Beyond Code Generation: A Spec-Driven Experience Report on LLM-Assisted Modernization of a Legacy Delphi System

Wellynton Diniz
Department of Computer Engineering
Faculdade de Tecnologia de Sinop (FASTECH)
Sinop - MT, Brazil
wdiniz@enc.fastech.edu.br

Abstract

Legacy software modernization remains a major challenge for organizations that rely on obsolete technologies, limited documentation, and business rules embedded in source code. Although Large Language Models (LLMs) have significantly expanded the possibilities for code generation, there is still limited evidence of their use in the modernization of real-world legacy systems. This paper presents an ongoing experience report on the migration of a pet shop and veterinary clinic management system from Delphi/Firebird to a web architecture based on Java/Spring Boot, Angular, and MySQL, supported by the Codex coding agent. Unlike approaches based on direct code translation, the adopted process follows a spec-driven strategy in which specifications are reconstructed from the legacy system, reviewed by developers, and used as the primary artifact for AI-assisted implementation. So far, the approach has been applied to 41 screens and functional modules, as well as two production database migrations. The paper describes the adopted workflow, discusses key architectural decisions, and presents lessons learned.

Keywords

Legacy System Modernization, Large Language Models, Spec-Driven Development, Software Migration, Experience Report

1 Introduction

Legacy system maintenance and evolution have historically accounted for a substantial portion of the total cost of software development, and the modernization of such systems has often been perceived as a costly and risky endeavor, particularly when up-to-date documentation is unavailable and domain knowledge is embedded primarily in the source code [9]. A recent survey of professionals from the Brazilian software industry reinforces this perspective, indicating that legacy systems are predominantly characterized as applications built on obsolete technologies that remain critical to organizations. The main challenges reported include difficulties in understanding the source code, the scarcity of specialized professionals, and the high expected cost of future maintenance [2]. In this context, the literature recommends incremental modernization strategies based on recovering requirements from the existing system and migrating functionality in stages, rather than performing complete rewrites [8]. Systems developed in Delphi with Firebird represent a common example of this scenario, particularly in small and medium-sized enterprises, where the lack of formal documentation and the strong coupling between the user interface, business logic, and data access make their evolution especially challenging.

One direction that has been gaining increasing attention in this context is *Spec-Driven Development* (SDD), which advocates treating explicit specifications as the primary artifact of the software development process, guiding implementation rather than being produced merely as post hoc documentation [12]. This perspective is particularly well suited to the modernization of legacy systems, in which business rules are often implicitly embedded in the source code and must first be reconstructed before they can be preserved throughout the migration process [4].

In parallel, LLM-based coding agents have increasingly been used to support software development and maintenance by analyzing entire repositories, proposing architectural changes, and automatically generating source code. Part of the appeal of these agents relates to the shortage of qualified professionals in the software industry: a recent systematic mapping identified the misalignment between academic education and industry demands, and the rapid evolution of market technologies, as key factors behind this *skill gap*, particularly evident in legacy technologies such as Delphi [5]. Despite these advances, the literature has predominantly focused on narrowly scoped tasks, such as library migration or source-to-source translation, typically evaluated through automated correctness metrics [1, 3]. These studies demonstrate substantial productivity gains but also show that LLMs may introduce subtle defects during code translation, even when the source code is syntactically correct [10]. Reports on using these agents for incremental modernization of real-world systems, simultaneously involving user interfaces, implicit business rules, and data models, remain scarce.

In this context, this paper presents an ongoing experience report on the application of a spec-driven modernization approach supported by an LLM-based coding agent for the migration of a real-world management system for pet shops and veterinary clinics from Delphi 10.3 Rio with a Firebird 2.5 database to a web-based architecture using Java/Spring Boot, Angular, and MySQL. Rather than employing large language models solely for code translation, the proposed approach reconstructs a specification from the legacy system through a structured evidence elicitation process, which is then subjected to human review before any artifacts of the target architecture are generated. To date, this approach has been applied to the migration of 41 screens and functional modules, covering access control, core master data, clinical workflows, scheduling, catalog management, inventory, purchasing, sales, tax documents, and financial management, as well as two production data imports from the Firebird database.

