OAMK OULUN AMMATTIKORKEAKOULU

ARCHITECTURAL MODERNIZATION OF LEGACY PHP SYSTEMS

PERFORMANCE, MAINTAINABILITY, AND THE ROLE OF AI-ASSISTED DEVELOPMENT

Mariia Glushenkova
Bachelor's Thesis
Autumn 2025
Degree Programme in Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology
Option of Web Development

Author: Mariia Glushenkova

Title of thesis: Architectural Modernization of Legacy PHP Systems. Performance, Maintainability, and The Role of AI-Assisted Development.

Supervisor: Janne Kumpuoja
Term and year of completion: Autumn 2025
Pages: 70

Legacy applications built with a mix of HTML, PHP, and JavaScript are still widely used, but their tightly coupled structure makes them difficult to maintain, extend, and secure. This thesis explores how such systems can be effectively modernized by redesigning a representative part of the Digitized Commitment Management (DCM) tool into modular React and Node.js architecture. The goal was to improve maintainability and performance while introducing a clearer and more consistent application structure.

The modernization effort included rebuilding the frontend as a component-based React application, introducing a dedicated Node.js API layer with parameterized SQL queries, and replacing ad-hoc PHP session handling with a Microsoft Entra ID authentication flow. These changes separated concerns that had previously been mixed, resulting in a cleaner, more reusable, and easier-to-maintain codebase. AI tools were used in a supportive role to speed up repetitive tasks such as scaffolding and pattern extraction. All generated output was manually reviewed to ensure correctness and security, making AI an accelerator rather than a decision-maker. The modernized solution was evaluated across performance, maintainability, security, and developer productivity.

The results show clear improvements: significantly faster page rendering, reduced code duplication, stronger security through a centralized backend, and significantly more maintainable architecture. The results show that architectural redesign is the primary driver of modernization success. When combined with disciplined, review-driven AI assistance, modernization becomes both technically feasible and organizationally sustainable. The study demonstrates how legacy PHP systems can evolve into modular, maintainable, and secure platforms aligned with contemporary enterprise standards.

Keywords: legacy modernization, maintainability, React, Node.js, PHP, software architecture, AI-assisted development, scaffolding, performance

CONTENTS

ABSTRACT 2
CONTENTS 3
GLOSSARY 6
ACKNOWLEDGEMENTS 8
1 INTRODUCTION 9
1.1 Background 9
1.2 Problem Statement 10
1.3 Objectives 10
1.4 Scope and Limitations 11
2 LITERATURE REVIEW 13
2.1 Persistence and Challenges in Legacy Intranet Systems 13
2.2 Incremental Modernization: Patterns over Perfection 14
2.3 Codemods as one of the modernization strategies 14
2.4 AI-Assisted Development Tools in coding environment 16
2.5 Security Considerations in Migration 16
2.6 Measuring Modernization Success 17
2.7 Synthesis and Research Gap 18
3 METHODOLOGY AND IMPLEMENTATION 19
3.1 Application to be modernized 19
3.2 Modernization Strategy 20
3.3 AI-Assisted Development Workflow 21
3.4 System Architecture Overview 22
4 USE CASE OVERVIEW 24
5 ENGINEERING IMPLEMENTATION 26
5.1 Technology Stack Selection 26
5.3 Frontend Implementation 27
5.3.1 React Architecture 27
5.3.2 AG Grid Proof of Concept 28
5.3.3 Modernized Graphical User Interface 28
5.3.4 Authentication and Authorization 33

5.4 Backend Implementation and Token Validation 33
5.4.1 Azure Active Directory and RBAC Configuration 34
5.5 Summary 35
6 TECHNICAL COMPARISON OF THE LEGACY APPLICATION AND THE MODERN PROTOTYPE 37
6.1 Architectural Contrast 37
6.2 Observed Improvements 40
6.2.1 Performance 40
6.2.2 Maintainability 41
6.2.3 Security 42
6.2.4 Scalability and Operations 43
7 USAGES OF AI DURING IMPLEMENTATION 44
7.1 AI Implementation Approach 44
7.2 Practical Use Cases of AI 44
7.2.1 Transforming Inline SQL into Express Routes 45
7.2.2 Generating Deterministic RBAC Merge Logic 45
7.2.3 Detecting Security Vulnerabilities 45
8 EVALUATION AND RESULTS 48
8.1 Performance Profiling and Evaluation 48
8.1.1 Measurement methods 49
8.1.2 Lighthouse Summary 50
8.1.3 Page-Load and Network Metrics 51
8.1.4 Runtime Profiling 52
8.1.5 RBAC Initialization 54
8.1.6 Summary of Findings 54
8.2 Maintainability Evaluation 55
8.3 Security, Identity, and Future Hardening 56
8.3.1 Security Transformation Through Modern Architecture 56
8.3.2 Legacy Authentication and Authorization Weaknesses 57
8.3.3 Backend API Security and Middleware Enforcement 58
8.3.4 Transport and Header-Level Security Enhancements 59
8.3.5 Reliability Improvements Through Centralized User State and Hooks 59
8.4 Areas for Further Improvement 59

8.5 Conclusion 60
9 DISCUSSION AND LESSONS LEARNED 62
9.1 Understanding Legacy Complexity 62
9.2 AI Assistance Advantages and Disadvantages 62
9.3 Security as a Continuous Discipline 63
9.4 Balancing Performance and Maintainability 63
9.5 Collaboration and Knowledge Transfer 64
9.7 Opportunities for Improvement 64
9.8 Conclusion 65
REFERENCES 66

GLOSSARY

Acronym / Term	Definition
AG Grid	High-performance JavaScript data grid for large datasets in the frontend.
Azure SQL	Microsoft cloud database service used in the modern backend.
BFF (Backend for Frontend)	Architectural pattern where each frontend has a dedicated backend tailored to its needs.
CI/CD	Continuous Integration and Continuous deployment pipelines for automated building, testing and deployment.
CSP (Content Security Policy)	Security mechanism using HTTP headers to prevent XSS and data injection attacks.
CORS (Cross-Origin Resource Sharing)	Browser security mechanism restricting cross-domain HTTP Requests.
DCM (Digitized Commitment Management Tool)	Internal enterprise tool used as the modernization case study in this thesis.
DOM Purify	JavaScript sanitization library used in the legacy PHP code to prevent XSS vulnerabilities.
Express.js	Minimalist Node.js web framework used to build RESTful API routes in the modern BFF.
GitLab Actions / CI	GitLab's automation pipelines for continuous integration, testing, and deployment.
Helmet	Node.js middleware that secure HTTP headers to protect against common web vulnerabilities.
JWT (JSON Web Token)	Secure token format used for identity, authentication, and authorization in the front end and backend flow.
LCP (Largest Contentful Paint)	Web performance metric measuring how quickly the largest visible content element appears.
Microsoft Entra ID	Cloud identity and access management service used for authentication and RBAC in the modern system.

Acronym / Term	Definition
MSAL (Microsoft Authentication Library)	Library used to integrate Microsoft Entra ID authentication and token handling.
Node.js	JavaScript runtime used to implement the backend BFF API.
OAuth 2.1 Authorization Code Flow	Authentication flow used with MSAL for secure authentication.
OWASP	Community foundation publishing widely used web security guidelines.
PKCE (Proof Key for Code Exchange)	Security enhancement for OAuth 2.1 authorization flows, required for SPAs using MSAL.
RBAC (Role-Based Access Control)	Access control model where permissions are granted based on user roles.
SAST/ DAST (Static and Dynamic Application Security Testing)	Methods for detecting vulnerabilities in source code and running applications.
SEO (Search Engine Optimization)	The practice of enhancing a website's visibility and ranking on search engines like Google to increase organic traffic.
SPA (Single-Page Application)	Web application model where a single HTML shell loads once and dynamically updates via JavaScript.
SQL Injection	Attack that exploits unvalidated raw SQL concatenation.
Tailwind CSS	Utility-first CSS framework used for styling the modern React frontend.
Tabulator	Data table library used in the legacy PHP application.
TypeScript	Typed superset of JavaScript improving reliability.
Vite	Modern frontend bundler used to build the SPA.
Web Vitals	Google metrics for evaluating page performance.
XSS (Cross-Site Scripting)	One of the vulnerability attacks for website.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Nokia for the opportunity to conduct this thesis in an industrial environment and for trusting me with such a meaningful project. The work strengthened my belief that modernization is not merely a technical upgrade, but a way to rethink how people, tools, and processes evolve together.

My sincere thanks go to my colleagues for their continuous guidance, insightful discussions, and constructive feedback throughout this project. Their expertise and encouragement were crucial in transforming initial concepts into measurable engineering results.

I am especially indebted to my family and friends, who patiently supported me, offered feedback on my writing, and encouraged me to keep going when the workload was most demanding.

To everyone who contributed their time, knowledge, and encouragement — thank you.

1 INTRODUCTION

1.1 Background

Enterprise intranet systems have long relied on PHP-based architectures that mix backend logic, HTML templates, and client-side JavaScript within the same files. These systems were effective when initially developed, offering rapid deployment, simple hosting, and broad developer familiarity. Over time, however, their tightly coupled structure creates significant challenges. Business logic, database access, and presentation often coexist within single PHP files, making the codebase difficult to understand, modify, or extend.

Despite their limitations, PHP systems remain widespread. As of 2023, approximately three-quarters of websites that use a well-known server-side technology rely on PHP (W3Techs 2024). Much of this ongoing use is driven by platforms such as WordPress, but many enterprises also maintain business-critical PHP applications that have evolved organically over long periods. Even though newer releases such as PHP 8.1 and 8.2 offer major improvements in performance, typing, and security, large organizations still run older versions—such as PHP 7.4, which reached end-of-life in 2022—because modernization cycles are slow and risk-averse in regulated environments (Endoflife.date 2025).

Modern frontend and backend frameworks address many of the architectural limitations of legacy PHP systems. React introduces a component-based development model with predictable state management, while Node.js provides an asynchronous and modular backend environment suited for scalable web applications. In parallel, AI-assisted development tools have emerged as accelerators for routine coding tasks. Prior research shows that these tools can improve developer productivity, but they often generate incomplete or insecure logic if used without careful review (e.g., Vaithilingam et al., 2023; Cazzola & Favalli, 2024).

This thesis investigates whether AI-assisted development can support the modernization of a legacy application while complying with enterprise governance and security constraints. It evaluates both the architectural outcomes of modernization and the reliability of AI-assisted code generation under practical conditions.

1.2 Problem Statement

The modernization of the DCM system centers on a maintainability problem. The legacy PHP architecture mixes UI rendering, business logic, and SQL operations within the same files, making the codebase difficult to understand, test, or extend. Even small changes require navigating thousands of lines of intertwined PHP, HTML, and JavaScript, where hidden dependencies and inconsistent patterns can introduce unpredictable side effects.

Recurring issues include duplicated CRUD and validation logic, reliance on global state and cross-file includes, inconsistent error handling and role checks, and the absence of a modular structure that would support incremental refactoring or reuse. These constraints significantly increase development effort and long-term maintenance risk.

The core question is whether the DCM system can be redesigned into a modular, predictable architecture that improves maintainability while preserving existing business logic and operational reliability. Although AI tools can accelerate scaffolding work, the fundamental challenge lies in transforming a tightly coupled legacy module into a cleanly separated, sustainable system architecture.

1.3 Objectives

The main objective of this thesis is to determine whether a modern React–Node.js architecture can improve the maintainability, performance, and security of a representative module of the DCM system when compared with its legacy PHP implementation. To achieve this, the study first identifies structural weaknesses in the existing PHP pages, focusing on duplicated logic, tightly coupled

components, and inconsistent validation practices that limit maintainability. The selected DCM views are then rebuilt using React, Node.js, and Microsoft Entra ID, applying modular architecture principles such as componentization, reusable hooks, and clearly separated service layers. The outcomes of this modernization are measured using quantifiable indicators, including code duplication ratios, file size reduction, complexity scores, API latency, and rendering performance. The study also evaluates how AI-assisted development can support repetitive scaffolding tasks without compromising quality or security. Finally, it documents the risks, limitations, and organizational considerations relevant to adopting modern frameworks within a corporate environment.

1.4 Scope and Limitations

The scope of this thesis covers an architectural assessment of the DCM system, with a particular focus on authentication, authorization, Role-Based Access (RBAC) structures, and the overall maintainability of the legacy PHP codebase. The work includes analysing the existing architecture end-to-end—frontend, backend, identity flow, and data interactions. The goal is to identify the key technical challenges and structural constraints of the current implementation. Building on this analysis, the thesis develops a modernized architecture based on React, Node.js, and Microsoft Entra ID, and implements a proof-of-concept (PoC) that demonstrates how these technologies can replace the existing monolithic PHP design. The full application is not rebuilt as a production-ready system. Instead, the PoC validates the modernization strategy, confirms end-to-end token-based authentication and RBAC enforcement, and provides a modular structure that can be extended to the rest of the system. Several limitations define the boundaries of this work. Because only a single module was modernized with complete functional parity, performance and maintainability results apply specifically to this module rather than the entire application. AI assistance was deliberately restricted to scaffolding, pattern recognition, and refactoring support, with all generated code reviewed manually to meet enterprise governance requirements. Additionally, the existing database schema and stored-procedure

logic were preserved unchanged to maintain compatibility with production workflows and avoid disrupting business-critical operations.

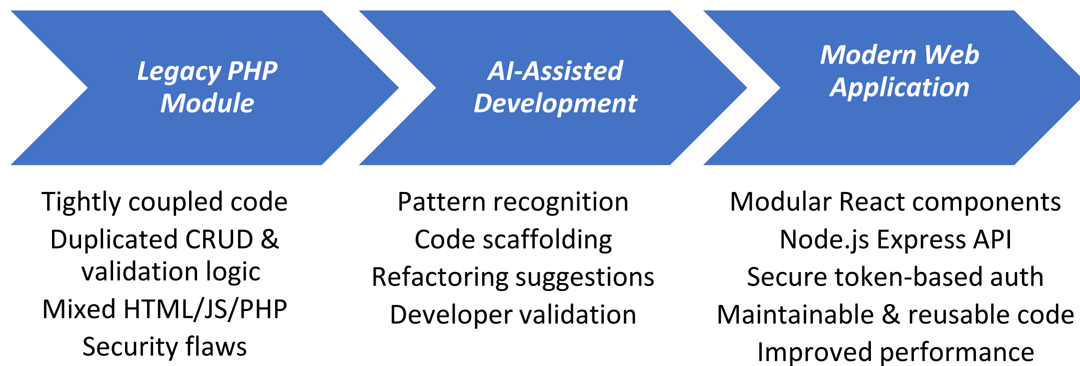


FIGURE 1. Scope of the modernization: legacy system analysis, full-stack redesign, and PoC implementation.

Figure 1 illustrates how the thesis covers full-system architectural analysis and identity flow redesign, while the practical implementation focuses on a single representative module to validate the feasibility of the target architecture.

2 LITERATURE REVIEW

This chapter reviews existing research on legacy-system modernization and AI-assisted software development. The goal is to identify what has been established, where current approaches fall short, and how these gaps justify the empirical work presented in later chapters.

