artifacts, preservation of ambiguity, or deeper reasoning about structural relationships. This allows the workflow to deepen the analysis without expanding the concern beyond its boundary. Two concrete rule examples illustrate how the challenges are mitigated at the design level: (i) *"do not produce business-capability claims before structural concerns are documented; each such claim must cite at least one finding from a prior spec"* (addressing scope drift); and (ii) *"a design artifact is invalid if it omits evidence sources, expected outputs, or an uncertainty-handling strategy"* (addressing grounding failure).

Before downstream work continues, a design review rule classifies the design as *GO*, *GO WITH GAPS*, or *NO-GO*. This checkpoint prevents weak specs from advancing without explicit acknowledgment of their limitations: a design may proceed with gaps when the missing information is known and documented, but not as if unresolved uncertainty did not exist. Additional rules govern gap analysis, task generation, and safe parallelization; tasks must produce concrete outputs, remain traceable to requirements, and progress from evidence collection to correlation, classification, and artifact production.

3.4 Evidence and Uncertainty Model

The approach uses a four-class evidence model that aims to mitigate hallucination risk and make uncertainty visible; measuring this mitigation effect is future work (Section 5). Classification is performed by the LLM agent during the *implementation* phase of each spec, guided by explicit rules that define the criteria for each class; human reviewers may challenge or revise classifications at any lifecycle checkpoint. A finding is *Verified* when directly supported by the legacy repository (source code, configuration files, scripts, or artifacts generated by previous specs); *Inferred* when it is a reasonable interpretation indirectly supported by available evidence, but not explicitly stated; *Unknown* when there is not enough evidence to answer the question and the path to resolve it is unclear; and *Confirmation Required* when the answer probably exists outside the repository and requires validation from domain experts, operators, internal systems, or non-versioned organizational artifacts.

This distinction is central to the approach. In legacy systems, not every relevant question can be answered from source code alone: some aspects depend on operational conventions, production schedules, external systems, institutional ownership, or tacit business knowledge, and treating them as model-inferable would increase the risk of plausible but unsupported documentation. Explicitly marking them as *Confirmation Required* preserves the boundary between repository evidence and organizational knowledge. The four classes cover the principal epistemic states relevant to source-code-based documentation—directly observable facts, reasonable interpretations, unresolvable absences, and externally-held knowledge—and are treated as preliminary, subject to revision in broader empirical studies.

The evidence model also supports human-centered validation: rather than inspecting only a final document, reviewers can challenge uncertainty when requirements are defined, design decisions are made, tasks are generated, findings are classified, and unresolved gaps are assembled—suiting incremental modernization contexts, where specialists need to know not only what the documentation says, but how strongly each statement is supported.

4 Preliminary Evaluation

The preliminary evaluation was designed as a feasibility assessment, not as a full industrial validation: its goal was to observe whether the spec-driven workflow could be executed end-to-end, whether the generated artifacts remained aligned with their intended concerns, and whether the evidence model helped distinguish supported findings from unresolved questions. We selected an open-source COBOL accounting system [17] as the pilot target: an

existing, independently maintained project implementing deposit, withdrawal, and balance inquiry as terminal-driven operations across three modules (`main.cob`, `operations.cob`, `data.cob`) and approximately 200 lines of source code. The project was deliberately chosen for its small, self-contained scope: its size makes it possible to execute all nine stages in a controlled timeframe, verify each generated artifact against the complete source code, and obtain focused specialist feedback covering all stages—conditions impractical for a large enterprise codebase at this exploratory phase. It exercises core COBOL constructs—CALL/PERFORM, fixed-point arithmetic, terminal I/O, and conditional guards—without mainframe-specific artifacts, making it a suitable workflow-feasibility target.

4.1 Generated Documentation Artifacts

The pipeline produced intermediate and consolidated documentation artifacts across all nine stages; Table 2 presents representative findings from six selected stages together with their original evidence classes.