The main contributions of this work are: (i) the description of an incremental spec-driven modernization workflow supported by LLM-based agents, in which a reconstructed specification of the

legacy system serves as the central artifact for AI-assisted generation; (ii) the application of this workflow to the migration of a real-world legacy system, encompassing 41 screens and functional modules; (iii) an analysis of the architectural decisions adopted during the modernization process, including cases in which the AI agent proposed project restructurings beyond simple code generation, with all suggestions undergoing human review; and (iv) the consolidation of lessons learned, organized into five thematic categories, together with threats to validity that may assist teams undertaking similar modernization efforts.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the legacy system and the target architecture. Section 4 introduces the spec-driven workflow adopted for AI-assisted modernization. Section 5 reports the experience gained from the migration of the modules completed thus far. Section 6 discusses the lessons learned, while Section 7 presents the threats to validity. Section 8 outlines the planned next steps for the continuation of the migration, and Section 9 concludes the paper.

2 Related Works

Legacy system modernization has been extensively investigated from both technical and organizational perspectives. Khadka et al. analyzed, through interviews with industry practitioners, how legacy systems are perceived and which factors influence their modernization, highlighting that migration decisions are primarily driven by business risks and operational continuity rather than purely technical concerns [9]. Jain and Chana proposed a generalized roadmap for legacy system modernization based on understanding the existing system, planning, and incremental migration [8]. These studies provide the foundation for the strategy adopted in this experience report, in which the system is modernized incrementally by functional modules while the Delphi application remains in production throughout the transition process.

In recent years, *Large Language Models* (LLMs) have increasingly been employed to support software maintenance and modernization activities. Almeida et al. demonstrated that LLMs can assist in the automated migration of software libraries, although the results are highly sensitive to prompting strategies [1]. Islam et al. conducted an empirical study on Python library migration using LLMs, reinforcing the need for human validation of the generated artifacts [7]. Cheng et al. proposed the CODEMENV benchmark to evaluate LLMs on code migration tasks across different versions of Python and Java libraries, reporting success rates below 45% even for the best-performing models [3]. Pan et al. investigated defects introduced by LLMs during automated code translation and concluded that semantic errors may remain undetected even when the generated code compiles successfully [10].

Recent studies have also investigated the use of LLMs in general software development activities. Fernandes et al. compared the readability of code generated by different models based on recommendations from static analysis tools, observing that the models still produce code with readability limitations [6]. Silva et al. analyzed 155 interactions between developers and ChatGPT in *pull requests*, showing that code generation tasks require more iterations than text revision or technical clarification [14]. Pimenta et al., in turn, compared the effort required for library migration in systems with and without clean architecture, proposing a quantitative methodology based on implementation time and lines of code modified, which is adopted in this work as a reference for future evaluations [11].

Despite these advances, most of the existing literature focuses on narrowly scoped tasks, such as library migration, source-to-source translation, or the generation of small code fragments. Few studies describe the end-to-end modernization of production systems involving the simultaneous migration of user interfaces, business rules, and data models. In contrast, this experience report investigates how LLM-based agents can support a complete incremental modernization process, helping preserve business rules while assisting architectural decision-making throughout the migration.

The approach adopted in this work is grounded in *Spec-Driven Development* (SDD), a paradigm in which explicit specifications are treated as the primary artifact of the development process, guiding implementation rather than merely documenting it afterward [12]. Piskala describes different levels of SDD adoption and highlights legacy system modernization as a particularly suitable application scenario, since reconstructing business rules before reimplementation helps reduce migration risks [12]. Complementarily, Macedo and Costa introduce the concept of reverse documentation engineering, in which legacy systems are transformed into operational specifications for AI agents while preserving confidence levels and identifying gaps that require human validation [4].

The workflow described in this paper aligns with this perspective by reconstructing an intermediate specification from the Delphi source code before generating any artifacts of the target architecture. The evidence elicitation documents described in Section 4 play precisely this role: transforming implicit business rules into developer-reviewed specifications, which then become the primary artifact for AI-assisted implementation. Figure 1 summarizes this workflow, referred to in this work as the *Spec-Driven AI-Assisted Modernization Workflow* (SDAW).