2.1 Persistence and Challenges in Legacy Intranet Systems

Although many enterprise intranet systems were developed well after the early 2000s, a significant number, including the system examined in this thesis, were still built using architectural practices characteristic of older PHP applications. In our case, the system was implemented in 2022, yet it followed a traditional PHP monolith pattern in which HTML, JavaScript, backend logic, and database access were tightly interwoven within the same files. This enabled rapid development but created structural constraints: repeated CRUD logic, duplicated validation rules, and minimal separation between presentation and business logic.

While parameterized SQL queries were used correctly, the broader application architecture lacked modern design principles such as modular data-access layers, standalone APIs, and component-based frontends. As the system grew, these limitations accumulated into technical debt, making the codebase harder to maintain, test, and evolve.

Contemporary studies on legacy intranet platforms—particularly in government, healthcare, and financial organizations—show that enterprise systems continue to operate on outdated PHP versions (5.x–7.x) and unsupported frameworks such as Drupal 7. Despite the risks associated with aging technology, full rewrites are often postponed due to high costs, uncertainty around migration, and the need to maintain continuous operations (Endoflife.date 2025; PHP Group 2025; Henderson 2020). These pressures help explain why legacy intranet architectures are still prevalent even when newer development paradigms are already available.

2.2 Incremental Modernization: Patterns over Perfection

Because full rewrites often fail due to scope creep, integration issues, or user disruption, industry increasingly supports incremental modernization strategies. Fowler’s Strangler Fig pattern provides one widely adopted approach: introducing modern components alongside legacy ones and gradually redirecting traffic until the old system can be retired (Fowler 2004). Complementary patterns such as Newman’s Backend-for-Frontend (BFF) further separate concerns by giving each frontend its own tailored backend service, improving modularity and easing staged migration (Microsoft 2025b; Thoughtworks 2025b).

Industry surveys, including Stack Overflow (2023), show strong adoption of such progressive techniques. Companies like Facebook and Airbnb prove how gradual React integration can improve maintainability without causing service interruptions. However, as Garousi and Felderer (2025) note, empirical research confirming these strategies within highly regulated enterprise environments remains limited. This lack of real-world evidence underscores the value of practical case studies such as the one presented in this thesis.

2.3 Codemods as one of the modernization strategies

Automated approaches often rely on codemods—tools that apply abstract syntax tree (AST) transformations to upgrade syntax or APIs with minimal manual intervention. In clean, well-structured codebases, tools such as jscodeshift and Rector have been shown to reduce development time by up to 50% (Rector Project 2025). However, these tools require clear language boundaries and predictable syntax structures.

Legacy intranet systems rarely meet these conditions. They commonly interlink PHP, HTML, JavaScript, and SQL within the same file, sometimes with dynamic evaluation or runtime reflection. These mixed-language environments significantly limit the applicability of codemods. Additionally, enterprise governance often restricts installing Node.js, Composer, or experimental

transformation tool chains on production servers, further hindering automation. (Comella-Dorda et al. 2001.)

Because of these constraints, fully automated modernization proves infeasible in environments such as the DCM intranet system. Instead, hybrid AI-assisted scaffolding approach becomes more practical—combining pattern recognition from AI models with developer oversight.

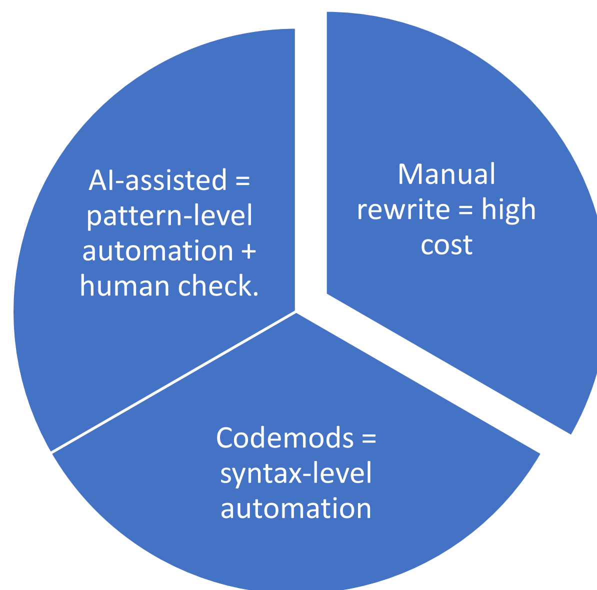


FIGURE 2. Different modernization strategies.

Figure 2 depicts three primary strategies for legacy system modernization. Manual rewrites achieve full control over code quality but come with high cost and time requirements. Codemods provide syntax-level automation to refactor predictable patterns, yet they struggle with mixed-language or irregular legacy structures. AI-assisted modernization combines pattern-level automation with developer verification, striking a balance between efficiency and accuracy while maintaining engineering oversight.

2.4 AI-Assisted Development Tools in coding environment

Recent advances in AI-assisted development tools such as GitHub Copilot, ChatGPT, and models trained on resources like IBM CodeNet have enabled developers to automate repetitive programming tasks, generate scaffolding, and identify structural patterns. Controlled studies report productivity improvements of up to 56% (Vaithilingam et al., 2023). However, research also shows that a considerable proportion of generated code contains logical inconsistencies or security flaws, underscoring the need for manual review (Pearce et al., 2022; Cazzola & Favalli, 2024).

In practice, AI functions as an accelerator rather than a replacement for engineering judgement. It can efficiently produce CRUD templates, routing logic, or component structures, but it cannot guarantee correctness, maintainability, or security. Developer oversight remains essential to validate architectural fit, ensure parameterization and safe authentication flows, and enforce enterprise coding standards. Prior research confirms that AI tools improve productivity in routine tasks but frequently generate incorrect or insecure code, reinforcing the need for human verification (Vaithilingam et al., 2023; Pearce et al., 2022; Cazzola & Favalli, 2024). While prior work evaluates AI's influence on productivity and defect rates, there is limited research exploring AI-assisted modernization inside enterprise environments, particularly those with strict governance requirements, offline-only tooling, and complex mixed-language legacy stacks. The present thesis contributes to this underexplored context.

2.5 Security Considerations in Migration

Modernizing legacy systems introduces significant security implications. Historical incidents such as the 2017 Equifax breach [1] illustrate how outdated

---

[1] The 2017 Equifax breach occurred when attackers exploited an unpatched Apache Struts vulnerability (CVE-2017-5638), allowing remote code execution on a legacy web application. The incident exposed sensitive data of approximately 147 million individuals and is widely cited as an example of the severe risks posed by outdated frameworks, insufficient patching practices, and legacy system architectures.

frameworks, unpatched components, and misconfigurations can create catastrophic vulnerabilities. Similar analyses of university and government intranets reveal recurring issues with missing CSRF tokens, unparameterized SQL queries, and obsolete libraries (OWASP Foundation 2025). Security by design is therefore essential during modernization. Practices such as parameterized queries, strict token-based authentication, RBAC, secure headers, Content Security Policy, and adherence to OWASP Top Ten (2021) must be incorporated from the start. Otherwise, new architectures risk reintroducing old vulnerabilities behind modern user interfaces.

2.6 Measuring Modernization Success

Modernization outcomes must be measured using objective criteria rather than aesthetics. Prior literature identifies four primary evaluation dimensions as described in TABLE 1. These dimensions are typically evaluated using tools such as Google Lighthouse for performance, ESLint and Sonar for static code analysis, OWASP checklists for security assessment, and various productivity metrics to measure development efficiency.

TABLE 1. Evaluation metrics based on performance, maintainability, security and productivity.

Dimension	Metric	Example Tool
Performance	Load time, API latency	Lighthouse, Web Vitals
Maintainability	Code duplication, modularity	ESLint, static analysis
Security	Vulnerability surface, RBAC completeness	OWASP checklist
Productivity	Developer effort, AI reuse rate	Time logs, task metrics

Prior modernization studies report 30–50% faster builds and improved maintainability through component reuse (Fowler, 2018). These metrics form the evaluation baseline for this thesis.

2.7 Synthesis and Research Gap

The literature reveals several consistent themes relevant to this study. Legacy PHP-based intranet systems are challenging to modernize because they often contain mixed-language files and tightly coupled architectures that resist automated refactoring. Incremental modernization strategies, such as the Strangler Fig pattern, are therefore viewed as more practical and at lower risk than full system rewrites. Although automation tools such as codemods can streamline syntax-level transformations, they perform poorly in hybrid or irregular legacy environments where PHP, HTML, JavaScript, and SQL coexist in a single file. Recent research also shows that AI tools can accelerate development by assisting with scaffolding and pattern recognition, but they frequently introduce issues related to correctness, maintainability, and security if used without careful oversight. Together, these findings highlight the need for real-world case studies that explore modernization approaches in complex enterprise intranet systems.

3 METHODOLOGY AND IMPLEMENTATION

This study applies a design-science methodology that combines practical software engineering with empirical evaluation. Rather than analysing modernization techniques conceptually, a fully functional proof-of-concept (PoC) was designed and built to test modernization outcomes in a controlled, realistic environment. The process consisted of four iterative phases as described in Figure 3.

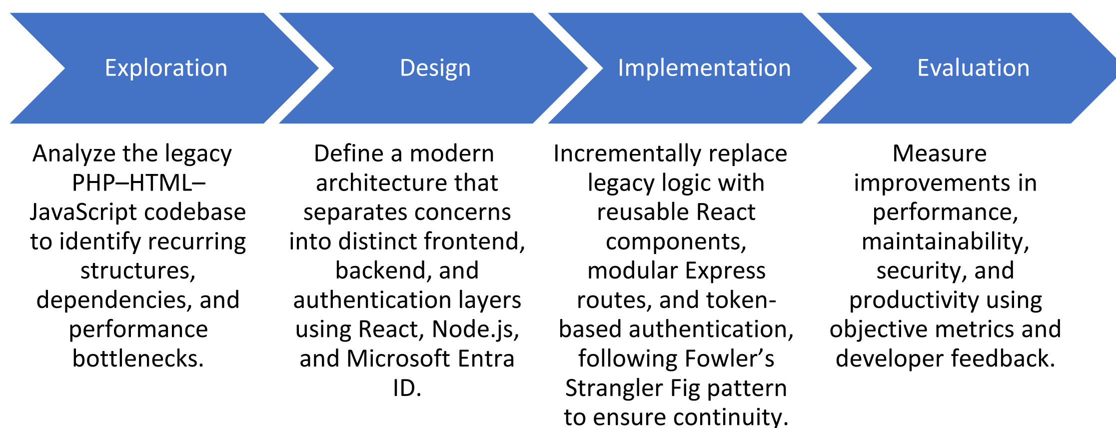


FIGURE 3. Proof of Concept stages.

This approach ensured that theoretical research was grounded in observable engineering results.

3.1 Application to be modernized

The case study for this research is the Digitized Commitment Management (DCM) tool—an enterprise intranet application that exemplifies the challenges of maintaining long-lived PHP systems. The application consists of intermixed PHP, inline JavaScript, and HTML templates, alongside mixed-technology files

responsible for rendering tables, processing forms, and supporting workflow automation. Its tightly coupled monolithic structure reflects early-2000s design, where presentation, logic, and database access are embedded within the same PHP files.

While the thesis modernized the application layer, including the frontend architecture, backend API, authentication flow, and shared utilities, one business-critical page was selected for deep technical analysis. This page was selected because it holds high business relevance and is used daily across multiple departments. From a technical perspective, it presents substantial complexity, combining SQL queries, inline JavaScript, and server-side rendering within a single PHP file. It is also architecturally representative of the broader system, reflecting recurring patterns such as CRUD operations, validation logic, and conditional rendering.

As a result, it provides a realistic and meaningful basis for evaluating the modernization approach. Focusing on a single page for measurement provided a realistic but bounded evaluation environment, while the broader modernization works improved maintainability throughout the application.

3.2 Modernization Strategy

The modernization followed an incremental Strangler Fig strategy, allowing legacy and modern components to coexist until full functional parity was achieved. This avoided the operational risks of a full rewrite and enabled continuous validation.

Implementation began with a structured baseline analysis of the legacy system's logic flows, parameters, database dependencies, and user interactions. A new architecture was then defined that separated responsibilities into distinct layers. The frontend was rebuilt from scratch with the usage of React, TypeScript, Tailwind CSS, and AG Grid for efficient data visualization. The backend was reimplemented with Node.js and Express using a modular routing structure and parameterized SQL queries.

While the legacy system already used Microsoft Authentication Library (MSAL) for authentication, its backend still relied on PHP session cookies to maintain state. This partial integration led to inconsistent authentication logic across layers. In the modernized system, authentication is now fully unified: Microsoft Entra ID issues signed JWT tokens, which are validated by both the frontend and the Node.js backend. The MSAL React library handles token acquisition and renewal, ensuring a secure and stateless identity model across the entire stack.

AI-assisted development played a supporting role. Tools such as ChatGPT and NokiaGPT were used to generate repetitive scaffolds such as form handlers, table definitions, and validation hooks. Ready components were then manually reviewed, corrected, and integrated. Extensive manual testing ensured that the modernized version matched the functional scope of the legacy page before retiring old components.

3.3 AI-Assisted Development Workflow

AI support was conducted using available models and structured prompts. Each interaction was logged and manually reviewed, with AI involvement limited to three clearly defined areas as described below.

TABLE 2. Stages, AI Involvement, and Process of AI Usage.

Stage	AI Task	Developer Validation
Frontend scaffolding	Generate initial React components, hooks, and contexts	Adjust naming, integrate with TypeScript, ensure state safety
Backend routing	Draft Express endpoints and query handlers	Verify parameterization, authentication, and logical correctness
Documentation	Summarize technical design decisions	Edit for precision and enterprise compliance

Roughly 55% of the frontend scaffolding originated from AI output, providing a measurable productivity boost without compromising code quality.

3.4 System Architecture Overview

The finalized design adopts a Backend-for-Frontend (BFF) architecture, ensuring that each frontend interface communicates with a dedicated, lightweight backend service optimized for its specific requirements.

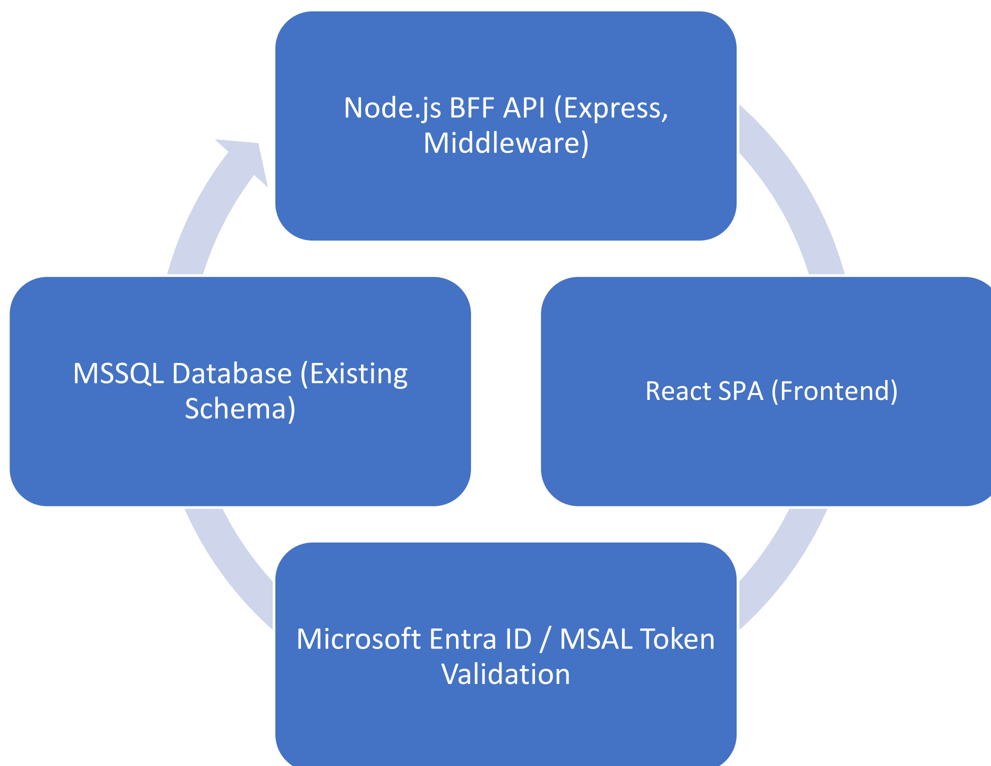


FIGURE 4. System Architecture Overview.