These selected examples illustrate that evidence classes depend on the support available for each finding rather than on the abstraction level of its stage. Structural and business-level findings can remain *Verified* when they are directly traceable to repository evidence and verified upstream artifacts. *Inferred* is used for conclusions supported indirectly by repository evidence, whereas *Unknown* records questions that static repository inspection cannot resolve. No finding selected from the generated pilot artifacts was classified as *Confirmation Required*.

To make the evidence flow concrete, consider the account-debit capability. Stage 1 identified the operation vocabulary and the three-module COBOL structure; Stage 3 documented the navigation-to-operations and operations-to-data boundaries; Stage 4 established that `STORAGE-BALANCE` is the authoritative session-scoped state; and Stage 6 traced both debit branches, including the `FINAL-BALANCE >= AMOUNT` guard, the conditional data-layer write, and the insufficient-funds outcome. Stage 7 then consolidated these findings into the Account Debit capability: an amount is subtracted only when sufficient funds are available, while a failed guard leaves the balance unchanged. The capability remained classified as *Verified* because its constituent claims were directly traceable to source code and previously verified artifacts. A complementary case illustrates unresolved uncertainty: the Interaction Surfaces stage could identify the COBOL `PIC 9(6)V99` display field, but not its exact runtime rendering for values such as 1000.00. That finding was classified as *Unknown* and deferred to live execution rather than being replaced by a plausible formatting assumption.

4.2 Observed Feasibility Results

The first observed result is that the approach was executable from early characterization to documentation assembly, with each spec producing outputs reused by subsequent stages: characterization and structure outputs supported the later documentation of processing flow and business capabilities—reducing the risk of isolated summaries that fail to cohere as a documentation set.

The second result is that the concern boundaries helped control scope expansion: the architecture stage focused on modules, call relationships, and boundaries; interaction surfaces on terminal inputs and outputs; state and data handling on balance storage, lifetime, and propagation; and processing and decision flow on operation paths and guards. This separation made the documentation easier to inspect because each artifact had a clear analytical responsibility.

The third result is that the evidence and uncertainty model was useful for documenting limitations. Rather than forcing definitive answers for every question, the workflow allowed the agent to mark unresolved issues explicitly—the lack of COBOL automated tests, the need to confirm exact runtime formatting, and the behavior of arithmetic overflow without a specific handler—showing that the approach can produce not only descriptions of what is known, but also a structured inventory of what still needs validation.

Evidence-class use and label review. The pilot was not instrumented to record aggregate per-class counts or execution metrics such as token usage, stage duration, and human-intervention frequency; these measures are included in the next evaluation stage (Section 5). *Verified* findings appeared across both structural and higher-level stages when claims could be traced directly to source code or verified upstream artifacts. *Inferred* was used for conclusions supported indirectly by repository evidence, whereas *Unknown* captured questions that static repository inspection could not resolve. All agent-produced classifications were manually reviewed by the authors against the corresponding source code and generated artifacts, which was feasible given the pilot's size.

4.3 Specialist Validation

The generated documentation was evaluated by two specialists with professional experience in COBOL and legacy-system maintenance, working in enterprise environments where COBOL systems are actively maintained in production.

The review followed a minimal protocol. Each specialist independently inspected the consolidated documentation, the intermediate stage artifacts, and the pilot source code, assessing three criteria: *technical faithfulness* (whether statements correctly describe the code), *traceability* (whether findings can be followed back to code evidence and prior artifacts), and *usefulness* for comprehension and modernization reasoning. Observations were then consolidated in a joint discussion with the authors, resolving divergences by re-inspecting the corresponding code and artifacts; no unresolved disagreements remained. The evaluation was qualitative, without numeric scoring.

Both specialists agreed with the overall approach and approved the specification rules governing agent behavior, the modularization of documentation functions, and the structure of the generated artifacts. The stages receiving the most positive feedback were *interaction surfaces*, *processing and decision flow*, and *business capabilities*, considered particularly effective in eliciting business rules that tend to remain implicit in legacy systems, distributed across procedural flows and data movement instructions without being explicitly documented.

Two limitations were identified: the pilot repository is small relative to a real enterprise context, limiting generalizability despite confirming applicability; and the artifacts exhibited some repetition across concerns—not characterized as redundancy, since each occurrence appeared in a distinct analytical context, but acknowledged as worth refining as the pipeline matures.