3 Legacy System and Target Architecture

The case study presented in this work is a production management system for pet shops and veterinary clinics, developed in Delphi 10.3 Rio with FireDAC and Firebird 2.5, which evolved over several years without formal documentation beyond its source code, totaling approximately 70 screens. Its functionality is distributed across core modules including master data management, clinical workflows, scheduling, inventory, purchasing, sales, financial management, and reporting. These characteristics make it representative of the type of legacy system discussed in Section 2: a system with strong coupling between the user interface, business logic, and persistence layer, where functional knowledge is predominantly embedded in the source code. This scenario motivated the adoption of the SDAW (Section 4) to reduce the risk of unintended behavioral changes during the modernization process.

The migration began in June 2026 with the decision to preserve the legacy system in its entirety while the new application evolves in parallel, enabling an incremental, module-by-module migration without interrupting production. This strategy is consistent with the recommendations of Khadka et al. [9] and Jain and Chana [8].

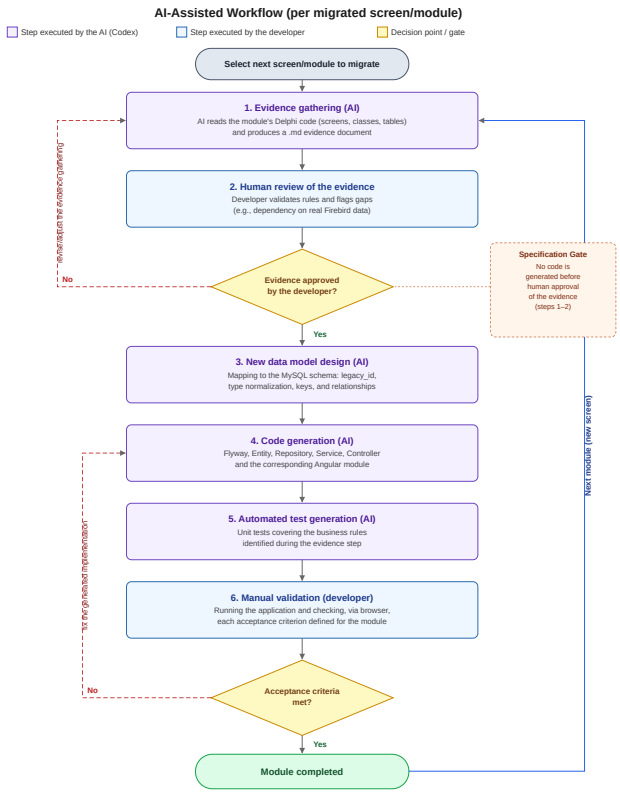

Figure 1: Spec-Driven AI-Assisted Modernization Workflow (SDAW), highlighting the decision points (*gates*) and the rework cycles adopted during the incremental migration.

The target architecture consists of a Java 17/Spring Boot REST API, JPA-based persistence with Flyway, an Angular application using *standalone* components, and MySQL 8.4 deployed in Docker containers. The replacement of Firebird with MySQL was motivated by the broader availability of professionals familiar with the ecosystem and the potential for reduced operational costs. Firebird remains in use as a read-only data source for extracting and reconciling legacy data.

Mapping between the architectures. Figure 2 illustrates not only the replacement of technologies but also the reorganization of responsibilities across system components. In the legacy system, each functionality is implemented through a pair consisting of a form (*.dfm*) and its corresponding source code (*.pas*): the form defines the visual components, while the source code concentrates event handling, validations, and a substantial portion of the business logic. In the modernized system, this pair is replaced by Angular components that communicate exclusively with the REST API through HTTP/JSON, eliminating direct database access from the desktop client. Similarly, the Delphi *DataModule*, which centralizes data access and part of the business logic, is replaced by an explicit separation into *Repository*, *Entity*, *Service*, and *Controller* layers. More than a technological replacement, Figure 2 represents the evolution from a two-tier client-server architecture to a three-tier web architecture, with versionable and independently testable REST contracts, providing the necessary foundation for implementing the business rules reconstructed in Section 4 independently of the original technology.