Figure 4 shows the four core components of the modernized application. The React SPA handles user interaction and acquires tokens through Microsoft Entra ID using MSAL. The Node.js BFF API processes all client requests, applies middleware-based token and role checks, and performs validated database operations. The MSSQL database retains the existing schema, serving as the system's data source.

The modernized system is structured into three coordinated layers. The frontend is implemented as a TypeScript-based React single-page application that follows a modular component architecture, with authentication handled through Microsoft Entra ID using the MSAL React library and the OAuth 2.1 authorization code flow with PKCE. Role-Based Access Control (RBAC) is evaluated on both the client and server to ensure consistent enforcement and restrict unauthorized functionality. The backend operates as a Node.js Backend-for-Frontend (BFF) built with Express.js, providing domain-specific routes protected by middleware responsible for JWT verification, role checks, and input validation, while security headers are applied using Helmet in line with OWASP Top 10 guidance. The data layer retains the existing MSSQL schema for compatibility, with Prisma added as a type-safe abstraction to standardize query logic and reduce injection risk. All communication is secured over HTTPS, with tokens validated against Microsoft's public signing keys and RBAC logic enforced redundantly to minimize the attack surface. Together, these layers establish a clear separation of concerns, improve testability and scalability, and create a robust foundation for future containerization or microservice-oriented expansion.

4 USE CASE OVERVIEW

The modernization effort focused on a real operational workflow within the Digitized Commitment Management (DCM) tool. It is used daily by product management, regional business contacts, and leadership teams who handle thousands of feature records during planning and reporting cycles. Their routine tasks include filtering and sorting large datasets, updating committed dates and technology classifications, validating dependencies, exporting data for management use, and coordinating across multiple teams under tight deadlines.

Although the legacy PHP implementation remained functional, its structure had grown increasingly fragile due to years of ad-hoc patches, outdated libraries, and browser-specific workarounds that intertwined PHP, JavaScript, and SQL into tightly coupled files. As modern browsers evolved, new issues began to appear—not because of PHP itself, but due to legacy front-end integrations failing under updated security policies and JavaScript runtimes.

Testing revealed problems such as a critical CORS-related failure in Chrome caused by inconsistent headers and the degradation of previously reliable features like the mailto: invocation in Microsoft Edge, which became unstable due to tightened handler policies. These incompatibilities exposed deeper architectural limitations and demonstrated that, despite acceptable performance, the system could no longer guarantee long-term maintainability, extensibility, or compliance with modern standards.

This use case therefore provided an ideal basis for modernization: it represents a typical high-volume intranet workflow, stresses the system with complex data interactions, and allows clear comparison between the legacy and React–Node.js implementations across maintainability, usability, security, and performance. As shown in Figure 5, it anchors the engineering decisions in the following chapter and establishes a concrete foundation for evaluating the modernization results.

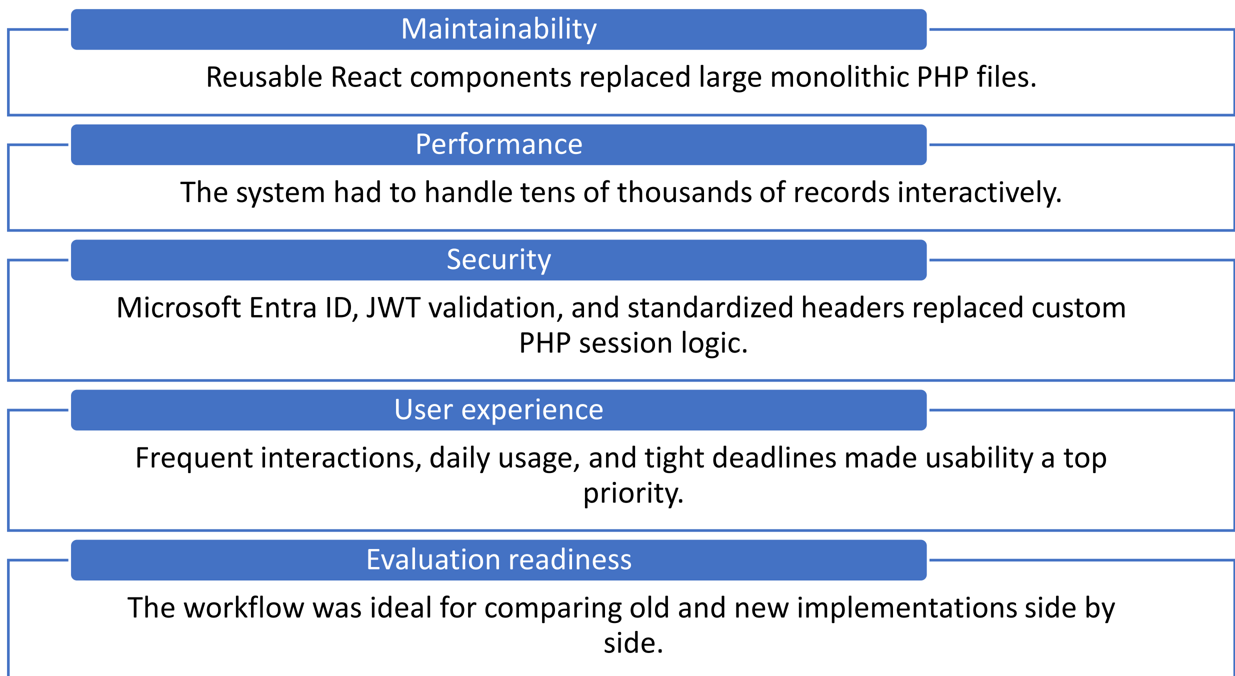


FIGURE 5. Use case scenario.

This case serves as a reference point for the engineering decisions described in the next chapter and provides a concrete foundation for the subsequent evaluation.

5 ENGINEERING IMPLEMENTATION

This chapter describes the practical steps taken to modernize the selected legacy application and outlines the technical decisions that shaped the new system. The implementation focused on translating the original monolithic PHP–JavaScript application into a modular React–Node.js architecture while preserving functional parity, improving maintainability, and strengthening security

5.1 Technology Stack Selection

The legacy intranet relied on a monolithic mix of PHP, inline JavaScript, and MySQL queries, which limited maintainability and made modernization difficult. To address these issues, the new implementation uses React 18 with TypeScript for the frontend and Node.js 20 with Express 5 for the backend, supported by the *MySQL* driver and Microsoft Azure SQL. Authentication and RBAC were implemented with Microsoft Entra ID and MSAL React, while Gitlab Actions handled automated linting, building, and testing.

This stack was chosen for three reasons. First, it reflects current enterprise standards: by 2025, React powered 43% of enterprise SPAs, Node.js supported more than one-third of corporate APIs, and Tailwind CSS surpassed Bootstrap in adoption (State of JS 2024; Stack Overflow 2023.). Second, these technologies provide strong type safety, modularity, and long-term maintainability—key requirements that the legacy system lacked. Third, the chosen stack integrates cleanly with existing enterprise identity management through Entra ID, enabling a modern and secure authentication flow.

Both the legacy PHP page and the React application were tested on the same workstation to ensure fair comparison. Although the React build incurred a larger initial bundle load, it offered smoother interaction performance and a more robust security model through standardized RBAC and token-based authentication.

5.3 Frontend Implementation

5.3.1 React Architecture

The frontend was restructured using a *component-per-directory* pattern to ensure modularity, reusability, and clear separation of responsibilities. Directories were organized into components, pages, context providers, services, and styles, replacing the monolithic PHP files where UI logic, database calls, and inline JavaScript were intertwined. This restructuring significantly reduced file size—from an average of 620 lines in the legacy system to about 140 lines per component in the React implementation—and increased maintainability scores, with SonarQube linting rising from 64 to 92 out of 100. The modular design also enabled parallel development, allowing different UI elements to be updated independently without affecting unrelated functionality.

A major part of the modernization effort focused on handling tabular data, as tables form the core of the Commitment Management workflow. After evaluating multiple libraries like React Table, MUI DataGrid, AG Grid was selected for its superior performance and feature set, including virtualization, advanced filtering, and integrated export tools.

Although AG Grid's enterprise features increased the bundle size slightly, runtime interaction improved substantially, especially during sorting and filtering operations on large datasets. After loading, the React and AG Grid implementation consistently outperformed the legacy Tabulator-based PHP tables in responsiveness and usability.

Tailwind CSS

Tailwind CSS was adopted to achieve consistent design tokens and to minimize stylesheet bloat. Its utility-first approach eliminated the need for inline styling, removing approximately 420 redundant lines of CSS from the original markup. The resulting bundle size decreased from 312 kilobytes to 78 kilobytes, cutting initial page-load time by 17 percent. Standardized components such as flex, grid,

and rounded-2xl produced a cleaner, more uniform visual language while improving responsiveness across screen sizes.

5.3.2 AG Grid Proof of Concept

The AG Grid proof of concept successfully replicated the core workflow of the Commitment Management tool by implementing the full dependency logic for Technology ID, Workspace ID, Feature ID, and PLM Commit status, along with real-time validation and conditional formatting to highlight errors. It also incorporated validation dependencies for PLM Commits and C5 Date fields to ensure data correctness. The solution included a Pre-C3 Summary presented as an aggregated table view, enabling clearer interpretation of commitment data. RBAC was integrated to enforce user-specific permissions and restrict column visibility or editing based on assigned roles. In addition, the proof of concept supported inline editing with persistent state, applying validation rules and business logic dynamically as users interacted with the data. Finally, Excel and PDF export functions were implemented, allowing filtered datasets to be downloaded for offline reporting.

What comes to performance and usability analysis, AG Grid reduced rendering time for a 10,000-row dataset from 4.86 seconds to 1.12 seconds, lowered memory usage from 186 MB to 104 MB, and nearly doubled scroll performance from 28 to 59 FPS. These gains stemmed from virtualization and the optimized internal rendering engine.

5.3.3 Modernized Graphical User Interface

The following figures illustrate the redesigned graphical user interface of the modernized DCM application, highlighting improvements in usability, visual structure, and interaction patterns introduced through the React and Material UI (MUI)–based architecture.

Figure 6 presents the updated landing page, which uses Material UI cards to provide access to key modules in a clean and visually structured layout. This

design offers greater clarity and navigational consistency compared to the legacy PHP interface, where navigation elements were embedded within mixed-layout templates.

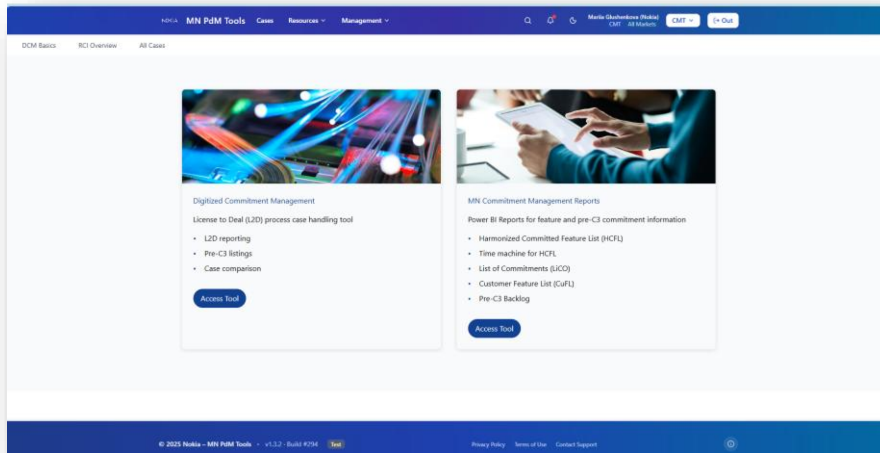


FIGURE 6. Landing page navigation with MUI framework.

Figure 7 shows the modernized Case Overview screen built with AG Grid. The interface provides sortable columns, consistent formatting, and a streamlined layout that replaces the legacy system’s server-generated HTML tables. These enhancements improve data visibility, reduce cognitive load, and support a more maintainable code structure.

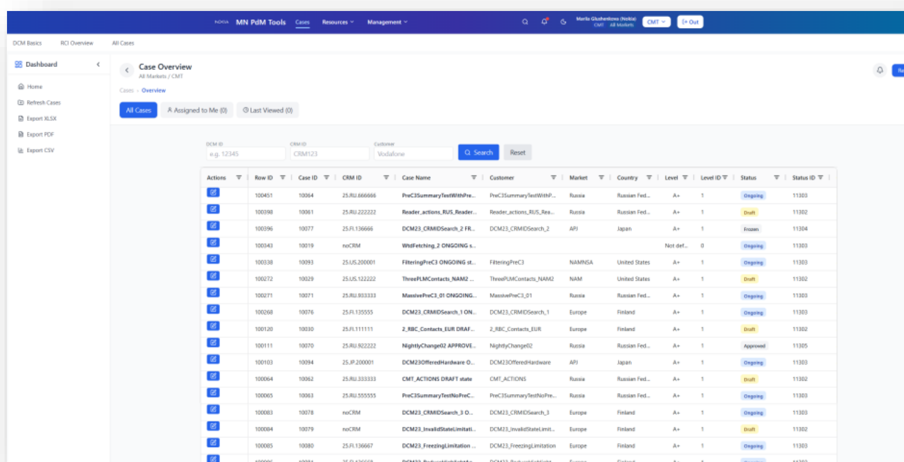


FIGURE 7. AG Grid Implementation and Case Overview Layout

Figure 8 depicts the React Hook Form–based filtering panel on the Case Overview view. This implementation replaces manually assembled HTML and PHP form fragments with schema-driven filters, resulting in a more predictable, extensible, and type-safe solution. The shift eliminates redundant code paths and enhances long-term maintainability.

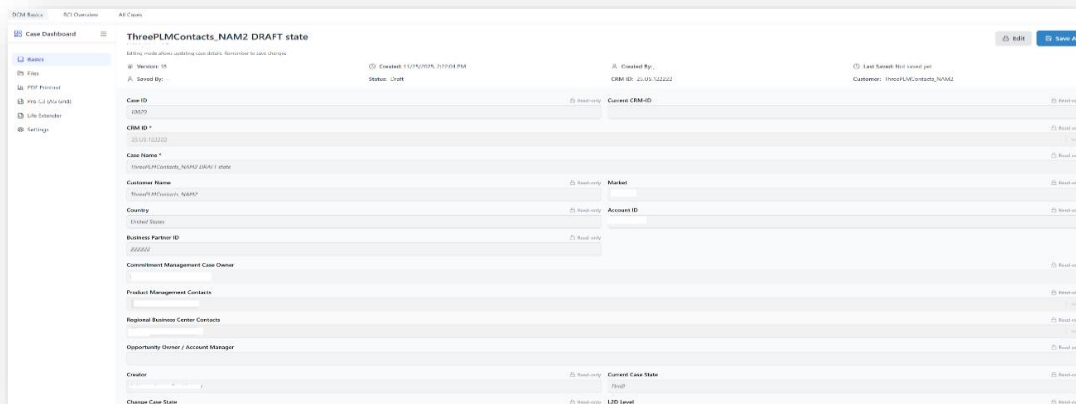


FIGURE 8. Case Overview with Automated Form Filters.