Table 2: Representative findings from selected pipeline stages and their original evidence classes in the pilot evaluation (V: Verified; I: Inferred; U: Unknown).

Pipeline Stage	Cl.	Representative Finding
System Characterization	V	The COBOL implementation comprises three modules: `main.cob` for navigation, `operations.cob` for operation logic, and `data.cob` for balance storage; its state is in-memory and session-scoped.
Interaction Surfaces	U	The exact runtime rendering of COBOL `PIC 9(6)V99` values, including whether values such as 1000.00 contain leading zeros or spaces, cannot be established without live execution.
Application Structure	I	The compiled COBOL executable is expected to be named `accountsystem`, inferred from the corresponding `.gitignore` entry because no observed build output is available.
State and Data Handling	V	`STORAGE-BALANCE` in `data.cob` is the authoritative balance variable, initialized to 1000.00 and represented as fixed-point `PIC 9(6)V99`; no persistent store is used.
Processing and Decision Flow	V	The debit guard is the sole condition that can suppress a data-layer write; an amount equal to the current balance passes the guard and produces a balance of 0.00.
Business Capabilities	V	The system exposes four user-facing capabilities: Balance Inquiry, Account Credit, Account Debit, and Session Management.

4.4 Limitations and Threats to Validity

The current evaluation has important limitations. The pilot system does not include typical mainframe artifacts such as JCL scripts, VSAM files, DB2 schemas, CICS transactions, or production-scheduling metadata, and the results cannot be generalized to large enterprise COBOL systems. The evaluation of a single, small-scale project was a deliberate choice for this stage: it enabled complete inspection of the source code and generated artifacts, as well as a controlled assessment of workflow execution and cross-stage coherence, before application to more representative targets.

Two further threats stand out. First, the evaluation includes no baseline comparison—such as single-prompt LLM summarization or parser-assisted reverse engineering—so we cannot yet quantify the benefit of the spec-driven structure over simpler alternatives; establishing such baselines is an explicit roadmap item (Section 5). Second, the pilot does not exercise the factors that most motivate the approach in industry—business logic dispersed across many programs, cross-system integrations, operational conventions, and tacit specialist knowledge—which are the target of the industrial validation stage. The preliminary results should therefore be interpreted as methodological feasibility evidence rather than as a demonstration of industrial effectiveness or scalability.

5 Discussion and Research Roadmap

The preliminary evaluation indicates that the proposed approach can organize LLM-supported documentation as a controlled reverse-engineering process. The rule layer and evidence taxonomy are designed to reduce grounding failures, the concern-bounded lifecycle and artifact chaining are intended to preserve reusable intermediate outputs, and the one-spec-per-concern principle is intended to limit scope drift. The controlled pilot provides feasibility evidence for executing these mechanisms coherently, but their comparative effectiveness remains to be evaluated through broader studies and explicit baselines.

The approach positions LLMs as governed documentation assistants rather than autonomous generators: the workflow directs the model to inspect repository evidence, classify findings by evidence class, reuse reviewed artifacts, and expose unresolved questions at lifecycle checkpoints. In practice, the resulting artifacts can support modernization planning by helping teams prioritize modules for migration, direct specialist attention to unresolved or externally held knowledge, identify documentation gaps that block reengineering decisions, and scope migration increments around documented business capabilities.

The research roadmap proceeds in two directions: (i) industrial validation on a financial-sector COBOL system, instrumented to collect execution metrics and compare against single-prompt summarization and parser-assisted reverse engineering; and (ii) specialized agents and parser-based structural extraction for larger codebases.

6 Conclusion

This paper presented a spec-driven pipeline for evidence-aware documentation of COBOL legacy systems with LLM support. The approach decomposes legacy understanding into nine concern-bounded stages and governs agent behavior through explicit rules and a four-class evidence taxonomy (*Verified*, *Inferred*, *Unknown*, and *Confirmation Required*), positioning LLMs as governed documentation assistants rather than autonomous generators of unverified summaries.