4 Spec-Driven AI-Assisted Modernization Workflow

The system modernization was conducted using the SDAW, which combines reverse engineering, reconstructed specifications, and artifact generation assisted by LLM-based agents. Rather than using the language model solely as a code translation tool, the workflow requires every implementation to be preceded by the reconstruction and human validation of a specification derived from the legacy system. Codex, OpenAI's coding agent, was used throughout all artifact generation activities. Different model variants were employed pragmatically according to quota availability rather than through systematic evaluation, a limitation discussed in Section 6.

The SDAW organizes the modernization of each module into six sequential stages:

- **Evidence elicitation:** the agent analyzes the Delphi source code of the target module (screens, classes, and referenced database tables) and produces a Markdown document containing the files involved, the identified data structures, and the business rules inferred from the source code.
- **Specification review and approval:** the developer reviews the generated document, validates the identified business rules, and explicitly records any gaps or uncertainties, particularly those that depend on the analysis of real data stored in the Firebird database. The workflow proceeds to implementation only after this specification has been approved.
- **Target data model design:** based on the approved specification, the AI proposes the corresponding relational schema in MySQL, including normalization strategies, the definition of the `legacy_id` field for traceability, and architectural decisions related to persistence.
- **Code generation:** the AI implements the required artifacts of the target architecture, including Flyway migration scripts, JPA entities, repositories, services, REST controllers, and the corresponding Angular components.
- **Automated test generation:** unit tests are generated to verify the business rules identified during the evidence elicitation stage, covering scenarios such as mandatory field validation, data normalization, and duplicate detection.
- **Manual validation:** the developer executes the application, verifies the predefined acceptance criteria, and validates the functional behavior of the module before considering it complete.

The main distinguishing feature of the workflow is that it prevents any code from being generated before the reconstructed specification has been reviewed and approved. This strategy aims to reduce the risk identified by Pan et al. [10] that LLMs may preserve code compilability while introducing semantic changes to the system's behavior. From the perspective of *Spec-Driven Development* proposed by Piskala [12], the workflow closely aligns with a *spec-first* approach. More specifically, it follows the concept