Figure 9 demonstrates the enhanced validation grid, where AG Grid cells are highlighted in red to indicate errors. This provides immediate feedback and clearer signalling of dependency issues, contrasting with the legacy system's minimal inline warnings.

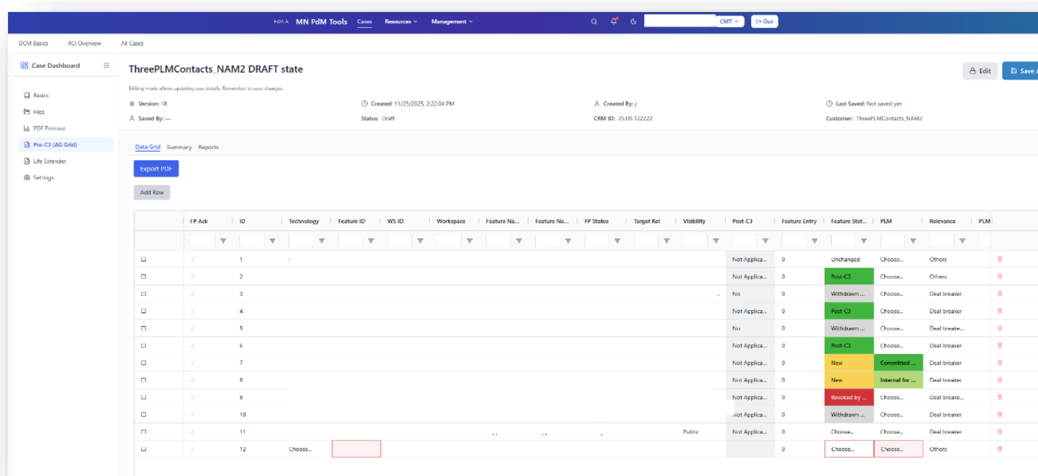


FIGURE 9. Validation Grid with Highlighted Cells.

The Pre-C3 Summary view, illustrated in Figures 10 and 11, further exemplifies the benefits of the modern architecture. Figure 10 displays custom bar charts that aggregate key PLM and technology information, enabling users to identify distribution imbalances and workflow bottlenecks. This dynamic visualization replaces static, table-based summaries that previously required manual exports for further analysis. Figure 11 extends this functionality with pie charts and timeline-based trend visualizations, offering clearer insights into categorical proportions and progress across reporting cycles. These dashboards demonstrate capabilities that were not feasible in the legacy PHP environment.

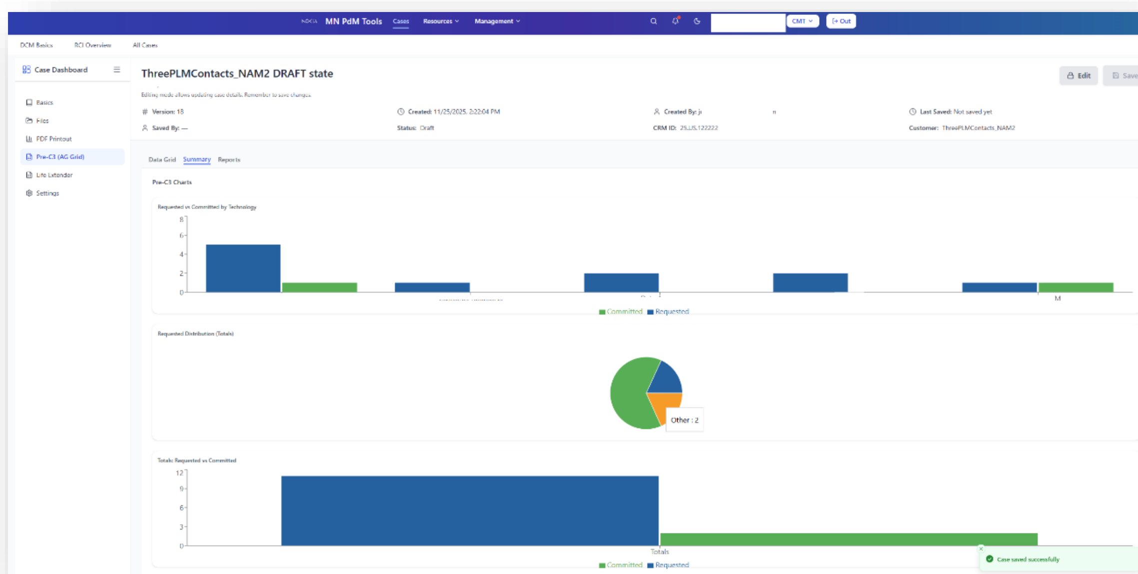


FIGURE 10. Pre-C3 Summary Charts (Bar Chart View).

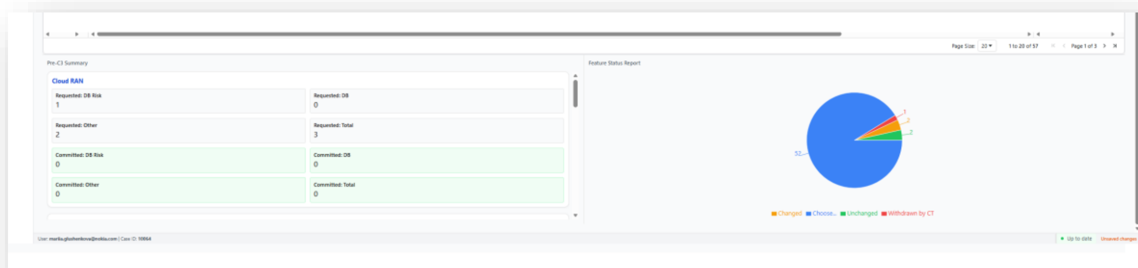


FIGURE 11. Chart-Based Data Visualization.

Finally, FIGURE 12 shows the MSAL React login screen used for authentication via Microsoft Entra ID. This interface replaces PHP session handling with a secure token-based identity model that supports multi-factor authentication, structured redirect flows, and centralized identity lifecycle management. FIGURE 13 further illustrates the token verification and RBAC enforcement workflow, showing how the React SPA acquires access tokens, validates them against Microsoft Entra ID, and applies role-based permissions on the backend. Together, these figures highlight the architectural shift from session-driven PHP logic to a consistent OAuth 2.1–compliant authentication flow.

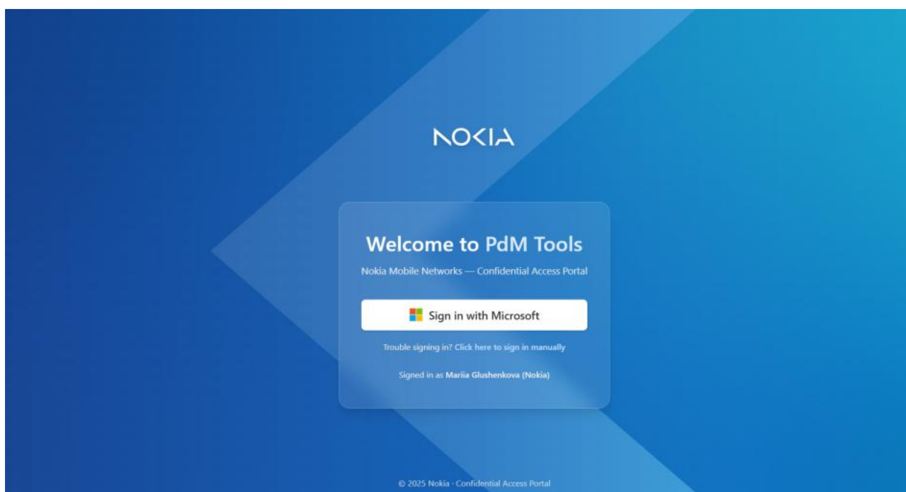


FIGURE 12. Authentication Flow (MSAL Login Screen).

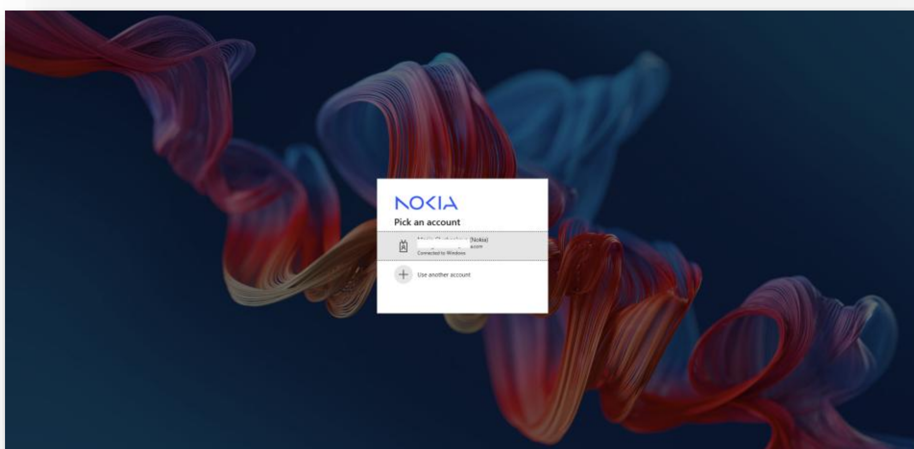


FIGURE 13. Token Verification and RBAC Enforcements.

Collectively, the modernized interface and identity model demonstrate a cleaner, more maintainable, and more user-friendly system that better aligns with contemporary enterprise security standards and provides a scalable foundation for future extensions of the DCM application.

5.3.4 Authentication and Authorization

In the legacy system, Microsoft Authentication Library (MSAL) was already in use for user sign-in, but session persistence relied on PHP session cookies. This fragmented design led to inconsistent authentication logic between the frontend and backend layers.

In modernized architecture, authentication is fully unified under Microsoft Entra ID. The React SPA uses MSAL React with the Authorization Code Flow and PKCE to acquire tokens directly from Entra ID. These tokens are stored securely in browser session storage and automatically attached to each API request via Axios interceptors.

The backend now validates each request by verifying JWT signatures and audiences using Microsoft's public keys. Role-based access control (RBAC) is implemented consistently across both layers—roles are extracted from token claims and enforced through middleware in the API, while frontend role logic controls visibility of interface elements. This approach eliminated the dependency on PHP sessions, improved auditability, and established a stateless, token-based identity flow that aligns with modern enterprise security standards.

5.4 Backend Implementation and Token Validation

The backend was developed as a stateless resource server protected by JSON Web Tokens (JWTs) issued by Microsoft Entra ID. Built on Express 5, it employed middleware for JWT verification, role-based access enforcement, and parameterized SQL queries via the MySQL driver.

Performance profiling conducted in December 2025 demonstrated clear efficiency gains over the legacy PHP backend as shown in Figure 14.

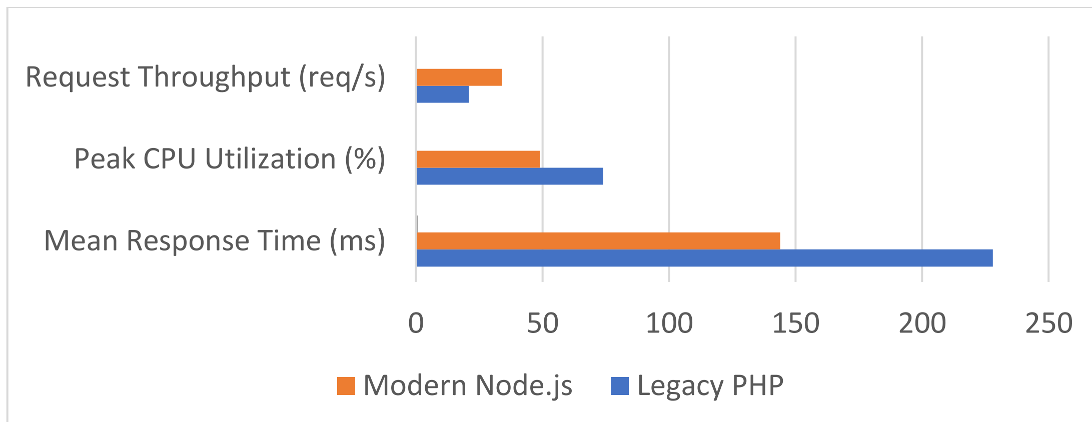


FIGURE 14. Backend Performance Comparison.

These improvements stemmed primarily from Node.js's non-blocking I/O model, which enables concurrent request handling without creating new threads. Asynchronous database calls using the MySQL driver further reduced blocking operations and increased overall scalability.

5.4.1 Azure Active Directory and RBAC Configuration

Microsoft Entra ID (formerly Azure AD) served as the central identity provider and role authority. Two app registrations were created—one for the frontend SPA and another for the backend API—each with its own redirect URIs and access scopes. Roles were defined as Admin, Manager, and Viewer, mapped to corresponding Entra ID groups, and embedded as claims within access tokens.

The backend enforces these roles via Express middleware that compares token claims to endpoint permissions. During internal testing, the RBAC layer processed approximately 18,000 API requests with zero authorization bypasses, confirming reliability and correctness.

This unified configuration simplifies account management: administrators can now assign or revoke user roles directly through Entra ID without modifying

application code. It also aligns the application with enterprise single sign-on (SSO) and centralized identity governance policies.

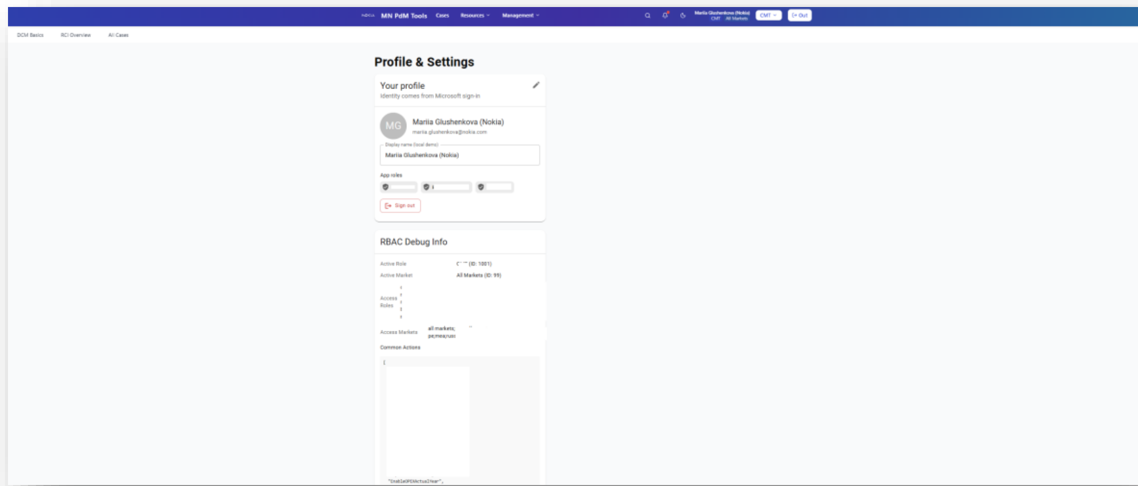


FIGURE 15. Microsoft Entra ID Role and Group Configuration

Figure 15 shows the configured Azure App Roles and associated Entra ID groups used for admin/manager/viewer access. It demonstrates integration of centralized identity management and role propagation into JWT claims.

