

Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing



A Monthly Journal of Computer Science and Information Technology

ISSN 2320-088X

IMPACT FACTOR: 7.056

IJCSMC, Vol. 14, Issue. 9, September 2025, pg.80 – 91

A COMPARATIVE ANALYSIS OF LLM-DRIVEN VS. MANUAL LEGACY CODE REFACTORING: A CASE STUDY IN .NET CORE MIGRATION

Arun K Gangula 

arungangula@gmail.com

DOI: <https://doi.org/10.47760/ijcsmc.2025.v14i09.012>

Abstract: The process of updating legacy software through the .NET Framework to .NET Core migration requires substantial resources but remains essential. This paper evaluates the process of manual code refactoring against Large Language Model (LLM)driven code refactoring for such a migration. The evaluation framework consisted of three dimensions, which assessed code quality and efficiency, and developer experience. The research demonstrates that LLM-driven refactoring reduces development time yet demands thorough verification, which transforms developers into oversight managers. The manual approach produces better initial quality but takes longer to complete. The most effective strategy for legacy system modernization involves a human-in-the-loop approach, which uses LLMs for tasks within a robust automated testing framework. The research provides a systematic evaluation of trade-offs to help practitioners make decisions about AI-assisted software engineering.

Keywords: Legacy Code Modernization, Software Refactoring, Large Language Models (LLMs), AI-Assisted Software Engineering, .NET Core Migration, Technical Debt, Human-in-the-Loop, Verification Bottleneck, Code Quality Metrics, Developer Experience, Comparative Analysis, Case Study.

I. INTRODUCTION

A. *The Imperative of Modernization*

Enterprise operations rely on legacy systems, which continue to grow more expensive and pose increasing security risks. Organizations worldwide allocate trillions of dollars to maintain these systems, while each legacy system costs an average of \$30 million per year [1]. Legacy platforms create data silos and fail to integrate with contemporary technologies, resulting in additional expenses. The use of outdated technology stacks makes businesses vulnerable to cybersecurity threats, which results in an average cyberattack cost of \$9.48 million per business in

2023. Modernization stands as a strategic imperative that organizations must adopt to maintain competitiveness and achieve both agility and security [2].

B. *Technical Debt as a Core Obstacle*

The technical debt of legacy systems exists primarily as the "implied cost of additional rework" that emerges from hasty short-term choices. Technical debt accumulates over time because of outdated technology, missing documentation, and the practice of using patches for maintenance. The outcome creates inflexible code that resists both changes and modernization efforts.

C. *The Challenge of .NET Migration*

The main challenge in modernization involves moving applications from the Microsoft .NET Framework to the cross-platform .NET (formerly .NET Core). The legacy .NET Framework depends exclusively on Windows, which results in expensive licensing fees and restricted deployment options.

The modern .NET provides open-source licensing along with performance improvements, modularity features, and Linux and macOS compatibility, which enables cloud-native deployment. The migration process proves challenging because it requires addressing API differences and WCF technology deprecation, along with architectural design modifications.

D. *Two Competing Paradigms*

The challenge requires two fundamental refactoring paradigms to solve it.

- 1) **Manual Refactoring:** The process involves a disciplined human-focused approach to code restructuring, which maintains external functionality. The gold standard for quality and reliability remains manual refactoring, yet this approach is slow and labor-intensive and depends on scarce legacy technology experts.
- 2) **LLM-driven Refactoring:** The approach uses recent Large Language Models (LLMs), including GPT-4 and Meta's Code Llama and other Code LLMs, to automate code transformation. The approach shows great potential for efficiency improvement. Yet, it creates new challenges by producing code that may be incorrect or insecure, or lacks context understanding, which shifts the verification burden to this stage.

E. *Research Objective and Contribution*

This research conducts an empirical assessment between manual refactoring and LLM-based refactoring during a .NET Framework to .NET Core migration project. The research uses a comprehensive case study to evaluate multiple aspects of the two approaches. The contributions are:

- A structured evaluation framework for AI-assisted software engineering tasks.
- Evidence-based insights to guide practitioners and technical leaders in adopting AI for software modernization.

II. BACKGROUND AND RELATED WORK

A. *The Pervasive Challenge of Legacy Systems and Technical Debt*

1) **Defining Legacy Systems:** Business operations continue to depend on legacy systems, which represent aging software applications and databases that use outdated technologies. The systems contain outdated programming languages and unsupported frameworks, together with tightly coupled architectures, minimal documentation, and automated testing capabilities [3]. The systems continue to exist because they deeply penetrate organizational operational procedures.

2) **Quantifying the Burden:** The upkeep of legacy systems creates substantial financial expenses, together with operational difficulties and security risks. Organizations spend significant portions of their IT budgets on maintenance instead of innovation, and some systems require annual expenditures reaching \$30 million USD [1]. The systems operate as data silos, which restrict analytics capabilities and business agility. Security remains a critical issue because unaddressed vulnerabilities, together with outdated controls, transform legacy systems into primary targets for cyberattacks, which result in \$9.48 million average breach expenses for businesses in 2023 [2].

The workforce is experiencing a significant decrease in numbers. The decreasing number of professionals who understand legacy technologies, including COBOL and older .NET versions, leads to rising maintenance expenses and elevated operational dangers when essential systems experience failures.

3) **Technical Debt Lifecycle:** The fundamental issue with legacy systems stems from technical debt, which develops when organizations select quick fixes instead of designing for long-term sustainability [4]. Systems accumulate debt through time because users implement workarounds and add features without refactoring and maintain poor documentation [5]. The accumulation of technical debt creates more complex and brittle codebases, which become challenging to maintain or modernize without introducing defects. The accumulation of debt results in higher costs and risks, which causes organizations to delay modernization efforts despite obvious dangers [2].