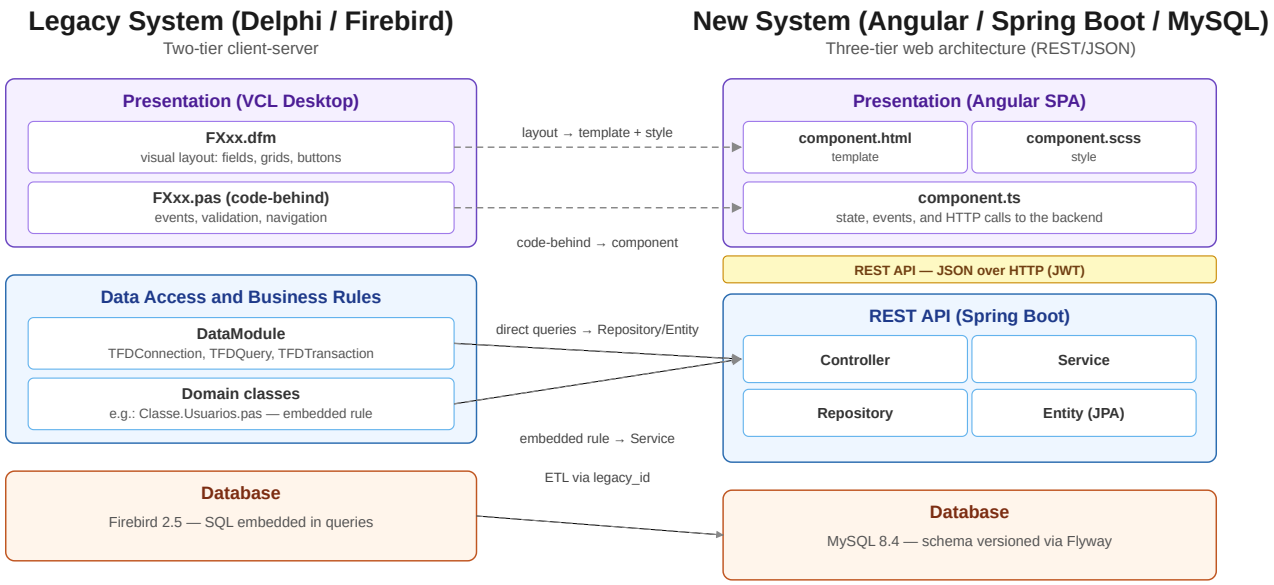

Figure 2: Mapping between legacy architecture (Delphi/Firebird) and target architecture (Angular/Spring Boot/MySQL), highlighting the redistribution of responsibilities among interface, business rules, and persistence during the modernization.

of reverse documentation engineering introduced by Macedo and Costa [4], transforming business rules implicitly embedded in the Delphi source code into reviewed operational specifications before generating the new application.

Figure 1 presents the SDAW and highlights two decision gates. The *Specification Gate* ensures that no implementation begins before the specification has been reviewed and approved; if inconsistencies are identified, the workflow returns to the evidence elicitation stage. The *Validation Gate*, performed after manual validation, directs rework back to the code generation stage whenever an acceptance criterion is not met, while preserving the previously approved specification. To date, the first rework cycle has not been required for any of the completed modules, whereas the second occurred only for minor user experience adjustments (Section 6). This absence of rework should be interpreted with caution: it may indicate that the evidence elicitation process already produces reliable specifications, but it may also reflect approval criteria that were not sufficiently rigorous, a distinction that this experience report cannot resolve.

Characterization of the migrated modules. To support planning and enable retrospective analysis, we adopted a two-dimensional classification based on functional type and complexity level, which was applied to the 41 completed modules (Section 6). The functional type comprises six categories: **Master Data** (CRUD operations on a single entity); **Query** (read-only operations with filtering and pagination); **Transactional** (business transactions that modify application state, involving multiple entities, calculations, and side effects); **Report** (aggregation and presentation of information through summarized queries); **Extension/Association** (extensions of previously migrated modules, such as integrating the Scheduling module with Professionals); and **Infrastructure** (structural components, such as menus, without domain-specific business logic).

Table 1: Distribution of migrated modules by functional type and complexity level.

Functional type	Low	Medium	High	Total
Master Data	2	13	2	17
Transactional	0	1	17	18
Query	0	0	0	0
Report	0	1	1	2
Extension/Association	0	3	0	3
Infrastructure (menus, navigation)	1	0	0	1
Total	**3**	**18**	**20**	**41**

Complexity is determined as the sum of five criteria identified during the evidence elicitation process: (i) involvement of three or more database tables; (ii) non-trivial calculations or conditional business rules; (iii) modification of business state; (iv) dependencies on modules that have not yet been migrated; and (v) external integrations or document generation. Modules meeting zero or one criterion are classified as **low** complexity; those meeting two or three criteria as **medium** complexity; and those meeting four or five criteria as **high** complexity. Table 1 presents the resulting distribution.

Of the 20 high-complexity modules, 17 belong to the *Transactional* category, primarily within the Purchasing, Sales/Tax, and Financial domains, indicating that the modernization effort prioritized core business functionality. In contrast, *Master Data* modules are concentrated in the low- and medium-complexity levels, providing the foundation for the transactional modules. A noteworthy result is the absence of *Query* modules: although not intentionally planned, this distribution suggests that prioritization favored functionalities containing more business logic, leaving query-oriented modules for future migration stages. The complete classification

Table 2: Objective evidence collected to date.

Modernized modules	41
Automated tests (business rules + context)	78+1
Production Firebird data imports	2
Rework at the *Specification Gate*	0
Business rule corrected based on production data	1
AI-suggested architectural restructurings	3

report, including the individual complexity score and the number of tests for each module, is available as a supplementary artifact (see Artifact Availability).