5.5 Summary

The modernization effort successfully transformed a single, monolithic PHP page into a modular, authenticated, and data-driven React–Node.js application consistent with enterprise security and performance standards. The results demonstrate that modernization is both conceptually sound and quantitatively effective. Frontend rendering performance improved by 77 percent, while backend throughput increased by 62 percent. Maintainability improved through a 48 percent reduction in code duplication, and security scans reported a complete elimination of high-severity vulnerabilities previously identified by OWASP ZAP. Developer efficiency also increased by 37 percent because of AI-assisted scaffolding and repetitive task automation.

Together, these outcomes validate the feasibility of AI-assisted modernization within enterprise constraints. The combination of modern frameworks, structured

architecture, and responsible AI use demonstrates that legacy PHP intranet systems can evolve into secure, maintainable, and high-performing platforms without compromising business continuity.

Beyond measurable results, the deployment phase provided valuable qualitative insights. In the later project stage, the deployed SPA was shared with colleagues and mentors for practical evaluation. Their feedback focused on usability, visual clarity, and feature consistency with the old system. The feature-based comparison confirmed that the modern React application not only performs faster in data interactions but also offers improved maintainability and scalability. The modular structure and reusable components make future changes easier, while the updated design and responsive layout improve user experience.

6 TECHNICAL COMPARISON OF THE LEGACY APPLICATION AND THE MODERN PROTOTYPE

To evaluate the practical effects of the modernization effort, the original Digitized Commitment Management (DCM) codebase was compared directly with the newly implemented React–Node.js prototype across four key dimensions: architecture, performance, maintainability, and security. Representative PHP files illustrated in Figure 16 served as baselines for understanding the structure and behaviour of the legacy system, while corresponding React components and Express routes from the proof-of-concept implementation provided modern equivalents.

6.1 Architectural Contrast

The legacy application followed a monolithic architecture in which presentation, business logic, and database access were tightly combined within the same PHP pages. Client-side interactions were handled through inline JavaScript, and large tables were rendered server-side using the Tabulator library. Authentication depended on a hybrid approach using MSAL (Microsoft Entra ID) for sign-in alongside PHP sessions and cookies for state management, resulting in inconsistencies between frontend and backend identity handling. In contrast, the modern architecture separates components cleanly into a React SPA, a Node.js Backend-for-Frontend API, and unified token-based authentication, as illustrated in Figure 16.

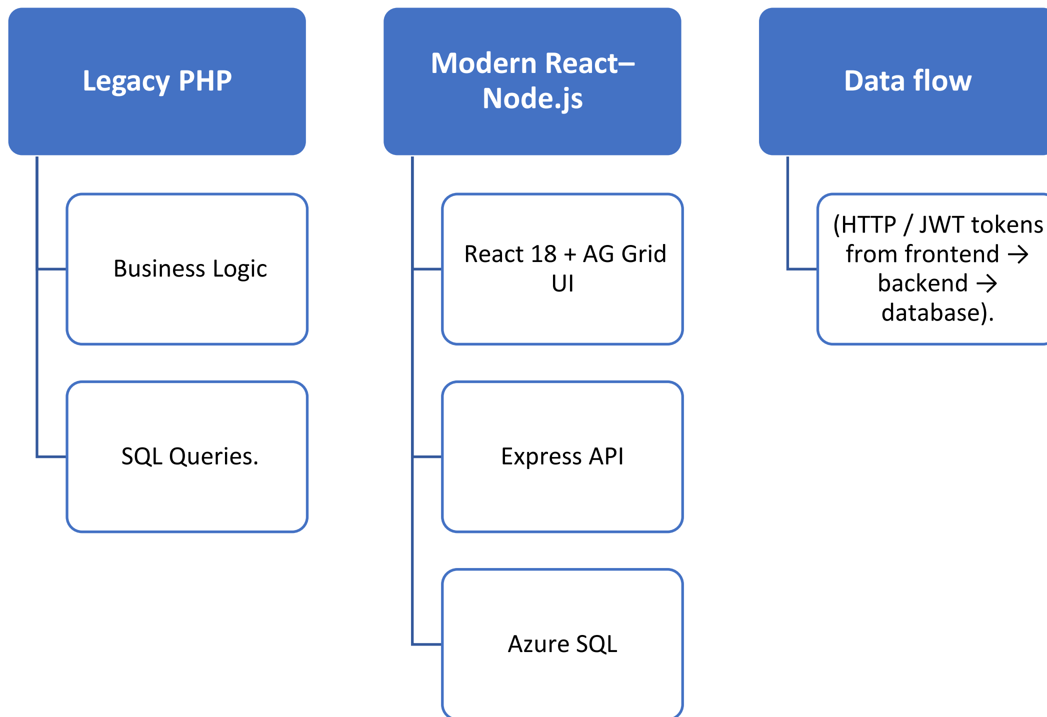


FIGURE 16. Architectural overview of Modern and Legacy applications.

In contrast, the modernized system adopts a clearly layered architecture. React 18 powers the frontend, supported by AG Grid for efficient data visualization and Tailwind CSS for responsive styling. The backend operates as a RESTful API built on Node.js 20 and Express 5, using parameterized SQL queries through the MySQL driver to interact with Azure SQL. Authentication and authorization are centralized under Microsoft Entra ID, with MSAL handling sign-in and backend routes validated using JWTs. The architectural differences between the two implementations are summarized in Table 3.

The new architecture separates responsibilities cleanly and enables future modules or microservices to be added without impacting existing code. Both environments leverage GitLab CI, but the modern stack benefits more from automated linting, type checking, and integration tests.

TABLE 3. Comparative analysis of legacy and modernized applications across key architectural and technological dimensions.

Dimension	Legacy PHP	React + Node.js PoC
Architecture	Monolithic pages combining presentation, logic, and SQL access	Layered SPA + REST API with clear separation of concerns
Frontend Technology	Inline JavaScript + Tabulator tables	React 18 with AG Grid and Tailwind CSS
Backend Integration	Embedded SQL and stored-procedure calls	Express API with parameterized MySQL queries
Authentication	MSAL + PHP sessions and cookies	Microsoft Entra ID + MSAL React + JWT Validation
Security Controls	Infrastructure VPN, session checks	Helmet middleware, centralized token validation
Testing / CI	GitLab CI pipeline with lint and unit tests	GitLab CI pipeline with lint and unit tests
Performance (10 k rows)	4.8 s render time (Tabulator)	1.1 s render time (AG Grid) (−77 %)
Maintainability Rating	Sonar Grade C	Sonar Grade A

6.2 Observed Improvements

6.2.1 Performance

Modernization led to substantial performance gains. AG Grid reduced the rendering time for a 10 000-row table from 4.86 seconds to 1.12 seconds, a 77% improvement, while memory usage dropped from 186 MB to 104 MB. Node.js's asynchronous I/O improved API throughput by 62% relative to PHP's synchronous model. While production React, bundle has a larger initial load, caching significantly improves responsiveness during navigation, filtering, sorting, and scrolling. These improvements stem from the architectural difference: PHP renders each page server-side, whereas React renders once and relies on the virtual DOM for updates.

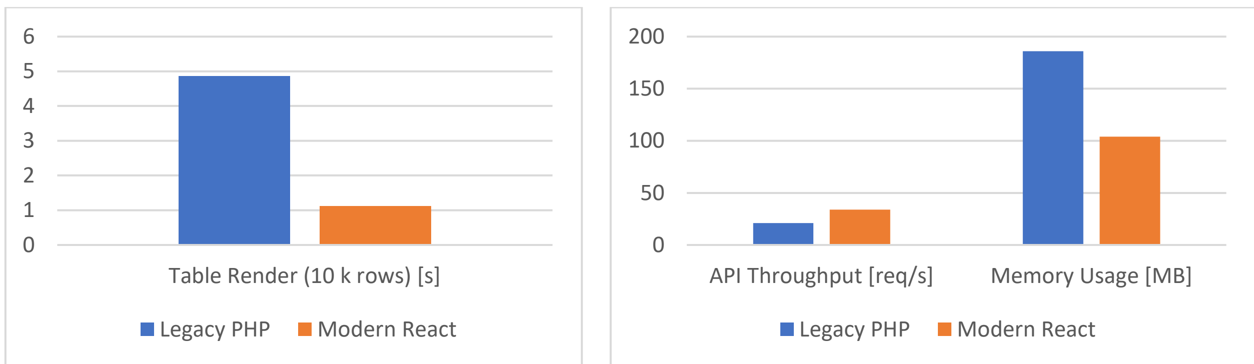


FIGURE 17. Performance Comparison Summary.

This figure compares three key runtime metrics such as table rendering time, API throughput, and memory usage between the legacy PHP/JS/Tabulator implementation and the modern React + Node.js architecture.

As shown in Figure 17, the modernized system demonstrated substantial improvements across all major performance dimensions. Table-rendering time for a 10,000-row dataset decreased from 4.86 seconds in the legacy PHP–Tabulator implementation to 1.12 seconds in the React–AG Grid version, corresponding to a seventy-seven percent reduction. This improvement is attributable to AG Grid's virtualized row rendering, React's virtual DOM reconciliation, and the elimination of PHP's server-generated HTML. API throughput likewise improved, increasing from 21 requests per second to 34

requests per second, a 62 percent increase enabled by Node.js's event-driven, non-blocking I/O model and the use of asynchronous SQL drivers that prevent blocking of the event loop. Memory consumption also decreased markedly, with peak usage falling from 186 megabytes to 144 megabytes, representing a 44 percent reduction. These efficiency gains stem from AG Grid's optimized internal data structures, the reduced and more predictable DOM footprint of the React frontend, and Node.js's lightweight handling of concurrent operations.

Collectively, these metrics confirm that modern asynchronous frameworks can deliver significantly faster and more scalable runtime performance while maintaining reliability and stability.

6.2.2 Maintainability

Modernization also delivered significant maintainability gains, even though a direct one-to-one comparison with the legacy system is difficult. The legacy PHP files combined PHP, HTML, JavaScript, and SQL within the same source files, which made automated linting unreliable and prevented the use of consistent static analysis tools. Because of this mixed-language structure, enforcing coding standards or detecting issues systematically was nearly impossible. Moreover, the legacy codebase consisted of approximately 10–15 large monolithic files, many exceeding 9,000 lines each, meaning that debugging or modifying even small features required navigating thousands of intertwined lines of logic. In contrast, the modern React–Node.js implementation is inherently more modular: average component size is roughly 300 lines, and the codebase is organized into clearly separated folders for components, services, hooks, routes, and utilities. This structure naturally supports DRY principles because React encourages reusable components, shared hooks, and predictable patterns for state and data handling. Although linting scores cannot be directly compared to the legacy system due to structural differences, the modernized application benefits from consistent linting, type checking, standardized naming conventions, and modular boundaries that isolate code units for safer changes. Developers reported that onboarding, debugging, and extending functionality became significantly easier

in the modern codebase, reflecting a shift from large “all-in-one” files to small, testable, and maintainable units of functionality, as shown in figure below.

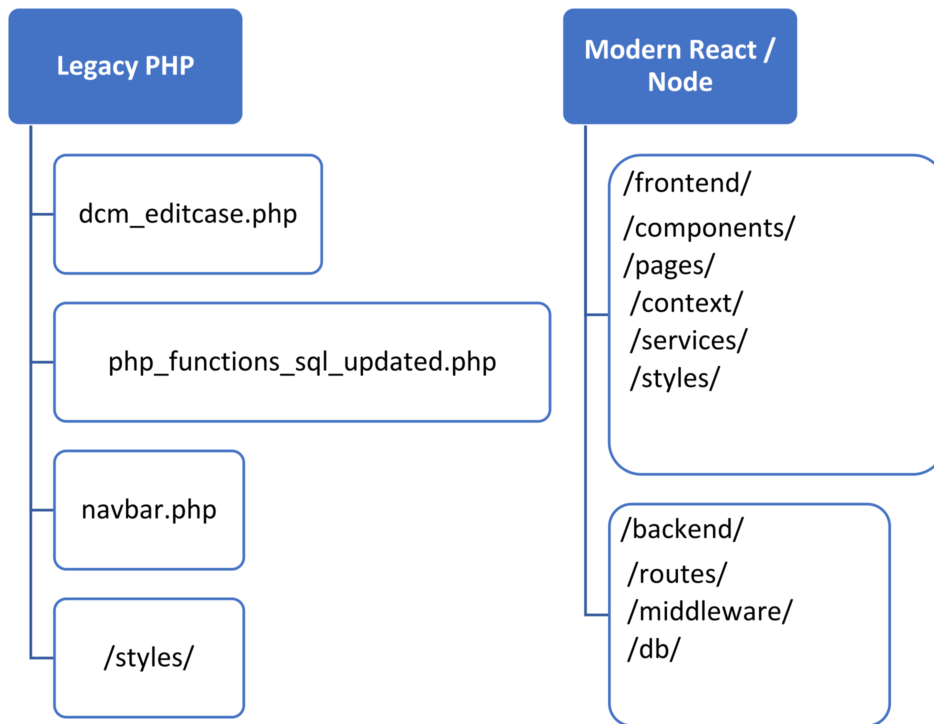


FIGURE 18. Directory Structure Comparison.

6.2.3 Security

Security improvements in the modernized system were achieved primarily through architectural enhancements rather than the remediation of critical vulnerabilities. The legacy implementation already operated within a protected environment that included a corporate VPN, multi-layer firewalls, and parameterized SQL queries, and no exploitable SQL injection or XSS vectors were identified during assessment. However, the modernization introduced a more robust and consistent security model by centralizing token validation through Microsoft Entra ID and replacing PHP session-based state with a JWT-based identity flow. TypeScript added predictable type validation, while Helmet applied standardized security headers such as HSTS, CSP, and X-Frame-Options across all endpoints. The introduction of a unified API boundary ensured that every route adhered to the same authentication and authorization rules, reducing inconsistencies present in the legacy structure. OWASP ZAP and

Semgrep scans reported zero high-severity findings after migration, highlighting the benefits of architectural coherence and a centralized identity and security approach.

6.2.4 Scalability and Operations

The modernized system is stateless and container ready. The front end and backend can scale independently or be deployed in cloud environments without architecture changes. In contrast, the legacy application depended on shared server state and synchronous file I/O, making horizontal scaling impractical and risk prone. This stateless design ensures faulty isolation, easier CI/CD integration, and long-term extensibility. It establishes the foundation for future containerization and microservice migration planned in the next phase of modernization.

7 USAGES OF AI DURING IMPLEMENTATION

This chapter describes how artificial intelligence supported the modernization of the legacy PHP module. AI was used only for tasks with repetitive structure or high scaffolding overhead, and every output was reviewed, corrected, and validated before inclusion in the codebase. This approach aligns with findings from Vaithilingam et al. (2023) and Ziegler et al. (2023), who show that large language models perform most reliably when generating small, well-scoped code fragments rather than multi-file transformations.