The controlled pilot provided initial evidence that the pipeline can be executed end-to-end, that intermediate artifacts can be reused across stages, and that the evidence model can preserve distinctions between directly supported findings, indirect interpretations, and unresolved questions. The qualitative review by two COBOL practitioners provided positive feedback regarding the technical faithfulness, traceability, and usefulness of the generated documentation, particularly for stages intended to surface business rules embedded in procedural flows and data movement. However, the small pilot and the absence of industrial baselines mean that the results constitute methodological feasibility evidence rather than comparative or industrial effectiveness.

Artifact Availability

The complete research artifact, including the SDD commands, rule layer, specifications, generated documentation, and replication guide, is available at https://github.com/ppgi-2025-cobol/preliminary-work.

References

[1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. *arXiv preprint arXiv:2408.16151* (2024).

[2] Keith H. Bennett, Magnus Ramage, and Malcolm Munro. 1999. Decision model for legacy systems. *IEE Proc. Softw.* 146, 3 (1999), 153–159.

[3] Keyuan Cheng, Xudong Shen, Yihao Yang, Tengyue Wang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. 2025. CODEMENV: Benchmarking Large Language Models on Code Migration. *arXiv preprint arXiv:2506.00894* (2025).

[4] Anh T. V. Dau, Hieu Trung Dao, Anh Tuan Nguyen, Hieu Trung Tran, Phong X. Nguyen, and Nghi D. Q. Bui. 2024. XMainframe: A Large Language Model for Mainframe Modernization. *arXiv preprint arXiv:2408.04660* (2024).

[5] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. 2005. A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd International Conference on Design of Communication (SIGDOC '05)*. ACM, 68–75. doi:10.1145/1085313.1085331

[6] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. 2002. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann.

[7] Colin Diggs, Michael Doyle, Amit Madan, Siggy Scott, Emily Escamilla, et al. 2024. Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation. *arXiv preprint arXiv:2411.14971* (2024).

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Xing Hu, Feifei Niu, Junkai Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. 2025. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. arXiv:2505.08903 [cs.SE]

[10] Irja Kankaanpää, Päivi Tiihonen, Jarmo J. Ahonen, Jussi Koskinen, Tero Tilus, and Henna Sivula. 2007. Legacy system evolution - a comparative study of modernisation and replacement initiation factors. In *Proceedings of the Ninth International Conference on Enterprise Information Systems (ICEIS 2007)*. 280–287.

[11] Jussi Koskinen, Jarmo J. Ahonen, Henna Sivula, Tero Tilus, Heikki Lintinen, and Irja Kankaanpää. 2005. Software Modernization Decision Criteria: An Empirical Study. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE Computer Society, 324–331.

[12] Andrea De Lucia, Anna Rita Fasolino, and Eugenio Pompella. 2001. A Decisional Framework for Legacy System Management. In *2001 International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society, 642.

[13] Ishaani M., Behrooz Omidvar-Tehrani, and Anmol Anubhai. 2024. Evaluating Human-AI Partnership for LLM-based Code Migration. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24)*. Association for Computing Machinery.

[14] Victor May, Diganta Misra, Yanqi Luo, Anjali Sridhar, Justine Gehring, and Silvio Soares Ribeiro Junior. 2025. FreshBrew: A Benchmark for Evaluating AI Agents on Java Code Migration. arXiv:2510.04852 [cs.SE]

[15] Giovanni Rosa, David Moreno-Lumbreras, Gregorio Robles, and Jesús M. González-Barahona. 2026. Understanding Specification-Driven Code Generation with LLMs: An Empirical Study Design. In *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Registered Report, arXiv:2601.03878.

[16] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. 2003. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.* Addison-Wesley Professional.

[17] Tejveer Singh. 2024. cobol-accounting-system. https://github.com/tjsingh85/cobol-accounting-system.

[18] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2024. A Survey on Large Language Models for Software Engineering. *Science China Information Sciences* (2024).

[19] Celal Ziftci, Stoyan Nikolov, Anna Sjövall, Bo Kim, Daniele Codecasa, and Max Kim. 2025. Migrating Code at Scale With LLMs At Google. In *Foundations of Software Engineering (FSE)*.