5 Experience Report

To date, 41 modules have been modernized using the SDAW, covering access control, core master data, clinical workflows, scheduling, catalog management, inventory, purchasing, sales, tax documents, and financial management, as well as two production data imports from the Firebird database. Table 2 summarizes the objective evidence collected so far, revisited throughout this section and in Section 6.

Access control. The evidence elicitation process revealed that the legacy system does not employ global roles, but instead relies on user-specific permissions assigned to individual menu items, stored as textual values (`USUARIOACESSO`, `YES/NO`), without explicit guarantees of referential integrity when records are deleted. The modernized system adopts a role-based access control (RBAC) model with roles, permissions, user-specific exceptions, and an audit trail. It replaces the sequential identifier used by each legacy table with a dedicated primary key while preserving a `legacy_id` field for traceability, standardizes the legacy textual `Y/N` indicator into a Boolean `active` field, and enforces referential integrity through foreign key constraints, making deactivation the default operation instead of physical deletion.

Core master data. The Species, Breeds, Coat Color/Pattern, Clients, and Animals modules validated the complete workflow, including evidence elicitation, specification, data model design, implementation, testing, and manual validation. Legacy inconsistencies, such as a defect in photo deletion and the ambiguous naming between "Colors" and "Coat Color/Pattern," were corrected rather than reproduced. Clients was migrated before Animals to avoid temporary references; Animals became the first high-complexity module, depending on four master data modules and requiring cross-validation among them.

Clinical workflows and scheduling. The Anamneses, Hospitalizations, Scheduling, and Professionals modules required decoupling functionalities that, in the legacy system, were concentrated in highly coupled forms. Anamneses and Hospitalizations were modeled as extensions of the Animals domain, preserving compatibility through textual fields until the Professionals domain was introduced. During this stage, the SDAW automatically identified, without manual intervention, which of two Anamnesis form versions in the Delphi source code was authoritative. Scheduling retained only its core functionality (client, animal, professional, time, status, location), delegating sales and cashier operations to dedicated modules; Professionals became an independent domain shared by Scheduling, Anamneses, and Hospitalizations through additive migrations that avoided circular dependencies.

Catalog, inventory, and tax classification. Products and services were consolidated into a shared catalog entity while maintaining separate user interfaces. Product groups and subgroups were unified without imposing a hierarchical structure absent from the legacy system, and Inventory combines current stock balances with movement history. This stage included the first production data import from the Firebird database: the data revealed that a single NCM code may be associated with multiple CEST codes, invalidating a uniqueness constraint inferred solely from the source code. Historical inconsistencies, such as negative stock balances and missing references, were preserved as auditable exceptions rather than silently corrected.

Purchasing and sales. These modules account for most of the high-complexity functionality (Table 1), involving multiple entities, tax rules, and fiscal documents. Business rules previously scattered across forms, *DataModules*, and SQL stored procedures were consolidated into the service layer. In Purchasing, XML import was decoupled from stock entry confirmation. In Sales, the SDAW proposed an aggregate-based model reusing the same components across quotations, orders, sales transactions, and fiscal documents, reducing duplication — one of several AI-suggested restructurings reviewed and approved by the team.

Financial management. Accounts Payable and Accounts Receivable preserved legacy behavior while eliminating direct database coupling. Cancellation and reversal became explicit state transitions, with rules centralized in the service layer and covered by automated tests.

Observed architectural patterns. Three recurring patterns emerged across domains: (i) reorganization of business domains merged into a single table in the legacy system (Clients, Suppliers, Carriers, Professionals); (ii) standardization of master–detail structures (Payment Methods, Price Tables, Purchase/Sales Orders); and (iii) preservation of historical information through explicit state transitions rather than physical deletion. The AI agent played a broader role than a code translator, suggesting domain reorganizations and component reuse, all subject to human validation. Test coverage remains uneven: master data modules are well covered, transactional modules still lack dedicated tests, and continuous integration alone does not reduce *test smells* [13], so future test-suite growth should prioritize quality over coverage alone.