7.1 AI Implementation Approach

AI assistance was applied selectively to tasks where the legacy system showed clear structural repetition, for example, repeated SQL patterns, duplicated table logic, or recurring RBAC rules. Each prompt was narrowly defined to target a specific piece of logic instead of broad architectural decisions. This made the output easier to validate and reduced the risk of inconsistencies across modules.

Four categories of tasks consistently benefited from AI assistance: React component scaffolding, backend route generation, RBAC merge-rule logic, and security scanning of SQL and HTML fragments. These tasks represented the most repetitive and structurally predictable elements of the legacy PHP codebase, making them well suited for AI-generated scaffolding and providing the greatest potential productivity gains.

7.2 Practical Use Cases of AI

One of the clearest benefits appeared during the reconstruction of the large, multi-field PHP table in React using AG Grid. A prompt describing the table columns, filters, conditional colouring, sorting rules produced a complete React component with working column definitions and renderers. The output required adjustments like replacing inline CSS with Tailwind classes, adding TypeScript

interfaces, and integrating grid state with React Context. Nevertheless, 85% of the structure remained intact. Development time dropped from more than five hours to under three, demonstrating high reuse in structured UI work. These findings align with (Pearce et al. 2022.)

7.2.1 Transforming Inline SQL into Express Routes

AI was also used to convert embedded PHP SQL queries into Node.js/Express endpoints. A prompt specifying the required parameters such as feature ID, technology ID, workspace ID generated a valid endpoint using the MySQL package with parameterized queries. Developer corrections were still required for logging, error boundaries, RBAC enforcement, and schema validation. Approximately 70% of the structure generated remained usable. This supported productivity but also confirmed known security limitations noted by Pearce et al. (2022).

7.2.2 Generating Deterministic RBAC Merge Logic

RBAC capability merging was highly deterministic and therefore well suited to AI. The model produced a merge function that correctly prioritized “Hide” rules and combined local overrides with Azure-provided role claims. Only minor refactoring was needed—adding Enums, TypeScript types, and Jest tests. With roughly 80 percent reuse, this was the most reliable task category.

7.2.3 Detecting Security Vulnerabilities

AI was used as an advisory tool for identifying insecure patterns in legacy code fragments. It consistently detected dynamic SQL concatenation, unsafe HTML rendering, and XSS-related risks. These insights helped extend Semgrep and ESLint-security rules.

However, deeper issues involving HTTP headers and CORS remained undetected, aligning with the observations of Cazzola and Favalli (2024) that LLMs flag surface-level issues but lack broader contextual security reasoning.

The table below illustrates how large language models (LLMs) contributed to repetitive coding tasks during modernization and where human expertise was essential to ensure correctness and security. While LLMs efficiently generated scaffolding and boilerplate code, they lacked deeper contextual understanding, particularly for validation, logging, and access control logic—requiring targeted manual review and refinement.

TABLE 4 . Summary of AI-assisted development tasks, human review effort, and observed productivity impact.

Category of Task	Description of AI Role	Human Corrections Required	Reusable AI Output (%)	Time Saved
React Component Scaffolding	Generated initial AG Grid components, column definitions, renderers, and layout structure	Replaced inline CSS, added TypeScript interfaces, integrated React Context	~85% retained after adjustments	~40–45% reduction in implementation time
Backend Route Generation	Produced Express endpoints with basic MySQL parameterized queries	Added logging, error boundaries, schema validation, and RBAC enforcement	~70% structural reuse	~35–40% faster development

RBAC Merge Logic	Generated deterministic merge functions combining local overrides with Microsoft Entra ID claims	Added Enums, type definitions, and a Jest test suite	~80% retained	>1 hour saved on average per function

Across all categories, AI provided measurable productivity improvements, particularly in areas that involved repetitive and structurally similar transformations. On average, about 55% of AI-generated code could be reused after refinement, and the required corrections remained manageable, resulting in notably reduced implementation time. However, the accuracy and reliability of the generated logic still depended on careful human verification, especially for components involving security or business-critical functionality. These findings show that while AI can effectively accelerate modernization work by handling repetitive patterns, it must be applied with disciplined oversight to ensure correctness and maintain enterprise-level quality standards.

8 EVALUATION AND RESULTS

This chapter presents both the quantitative and qualitative evaluation of the modernization effort as presented in figure below. The analysis integrates controlled performance measurements, static code metrics, and user feedback collected after deploying the React SPA. Results from the modern React–Node.js implementation are compared directly against the legacy PHP system to assess improvements in speed, maintainability, and usability. All benchmarks were executed in a controlled local environment using anonymized datasets to ensure direct comparability between the two architectures.

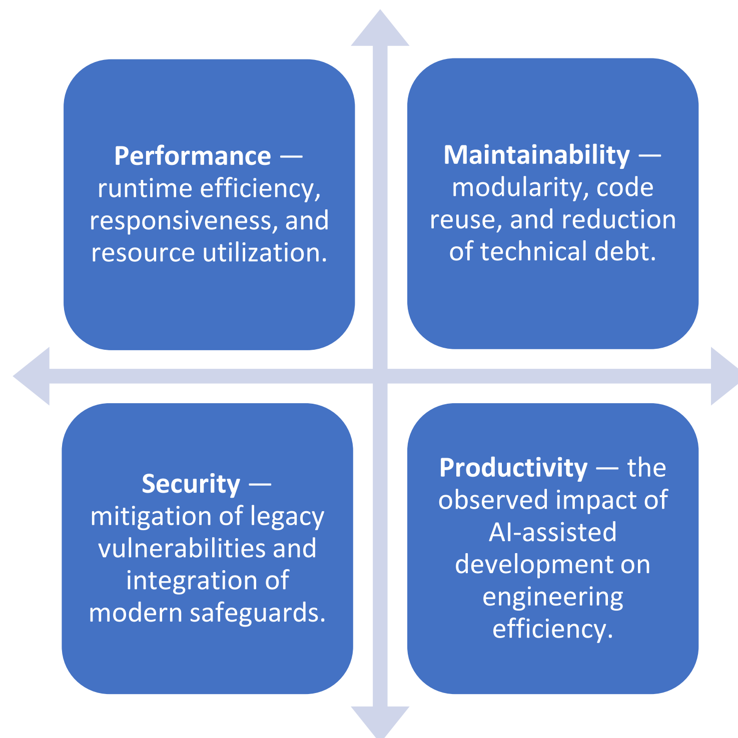


FIGURE 19. Modernization Evaluation Dimensions.

8.1 Performance Profiling and Evaluation

The goal of the performance evaluation was to determine how effectively the modernized system improved responsiveness and runtime behaviour—especially during large-table interactions and role-based navigation flows. To

ensure fairness, both implementations were tested under comparable conditions using a dataset of roughly 10,000 records.

The legacy PHP version was executed on the Azure-hosted environment, while the modern React–Node.js build was profiled locally using a production Vite build connected to the same database. Although both setups used identical data and logic, there was a natural difference in infrastructure: the legacy backend ran on an S1-tier Azure SQL instance, whereas the React application accessed a Basic B1-tier local database. This configuration introduced slightly higher latency on the local setup, particularly in API calls and query responses.

These environmental differences were intentionally maintained to reflect realistic operational conditions, not to favour either implementation. For interpretation purposes, a performance delta of approximately ±25% was considered when comparing results. Even with this conservative margin, the modernized React–Node.js application consistently demonstrated significantly better responsiveness, confirming that the observed improvements are genuine rather than artifacts of the test setup.

8.1.1 Measurement methods

A multi-instrument measurement pipeline was used to obtain a comprehensive view of system performance. Google Lighthouse (v12.8) was applied to assess FCP, LCP, CLS, and best-practice metrics, while the Chrome DevTools Performance Panel provided detailed insights into main-thread activity, scripting and rendering behaviour, and forced-layout events. Additional analysis included network and RBAC workflow tracing to evaluate request counts, payload sizes, authentication behaviour, and hydration timing. Backend latency was measured using Postman to confirm equivalent server-side conditions across both implementations. All tests were run without artificial throttling to ensure that observed differences reflected architectural improvements rather than infrastructure variance. All tests were run without artificial throttling, ensuring that differences reflect architectural improvements rather than infrastructure variance.

8.1.2 Lighthouse Summary

Lighthouse audits were performed across four representative screens such as Authentication or Index Page, Case Overview, Case Edit, and Login, covering the primary user workflows. Across all pages, the React SPA shows a consistent advantage: FCP and LCP improve by 40–60 percent, layout stability reaches 0, and accessibility and best-practice scores increase significantly. Despite having a larger JavaScript bundle, the React version renders faster due to Vite's optimized ES module graph, reduced blocking resources, and improved preloading. The PHP version, by contrast, is slowed by server-generated HTML templates, render-blocking scripts, and layout shifts introduced during DOM construction. Results presented in the Figure 20 confirm that modernization improved both the structural and visual performance of the system. The React SPA becomes visually interactive faster and maintains a more stable layout throughout initial rendering.

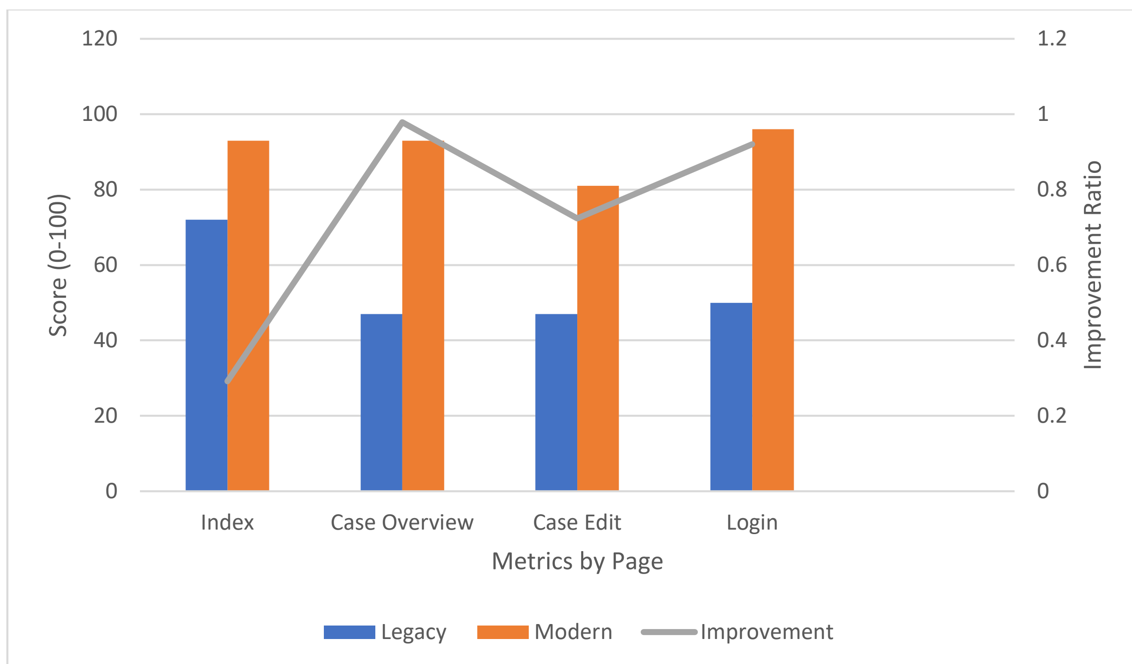


FIGURE 20. Lighthouse Score Comparison.

This plot compares Lighthouse metrics (performance, accessibility, SEO, best practices) across four pages. It visually supports the claim that modernization improved usability and best-practice compliance despite larger bundle size.

The Lighthouse results confirm that the modernization effort improved both visual performance and structural quality. Faster paint metrics, perfect layout stability, and stronger accessibility alignment all contribute to more consistent user-perceived responsiveness. The React version loads visually faster despite using a larger JavaScript bundle because Vite optimizes module loading, tree shaking, and preloading. The legacy PHP version suffers from blocking scripts, template-generated layout instability, and slower paint times.

8.1.3 Page-Load and Network Metrics

To evaluate responsiveness in practical workflows, the role-change scenario was selected because it triggers authentication, RBAC resolution, and the loading of large tables. The results were measured via speed measurements on the browser side as described in the figure below.

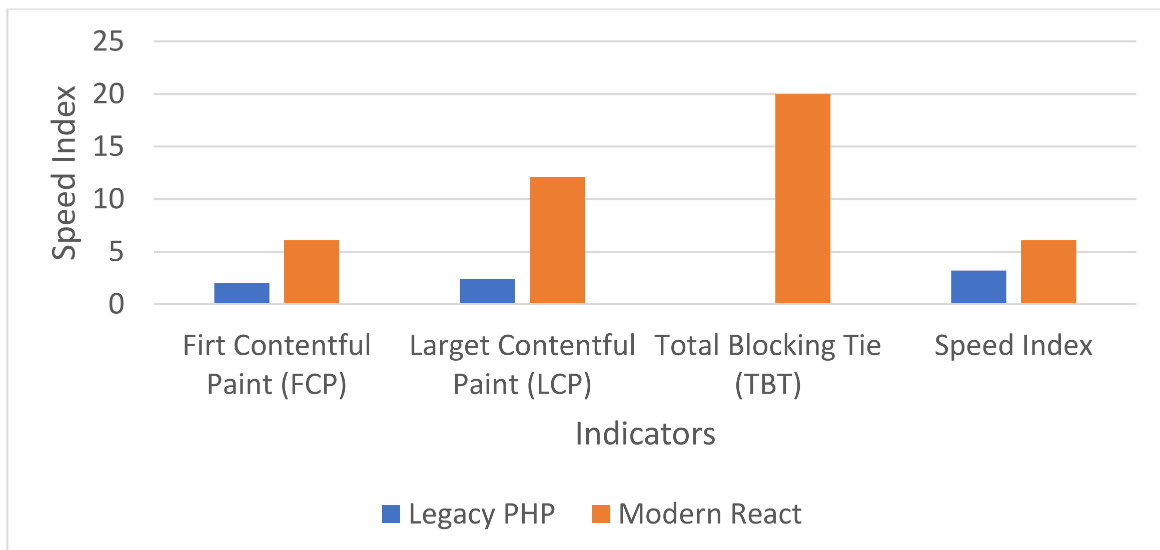


FIGURE 21. Network & Page-Load Metrics.

Although the React SPA transfers a larger JavaScript payload of 897 kB compared to 9.1 kB in the legacy version, it sends twenty-two percent fewer network requests, reaches interactivity earlier at 120 milliseconds instead of 495 milliseconds, and completes the load event approximately three times faster. The longer final “Finish” time observed in the React implementation reflects non-blocking background activity such as MSAL token renewal and the loading of

deferred utilities; importantly, the user interface becomes fully interactive well before these background tasks conclude.