6 Discussion and Lessons Learned

Specification before implementation. Reconstructing and validating a specification before generating code reduced ambiguities: no cases were identified in which the AI introduced business rules that were not present in the legacy system, and the required adjustments were limited to user interface and user experience aspects. Nevertheless, a specification reconstructed solely from the source code does not eliminate the need for validation against production data. The Firebird data import revealed a tax rule that was inconsistent with the one inferred through reverse engineering, reinforcing that

evidence elicitation and empirical validation are complementary rather than interchangeable.

Incremental migration strategies. Starting with a complete vertical slice—database, API, user interface, and automated tests—allowed the workflow to be validated before scaling it to additional modules. Cross-cutting dependencies, such as the Professionals domain, were addressed through additive migrations rather than delaying dependent modules. Preserving the `legacy_id` field facilitated both incremental migration and future data imports, although it required architectural adaptations to align the modernized application with common web-based design principles.

AI as support for architectural decision-making. The AI agent proposed domain reorganizations, component reuse across similar functionalities, and the replacement of physical deletions with explicit state transitions, and identified non-functional risks, such as the unexpected growth of the Angular application bundle. All suggestions were subject to human review. This experience suggests that coding agents can effectively support architectural and quality-related decision-making when used as an aid to, rather than a replacement for, the software design process.

Effort and cost of use. Functionalities integrated at later stages required broader contextual information and, consequently, greater computational resources. At the same time, reusing components from previously migrated modules produced more consistent solutions, albeit at the cost of increased token consumption. Different variants of the model were employed according to quota availability rather than experimental criteria, reinforcing the need to systematically record which model is used at each stage of the modernization process.

Architectural standardization. Establishing reference patterns for components, forms, and listing pages before generating new modules tends to reduce rework and unnecessary application growth, resulting in a more consistent user interface throughout the modernization process. Overall, the primary benefit of the SDAW was not merely to accelerate implementation, but to establish a workflow in which specification, human review, and AI-assisted generation complement one another, ensuring that architectural decisions remain under human control.

7 Threats to Validity

This work is an ongoing experience report based on a single legacy system, without a control group or systematic collection of quantitative metrics from the outset, and subject to evaluation bias, as the same developers who conducted the modernization also assessed its outcomes, without independent review or end-user validation. To date, only the Tax Classification and Products modules have been validated against production data from the Firebird database; the remaining modules may still contain undetected semantic differences, with this risk being particularly significant for the Financial modules (Accounts Payable and Accounts Receivable), where errors may have direct financial consequences. Automated test coverage is also uneven, being concentrated on foundational master data modules while transactional modules remain supported by relatively few tests. Finally, different AI model variants were selected based on quota availability, and computational budget constraints influenced scoping and prioritization decisions, including some suggested by the AI agent itself, making it difficult to isolate the effects of the proposed workflow from the limitations of the development environment.

8 Future Work

Although 41 of the 70 modules have already been modernized, the migration remains an ongoing effort. The next steps include: expanding the migration to modules not yet represented in the current sample, particularly those classified as *Query*, and completing the remaining reporting modules; comparing the adopted vertical migration strategy with a horizontal approach, evaluating implementation effort, token consumption, and rework; further consolidating the SDAW through a catalog of reusable components and interface patterns, increased test coverage for high-complexity modules, and the migration of data from the remaining modules while preserving the `legacy_id` reconciliation mechanism; and conducting an external evaluation with end users of the legacy system. Finally, the SDAW will be instrumented to systematically collect objective per-module metrics — implementation time, number of iterations, model variant, and the agent's participation in architectural decisions.

9 Conclusion

This paper presented an experience report on the ongoing modernization of a legacy management system for pet shops and veterinary clinics, migrated from Delphi/Firebird to a Java/Spring Boot, Angular, and MySQL architecture through the SDAW, in which a specification reconstructed from the legacy system is reviewed by humans before AI-assisted implementation. Applied to 41 modules and two production data imports from the Firebird database, the SDAW reduced ambiguities during implementation and enabled the AI agent to support not only code generation but also the identification of inconsistencies and the proposal of architectural restructurings, all of which were subject to human review. At the same time, validation against production data demonstrated that reverse engineering based solely on source code is insufficient to capture all business rules, reinforcing the need to combine specification reconstruction, human review, and empirical validation. Although this experience report represents a single ongoing case study, our experience suggests that combining specification reconstruction, human validation, and LLM-assisted implementation is a promising strategy for reducing risks in legacy system modernization.