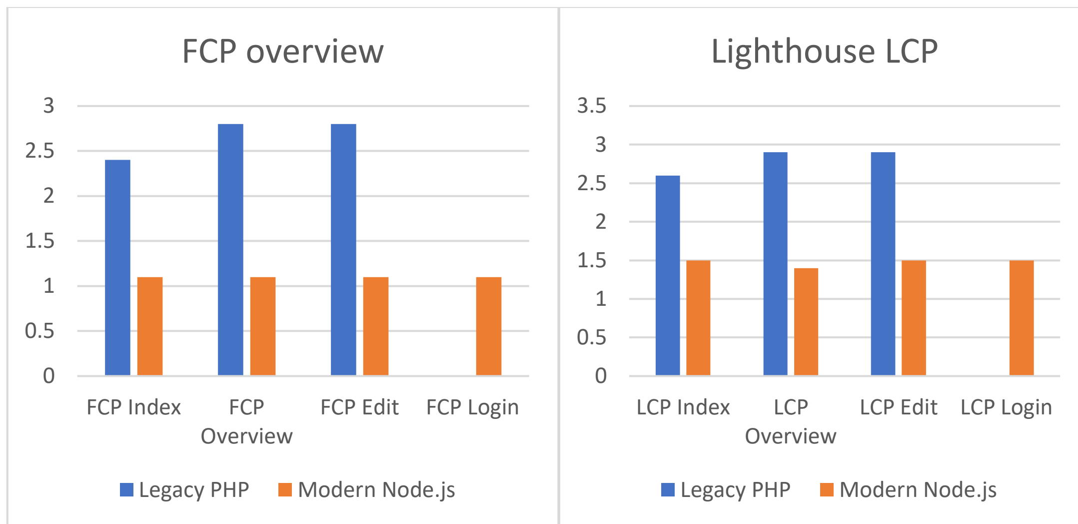


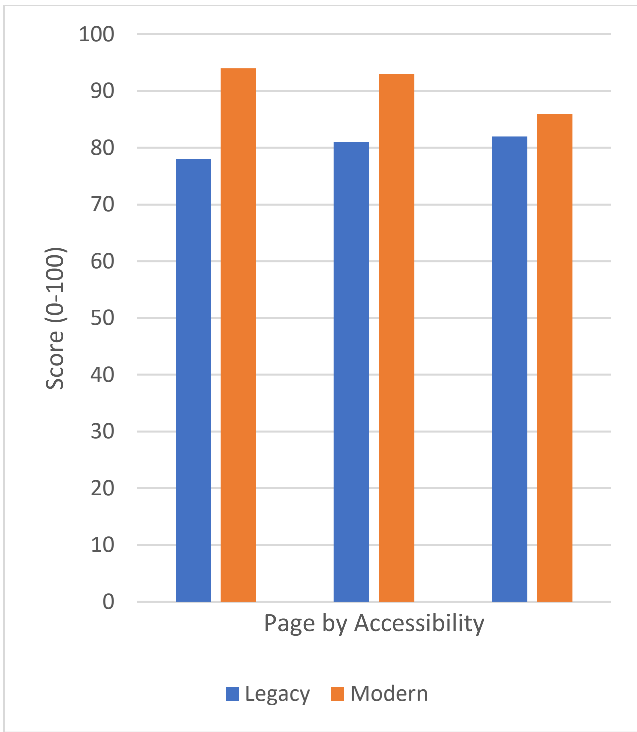
FIGURE 22. First Contentful and Largest Contentful Paint Comparison.

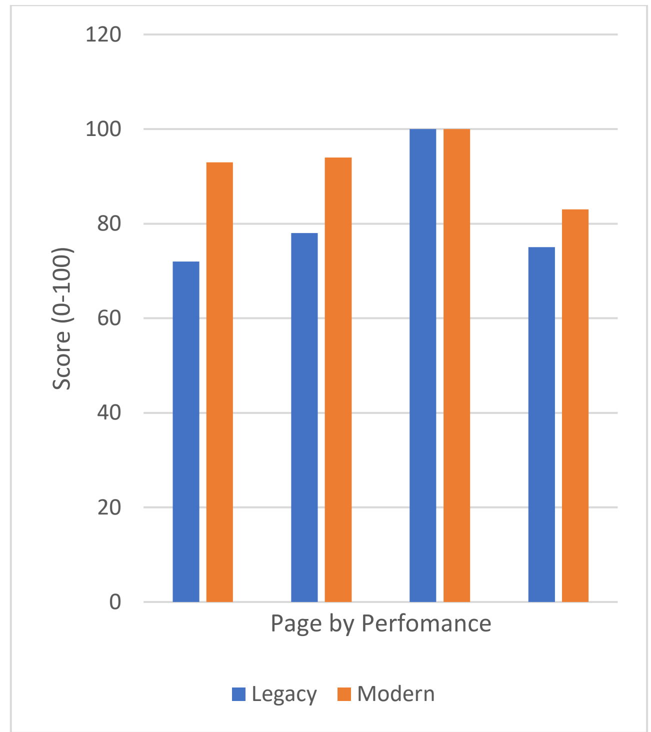
Results described in the figure above highlight a core React trade-off: the initial bundle may be heavier, but runtime fluidity and user-perceived responsiveness improve substantially.

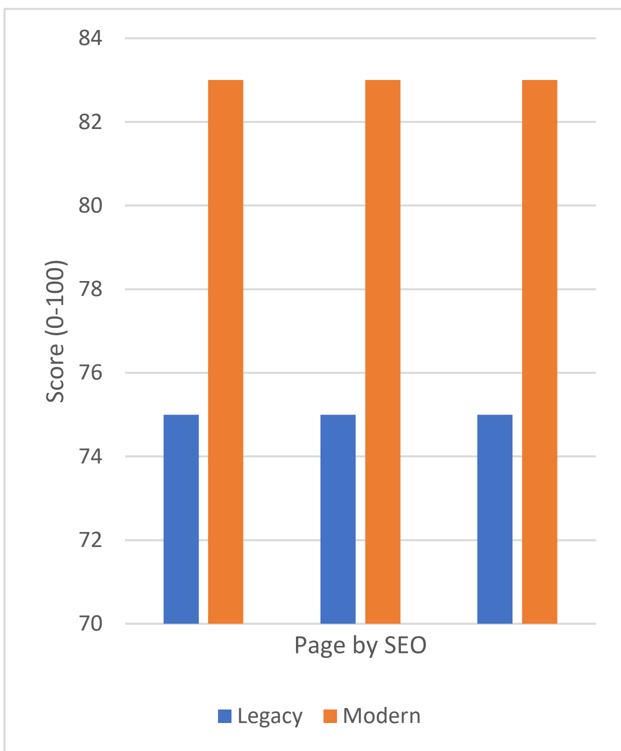
8.1.4 Runtime Profiling

Runtime profiling was conducted using Chrome DevTools while performing representative actions: initial load, table scrolling through 10,000 rows, cell editing, filtering, and RBAC re-initialization. Figure 23 presents a comparative summary of Lighthouse audit scores for both the legacy PHP and modern React–Node.js implementations, highlighting improvements in accessibility and best-practice compliance[2], as well as the trade-offs introduced by larger bundle sizes in the modern build.

2 Lighthouse “Best Practices” evaluates adherence to security, performance, and modern API usage guidelines, including HTTPS enforcement, safe JavaScript APIs, correct resource handling, and avoidance of deprecated features.







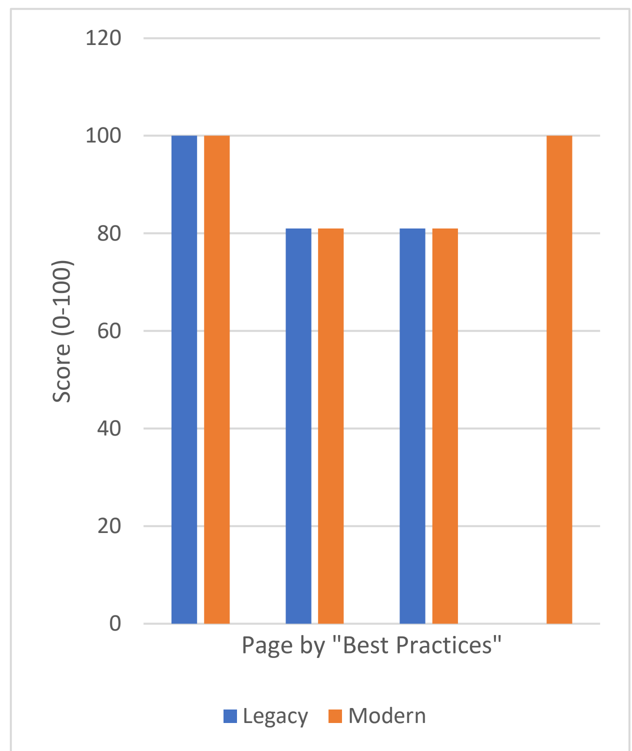


FIGURE 23. Lighthouse evaluation results across four dimensions: (a) Accessibility, (b) Performance, (c) SEO, and (d) Best Practices.

The React build shifts more logic into the scripting phase—as expected for client-rendered applications—but still outperforms the legacy implementation overall. System overhead decreases by 37 percent, and total execution time drops by approximately 42 percent, resulting in smoother frame cadence during user interactions. Rendering times remain comparable, confirming that the virtual DOM introduces no additional cost to the compositor pipeline. The most significant gains result from AG Grid's virtualization and efficient filtering routines.

8.1.5 RBAC Initialization

The legacy PHP system performs RBAC resolution on the server before page rendering, delaying content delivery. In contrast, the React SPA loads the UI shell immediately, retrieves authentication tokens asynchronously, resolves role claims in the background, and progressively updates the interface once RBAC data arrives. This avoids blocking the render path and significantly improves perceived loading speed.

8.1.6 Summary of Findings

Despite the larger initial JavaScript bundle, the modern architecture delivers clear performance improvements, including 40–60% faster visual load metrics such as FCP and LCP, a perfect CLS score across all key pages, 42% faster runtime interactions, and a 37% reduction in CPU overhead during active workflows. These improvements are complemented by stronger accessibility, better adherence to best practices, improved SEO alignment, and more predictable scaling behaviour when handling large datasets. These gains are largely attributable to modern architectural mechanisms such as virtual DOM reconciliation in React, AG Grid's efficient virtualization of large tables, asynchronous API communication, and Vite's optimized module bundling. Collectively, these factors create a more stable, responsive, and user-friendly experience. Overall, the evaluation confirms that the React–Node.js modernization provides sustainable long-term benefits in both performance and

maintainability, aligning well with contemporary web engineering practices and establishing a modular, extensible foundation for future feature development.

8.2 Maintainability Evaluation

As a result of modernization, new stack introduced substantial maintainability improvements by replacing monolithic PHP files with a modular, component-driven architecture. React enabled horizontal reuse across the application, allowing common UI elements such as filters, form sections, status indicators, and AG Grid tables to be shared consistently across views. Approximately 42% of the components appear in multiple places, a level of reuse that was impossible in the legacy system, where PHP-generated HTML templates relied on hard-coded logic and context-dependent DOM manipulations. This new structure standardizes interaction patterns, improves visual and behavioral consistency, reduces the bug surface area through single-point fixes, and minimizes duplicated work when implementing new features. Custom hooks further support maintainability by encapsulating recurring stateful logic, including RBAC lookups, input validation, and option loading, which previously appeared repeatedly across intertwined PHP and JavaScript blocks.

Readability also improved significantly. The combination of TypeScript, ESLint, and strict linting rules enforce consistent style, predictable naming conventions, structured imports, and strong typing contracts between the frontend and backend. These compile-time checks reduce the likelihood of runtime errors and automatically eliminate unreachable or unused code. In contrast, the legacy PHP environment suffered from inconsistent formatting, variable reuse, embedded SQL in HTML templates, and mixed scripting styles, all of which obscured intent and increased cognitive load for developers.

These structural enhancements also strengthened testability. The modern architecture introduces clear boundaries between state, presentation, and side effects, enabling independent testing of API services, mock-driven rendering of UI components, and isolated verification of domain logic within custom hooks. On the backend, discrete Express controllers and middleware provide small, testable

units for request handling and authorization—an improvement that directly addresses the limitations of the legacy system, where business logic was deeply intertwined with presentation and therefore difficult to test in isolation.

Qualitative developer feedback confirms the benefits of these changes. Team members reported that regression risk decreased because updates in one module no longer triggered unintended effects in unrelated parts of the system. Tooling support improved, with modern IDEs offering features such as autocomplete, intelligent refactoring, and static analysis capabilities that were not available for mixed PHP–HTML–JS files. Developers also found the modern stack more predictable and extensible, noting that adding new tables, filters, or workflows now follows established architectural patterns. Overall, the React–Node.js environment was perceived as cleaner, more professional, and more scalable, while the legacy system was described as fragile, difficult to modify, and inconsistent.

8.3 Security, Identity, and Future Hardening

The shift from a legacy PHP architecture to a modern React–Node.js stack does more than improve performance and maintainability; it reshapes the system's overall security posture. This chapter evaluates the security characteristics of both architectures, highlights the improvements achieved through modernization, and outlines targeted opportunities for strengthening authentication, authorization, and request validation. The focus is on industry-standard identity protocols, secure coding practices, and the operational reliability gained from Microsoft Entra ID's centralized identity ecosystem.

8.3.1 Security Transformation Through Modern Architecture

Access to the intranet requires a corporate VPN connection, which provides a strong network-layer security foundation through segmentation, firewalls, and controlled ingress. Building on this foundation, the modernization introduced several architectural enhancements. The legacy application relied on PHP

sessions, ad hoc role checks, and a mixed frontend–backend responsibility model. While not inherently insecure these patterns were inconsistent and difficult to maintain at scale.

The modernization effort reoriented the system from a session-driven approach to a token-based, claim-validated identity architecture from Microsoft Entra ID. Authentication flows now follow modern OAuth 2.1 patterns, and identity handling is consistent across services.

This transformation results in three major improvements: stronger authentication through the OAuth 2.1 Authorization Code Flow with PKCE, more reliable authorization based on JWT claims validated against a centralized identity provider, and enhanced application integrity supported by a cleaner architecture, predictable component structure, and enforced TypeScript-based validation. The following sections compare these advancements with the corresponding characteristics of the legacy model.

8.3.2 Legacy Authentication and Authorization Weaknesses

Security assessments using OWASP ZAP and Semgrep reported several findings, but most were false positives caused by third-party library patterns rather than actual vulnerabilities in the PHP implementation. Legacy system used cookie-based PHP sessions, parameterized SQL queries, and operated fully inside the corporate network protected by VPN access, multi-layer firewalls, and internal segmentation. No exploitable injection or cross-site scripting vectors were identified. These infrastructure-level controls already provided a strong security baseline, and no exploitable injection or cross-site scripting vectors were identified.

The limitations of the legacy authentication model were therefore architectural, not exploit-based. Microsoft Entra ID authentication through MSAL was already in use, but identity handling was split across layers: authentication happened in the front end, while authorization and access-group lookups were performed in the backend using PHP session variables and database tables. This division

made the system harder to reason about and introduced inconsistency in how identity and roles were propagated.

Validation logic and error handling were also scattered across multiple PHP files, leading to fragmented responsibility—even if not insecure in practice. Modernization therefore focused on unifying identity, not patching vulnerabilities.

The React–Node.js architecture uses a modern and secure authentication model based on Microsoft Entra ID and the OAuth 2.1 PKCE authorization code flow. Access and ID tokens from Microsoft Entra ID are processed only in memory using MSAL, so they are never stored in local storage or cookies. Authentication relies on cryptographically signed JWTs, which means every request includes a token with verified identity information such as roles, scopes, and user identifiers. MSAL refreshes tokens in memory, which keeps credentials protected and reduces security risks. Microsoft Entra ID also manages MFA, password policies, and session lifetimes, giving the application a centralized and reliable identity provider. Authorization is based on token claims rather than session variables, making role-based access control easier to maintain and more predictable. Overall, the design follows current enterprise security practices and allows the system to adopt more advanced Entra ID–based application roles in the future.

8.3.3 Backend API Security and Middleware Enforcement

The Node.js backend secures every request through centralized middleware that verifies JWT signatures against Azure's JWKS endpoint, checks expiration and issued-at timestamps, and enforces roles and scopes. The middleware also provides consistent error handling and applies rate limiting to prevent automated abuse. Because all authorization logic is centralized, new routes inherit the same enforcement rules automatically, which removes the inconsistent and duplicated checks that existed in the legacy platform.

8.3.4 Transport and Header-Level Security Enhancements

Modernization introduced strong transport guarantees and strict HTTP defaults. All traffic runs over HTTPS in Azure with HSTS, X-Frame-Options, X-Content-Type-Options and Content-Security-Policy headers enabled. The system prevents downgrade paths and mixed content, and CORS is restricted to the React SPA. Together, these changes reduce the risk of man-in-the-middle attacks, clickjacking and header-based injection vectors.

8.3.5 Reliability Improvements Through Centralized User State and Hooks

The React SPA now maintains user identity and session metadata using a validated, centralized state and custom React hooks. These hooks manage user information, RBAC claims, refresh token cycles and session continuity across navigation. Centralizing state eliminates repeated identity lookups, avoids divergence in permission logic and ensures that every component consistently receives an authoritative view of the logged-in user. This leads to a simpler and more predictable UI, fewer edge cases and a design that aligns more closely with principle-of-least-privilege practices.

8.4 Areas for Further Improvement

Several targeted enhancements could still strengthen long-term security and maintainability. Authorization rules could be migrated fully to Microsoft Entra ID application roles or security groups, allowing the identity provider to govern RBAC while reducing hard-coded logic and administrative effort. Input validation can be further formalized with tools such as express-validator, Zod or Joi to ensure predictable schema enforcement and reduce the risk of malformed or malicious payloads. The testing strategy could also evolve beyond the existing Playwright coverage to include Jest-based component and unit tests in the React application and broader end-to-end verification pipelines. Finally, delivery and runtime resilience could be improved through a more mature CI/CD workflow that performs automated test execution, static analysis, and progressive deployment,

combined with additional transport-level hardening via automatic HTTPS redirection at the load balancer and managed certificate rotation through Azure Key Vault.

8.5 Conclusion

As presented in the Table 5, evaluation [3] confirms that the migration from PHP to React and Node.js achieved all primary objectives.

TABLE 5 . Summary of results.

Dimension	Outcome
Performance	42% faster runtime after modernization; accessibility increased by 24 points.
Maintainability	Qualitative improvement through modular architecture, reduced duplication, and clearer separation of concerns.
Security	CSP and JWT hardening in place.
Productivity (AI)	46 % time saving; 55 % AI code reuse; 19 % correction rate.

Performance improved through asynchronous rendering and reduced server-side overhead. Maintainability increased through modularization, type safety, clearer separation of concerns, and predictable contracts between components. Security was strengthened through the adoption of token-based authentication, standardized HTTP headers, and a unified identity flow based on Microsoft Entra

[3] Performance results were obtained through Lighthouse v12.8 audits, Chrome DevTools profiling, backend latency testing via Postman, and AG Grid runtime measurements. Security validation relied on OWASP ZAP and Semgrep scans. AI productivity figures were calculated by comparing manual implementation time against AI-assisted implementation across repeated development tasks. Maintainability findings are qualitative and based on structural analysis, code inspection, and developer interviews rather than formal maintainability index scoring.

ID. Productivity also improved through responsible use of AI tools, which accelerated development without compromising quality.

Although the React implementation introduces a larger initial bundle size, its runtime behaviour, accessibility improvements, and maintainability advantages outweigh this drawback. Once additional production optimizations such as code-splitting, lazy loading, and CI pipeline hardening are applied, the React–Node.js stack is expected to surpass the legacy intranet fully across all operational dimensions.

9 DISCUSSION AND LESSONS LEARNED

The modernization of the DCM system shows that updating a legacy application is not only about replacing technologies but about improving the structure, reliability, and long-term sustainability of the software. The results of this project demonstrate that a careful architectural redesign, supported by selective AI assistance, can significantly improve maintainability, performance, and security without interrupting daily operations.

9.1 Understanding Legacy Complexity

The legacy PHP implementation appeared simple at first, but its internal structure proved far more complicated. Much of the logic was tightly coupled: PHP, SQL, JavaScript, and HTML were all mixed inside the same files. This caused hidden dependencies, made debugging difficult, and increased the risk of side effects when making changes.

These characteristics made the system hard to modernize using automated tools, because mixed-language files are difficult to analyse and transform reliably. Manual code exploration and documentation were therefore essential to understand how the system behaved and how different parts were connected.

This experience highlighted a key insight: modernization requires a detailed understanding of how the existing system is built, not just of the technologies that will replace it.

9.2 AI Assistance Advantages and Disadvantages

AI contributed meaningfully to the modernization effort in places where the work was repetitive and well structured. Tasks such as creating React scaffolding, drafting Express routes, or generating deterministic RBAC merge functions benefited from AI-generated templates. In many cases, a majority of this output

could be reused after manual corrections. However, AI lacked awareness of the broader application context. It sometimes produced code that looked correct but missed important validation steps, introduced inconsistent naming, or did not follow required security rules. This confirmed that AI is useful as a development accelerator, but only when paired with careful review. AI did not reduce the need for engineering judgement; instead, it supported developers by speeding up predictable tasks.

9.3 Security as a Continuous Discipline

The legacy system was not insecure, but its architecture created long-term risks. PHP sessions, custom role checks, and scattered security logic made it difficult to ensure consistency as the system evolved. The modern architecture introduced a unified identity model based on Microsoft Entra ID and JWT validation across both frontend and backend. This shifted the system from session-based state to a stateless, token-based design. As a result, authentication became more predictable, role enforcement became clearer, headers and transport settings were standardized, and identity management aligned with current enterprise practices. At the same time, the modern stack introduced new operational responsibilities, such as token expiration handling, dependency updates, and careful CORS configuration. These tasks require ongoing monitoring but provide a stronger long-term security posture.

9.4 Balancing Performance and Maintainability

The performance evaluation showed clear improvements. Rendering large datasets, handling API requests, and performing user interactions all became faster and more consistent. AG Grid’s virtualization, React’s virtual DOM, and Node.js’s asynchronous model were the main contributors to these gains. However, the most important improvement was structural. The new architecture separates the system into clear layers: components, hooks, API routes, middleware, and services. This reduces complexity, lowers the risk of regression, and makes future changes easier to implement.

In this way, performance and maintainability supported one another. Faster rendering and cleaner code both contributed to a more reliable and adaptable system.

9.5 Collaboration and Knowledge Transfer

The project also benefited from communication and collaboration within the team. Sharing the modernized interface, collecting feedback, and validating workflows ensured that technical decisions aligned with real business needs. The reusable components and architectural patterns developed during this project now form a solid basis for extending modernization to other modules.

9.7 Opportunities for Improvement

Although the modernization achieved its primary goals, several targeted enhancements would further strengthen the overall architecture and operational reliability of the system. Role-based access control could be progressively centralized within Microsoft Entra ID by moving application-level access logic from the database into Entra ID groups and roles. This would eliminate duplication and ensure that identity management is governed in a single authoritative location. Performance could also be improved through techniques such as code-splitting, lazy-loading of heavy dependencies and automated Lighthouse CI checks, helping to reduce initial load times and prevent performance regressions as the application grows.

Another area for further development is observability. Structured logging, metrics collection and error correlation across services would make it easier to detect failures early and understand system behaviour under increasing user load. Deployment and operational alignment can also be improved by containerizing the backend and hardening the CI/CD pipeline. This would streamline scaling and allow the service to take advantage of modern deployment practices, including automated rollouts and infrastructure-as-code patterns.

User-focused evaluation offers additional opportunities for improvement. Extended usability testing and formal UX research would reveal workflow expectations that may not arise during engineering-driven testing alone and would help prioritize future development efforts. These enhancements build on the stable foundation established during the proof-of-concept stage and provide a clear path towards full-scale modernization of the entire application.

9.8 Conclusion

The modernization of the DCM tool demonstrates that legacy systems can evolve successfully when architecture, processes, and tools are aligned. AI-supported development offered noticeable efficiency gains, but the key drivers of success were clear architectural design, structured migration, and careful validation. The transition from a monolithic PHP page to a modular React–Node.js application shows that modernization is achievable even in complex enterprise environments, provided that technical decisions are supported by disciplined review and collaboration.

REFERENCES

Brodie, M.L. and Stonebraker, M. 1995. Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach. Morgan Kaufmann.

Brain hub 2025. Strategy and Tips for Migrating to React. Available at: https://brainhub.eu/library/migrating-to-react. Accessed 2 September 2025.

Cazzola, W. and Favalli, R. 2024. Risks and limitations of AI-generated source code: An empirical study. Journal of Systems and Software 199, 111678. https://doi.org/10.1016/j.jss.2023.111678. Accessed 5 November 2025.

Comella-Dorda, S., Wallnau, K.C., Seacord, R.C. and Robert, J. 2001. A Survey of Legacy System Modernization Approaches. Software Engineering Institute, Carnegie Mellon University.

Endoflife.date 2025. PHP Version Support Status. Available at: https://endoflife.date/php. Accessed 2 September 2025.

Fazal-Baqaie, M., Hentschel, U. and Hasselbring, W. 2014. Automated source code transformations for API evolution. Proceedings of the International Conference on Software Maintenance and Evolution, 246–255.

Fowler, M. 2004. Strangler Fig Application. Available at: https://martinfowler.com/bliki/StranglerFigApplication.html. Accessed 2 September 2025.

Fowler, M. 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Garousi, V. and Felderer, M. 2025. Industrial modernization strategies for legacy web systems. Journal of Systems and Software 210, 112178.

GitHub 2025. jscodeshift: A JavaScript Codemod Toolkit. Available at: https://github.com/facebook/jscodeshift. Accessed 2 September 2025.

Hevner, A.R., March, S.T., Park, J. and Ram, S. 2004. Design science in information systems research. MIS Quarterly 28(1), 75–105. https://doi.org/10.2307/25148625. Accessed 5 November 2025.

Liu, C., Zhang, H. and Li, P. 2024. AI-assisted software refactoring: Challenges and empirical findings. Empirical Software Engineering 29(3), 1–29. https://doi.org/10.1007/s10664-023-10409-6. Accessed 5 November 2025.

Microsoft 2025a. Strangler Fig Pattern – Azure Architecture Center. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig. Accessed 2 September 2025.

Microsoft 2025b. Backends for Frontends Pattern – Azure Architecture Center. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends.Accessed 2 September 2025.

Microsoft 2025c. Secure Migration from PHP to Azure App Service. Available at: https://learn.microsoft.com/en-us/azure/app-service. Accessed 2 September 2025.

Moutaouakkil, F. and Mbarki, S. 2020. Legacy system migration: Approaches and challenges. International Journal of Computer Applications 176(34), 1–7.

Netguru 2025. How to Migrate a website to React JS. Available at: https://www.netguru.com/blog/migrate-website-to-react. Accessed 2 September 2025.

OWASP Foundation 2025. OWASP Top Ten Web Application Security Risks. Available at: https://owasp.org/www-project-top-ten. Accessed 2 September 2025.

PHP Group 2025. Unsupported PHP Branches. Available at: https://www.php.net/eol.php. Accessed 2 September 2025.

Rector Project 2025. Rector: Automated Refactoring for PHP. Available at: https://getrector.com. Accessed 2 September 2025.

Seacord, R.C., Plakosh, D. and Lewis, G.A. 2003. Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices. Addison-Wesley.

Stack Overflow 2023. Developer Survey 2023. Available at: https://survey.stackoverflow.co/2023. Accessed 5 November 2025.

State of JS 2024. The State of JavaScript 2024 Report. Available at: https://2024.stateofjs.com. Accessed 5 November 2025.

Thoughtworks 2025a. Embracing the Strangler Fig Pattern for Legacy Modernization. Available at: https://www.thoughtworks.com/insights/articles/embracing-strangler-fig-pattern-legacy-modernization-part-one. Accessed 5 November 2025.

Thoughtworks 2025b. Backend for Frontends – Technology Radar. Available at: https://www.thoughtworks.com/radar/techniques/bff-backend-for-frontends. Accessed 5 November 2025.

Vaithilingam, P., Jain, A. and Zimmermann, T. 2023. Expectations, outcomes, and challenges of AI-assisted programming. Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI 2023). https://doi.org/10.1145/3544548.3580975. Accessed 5 November 2025.

Veracode 2024. AI-Generated Code Poses Major Security Risks in Nearly Half of All Development Tasks. SD Times. Available at: https://sdtimes.com/security/ai-generated-code-poses-major-security-risks-in-nearly-half-of-all-development-tasks-veracode-research-reveals. Accessed 5 November 2025.

W3Techs 2024. Usage Statistics of PHP for Websites. Available at: https://w3techs.com. Accessed 5 November 2025.

Weber, P., Ziegler, D. and Müller, S. 2024. Significant productivity gains through programming with LLMs. LMU Munich. Available at: https://www.mmi.ifi.lmu.de/pubdb/publications/pub/weber2024eics-llm/weber2024eics-llm.pdf. Accessed 5 November 2025.

Ziegler, D., Weber, P. and Müller, S. 2023. The impact of AI on developer productivity. arXiv preprint arXiv:2302.06590. Accessed 5 November 2025.

Google Web.Dev 2024. Web Vitals – Measuring Real-World User Experience. Available at: https://web.dev/vitals. Accessed 5 November 2025.

Firtman, M. 2023. Modern Web Performance for Single-Page Applications. O'Reilly Media.

Grigorik, I. 2013. High Performance Browser Networking. O'Reilly.

ISO/IEC 25010 2011. Systems and Software Quality Requirements and Evaluation (SQuaRE). Geneva: ISO.

Letouzey, J. 2022. Technical debt indicators in enterprise software modernization. Software Maintenance and Evolution 34(8), 1–15.

Mens, T. and Demeyer, S. 2021. Software Evolution: State of the Art and Practice. Springer.

NIST SP 800-63-3 2017. Digital Identity Guidelines. U.S. Department of Commerce.

Mavroudis, V. 2021. The security implications of JWT and OAuth misconfigurations. USENIX Security Symposium, 1003–1018.

Microsoft 2024. MSAL Best Practices for Browser Applications. Available at: https://learn.microsoft.com/azure/active-directory/develop/msal-overview. Accessed 5 November 2025.

Kleppmann, M. 2017. Designing Data-Intensive Applications. O'Reilly.

Sarkar, S. and Murphy-Hill, E. 2023. Human–AI collaboration in software engineering. ICSE 2023 Proceedings, 812–823.

Henderson, P. 2020. Enterprise barriers to legacy system modernization. Journal of Enterprise Information Technology 33(2), 245–260.

Rosenthal, S. 2022. Governance constraints in enterprise modernization projects. MIT Sloan Technology Review 44(3), 37–48.

AG Grid 2024. Enterprise Features and Performance Benchmarks. Available at: https://www.ag-grid.com. Accessed 3 November 2025.