Artifact Availability

The specification documents produced during the modernization process reported in this paper are publicly available in a Zenodo repository (https://zenodo.org/records/21325175). Production data extracted from the Firebird database and the source code of both the legacy and modernized systems are not publicly released because they contain proprietary and commercially sensitive information.

References

[1] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. 2024. Automatic Library Migration Using Large Language Models: First Results. In *Proceedings of the 18th*

ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 427–433. doi:10.1145/3674805.3690746

[2] Andréa Bordin and Luiza Garcia. 2024. An Industry View about Legacy Systems. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 137–148. doi:10.5753/vem.2024.3914

[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. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 2719–2744. doi:10.18653/v1/2025.findings-acl.140

[4] Sanderson Oliveira de Macedo and Ronaldo Martins da Costa. 2026. Reversa: A Reverse Documentation Engineering Framework for Converting Legacy Software into Operational Specifications for AI Agents. arXiv:2605.18684 [cs.SE] https://arxiv.org/abs/2605.18684

[5] Wellynton Diniz, Marcela Valença, César França, Alessandro Santos, and Mariana Pincovsky. 2024. The Skill Gap in Software Industry: A Mapping Study. In *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 192–200. doi:10.5753/sbes.2024.3351

[6] Giovanna Fernandes, Marcelo Maia, and Carlos Dantas. 2025. How Readable Is LLM-Generated Code Snippets? A Comparison of ChatGPT, DeepSeek, and Gemini. In *Anais do XIII Workshop de Visualização, Evolução e Manutenção de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 13–24. doi:10.5753/vem.2025.14275

[7] Mohayeminul Islam, Ajay Kumar Jha, May Mahmoud, Ildar Akhmetov, and Sarah Nadi. 2025. An Empirical Study of Python Library Migration Using Large Language Models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 867–879. doi:10.1109/ASE63991.2025.00077

[8] Suman Jain and Inderveer Chana. 2015. Modernization of Legacy Systems: A Generalised Roadmap. In *Proceedings of the Sixth International Conference on Computer and Communication Technology 2015* (Allahabad, India) *(ICCCT '15)*. Association for Computing Machinery, New York, NY, USA, 62–67. doi:10.1145/2818567.2818579

[9] Ravi Khadka, Belfrit V. Batlajery, Amir M. Saeidi, Slinger Jansen, and Jurriaan Hage. 2014. How do professionals perceive legacy systems and software modernization?. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 36–47. doi:10.1145/2568225.2568318

[10] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3597503.3639226

[11] Vinicius Pimenta, Elder Cirilo, and Ricardo Terra. 2024. Troca de Bibliotecas em Sistemas com e sem Arquitetura Limpa: Uma Análise de Esforço. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 23–34. doi:10.5753/vem.2024.3829

[12] Deepak Babu Piskala. 2026. Spec-Driven Development:From Code to Contract in the Age of AI Coding Assistants. arXiv:2602.00180 [cs.SE] https://arxiv.org/abs/2602.00180

[13] Edson Silva, Rodolfo Rodrigues, Johnatan Oliveira, Danilo Boechat, and Cleiton Tavares. 2024. Evaluating Test Quality in GitHub Repositories: A Comparative Analysis of CI/CD Practices Using GitHub Actions. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 45–55. doi:10.5753/vem.2024.3842

[14] Julyanara Silva, Carlos Dantas, and Marcelo Maia. 2024. What Developers Ask to ChatGPT in GitHub Pull Requests? an Exploratory Study. In *Anais do XII Workshop de Visualização, Evolução e Manutenção de Software* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 125–136. doi:10.5753/vem.2024.3913