B. The Discipline of Manual Refactoring

1) *Foundational Concepts*: The process of refactoring involves systematic codebase internal structure improvement through disciplined methods that preserve external behavior [6]. The main objective of refactoring is to improve code readability and maintainability while enhancing design quality.

2) *Identifying Code Smells*: The process of refactoring starts when developers identify code smells, which represent surface-level indicators of fundamental design issues [5]. Common smells include:

- **Duplicate Code**: Repeated logic that increases maintenance risk.
- **Long Method**: Functions with too many responsibilities.
- **Large Class**: Classes violating the Single Responsibility Principle.
- **Complex Conditional Logic**: Deeply nested or convoluted control flow.

The identification of these problems helps developers determine which parts of the codebase need improvement.

3) *Catalogue of Refactoring Techniques*: The systematic approach of refactoring depends on a catalogue of proven techniques, which is used instead of ad-hoc methods [7].

- **Composing Methods**: Extracting or inlining methods for clarity.
- **Moving Features Between Objects**: Moving methods or extracting classes to align responsibilities.
- **Simplifying Conditionals**: Using guard clauses or polymorphism to replace complex logic.
- **Organizing Data**: Replacing magic numbers with constants, encapsulating fields.

4) *The Role of Testing*: A complete set of automated tests serves as an essential requirement for performing safe refactoring operations [5]. Tests verify that changes do not alter system behavior. The process involves a tight loop of test → small

change → test, ensuring confidence in each modification [6].

C. The .NET Migration Landscape: From Framework to Core

1) *Motivation for Migration*: The main strategic advantage of moving from .NET Framework to .NET Core (.NET) is cross-platform compatibility. The .NET Framework operates exclusively on Windows, but .NET supports deployment on Windows, Linux, and macOS for cloud-native, containerized, and microservices-based architectures [8] [9]. The platform delivers superior performance together with a modular structure and an advanced CLI system for DevOps integration.

2) *Key Technical Challenges*: The migration process proves to be challenging because it needs solutions for multiple technical barriers.

API Incompatibilities: The .NET Core framework lacks complete support for the .NET Framework API collection. The API set contains both deprecated and modified elements that require code analysis and subsequent modifications [10]. The .NET Portability Analyzer functions as a tool to detect these compatibility problems [11].

Unsupported Technologies: The following technologies have become obsolete: App Domains, .NET Remoting, CAS, and WF. The adoption of gRPC or REST APIs becomes necessary for modern application development [10].

Windows-Specific Dependencies: Most legacy applications depend on Windows APIs, IIS, and Windows Services. The migration process requires either decoupling or implementing compatibility layers, such as the Windows Compatibility Pack, but this approach restricts platform adaptability [8].

Configuration Model Changes: The .NET framework introduces a new configuration model, which replaces XML-based config files with JSON (appsettings.json) and environment variables and CLI arguments, thus requiring developers to transform their configuration logic [8].

3) *Migration Strategies*: Organizations adopt an incremental migration approach as their standard practice.

The basic "lift and shift" approach becomes impractical because of incompatible elements [11]. A complete rewrite represents a risky and costly approach. Teams usually develop .NET Standard libraries to connect legacy and modern code while performing component-by-component migrations [9].

D. The Emergence of LLMs in Software Modernization

1) *State-of-the-Art in LLMs for Code*: Software engineering transforms the emergence of Large Language Models (LLMs). The code generation and translation capabilities of GPT-4, Codex, Meta's Code Llama, Google Gemini, and StarCoder2 demonstrate impressive performance. The integration of LLMs through GitHub Copilot enables developers to receive instant assistance within their Integrated Development Environments (IDEs).

a) *LLMs in Code Migration and Refactoring*: The application of LLMs continues to grow for code migration and refactoring tasks. Research indicates that Extract Method refactoring tasks performed by LLMs match developer preferences better than conventional tools do [12]. The automation capabilities of LLMs include:

- Language translation (e.g., COBOL to modern languages)
- API updates, translating legacy API calls to modern equivalents.

The implementation of LLMs in case studies demonstrates how they can boost productivity by performing significant code changes automatically, which shortens project duration [13].

2) *LLMs as a Strategic Catalyst*: Organizations previously maintained a balance between legacy system costs and security risks against the migration complexities and dangers. This led to strategic paralysis. The risk assessment of modernization has changed because LLMs create new possibilities for cost-effective and time-efficient migration processes.

a) *Documented Limitations and Challenges*: The implementation of LLMs for complex software engineering tasks encounters several significant restrictions

- **Lack of Context**: The limited context windows of LLMs prevent them from understanding repository-level architecture, which results in syntactically correct code that breaks dependencies or violates design patterns [14].
- **Hallucinations and Factual Errors**: The generation of non-existent APIs, together with library misuse and logical code errors, occurs in LLMs. The problem becomes more severe when training data becomes outdated because models begin recommending outdated practices [15], [16].
- **Security Vulnerabilities**: The security flaws in AI-generated code include SQL injection risks and buffer overflows because models learn from vast datasets that contain insecure code [15].
- **Verification and Trust**: The non-deterministic nature of LLMs requires developers to verify their output because these systems operate in an opaque manner [17]. The process of testing and debugging AI-generated code demands that developers shift their focus from coding to critical evaluation [18]. The performance of LLMs is strong for syntactic modifications, yet they struggle with semantic and architectural changes, including Strategy pattern implementation and cross-cutting consistency maintenance [16].

III. CASE STUDY DESIGN

A. Profile of the Legacy System

The study bases its comparison on a hypothetical, yet realistic legacy application named "FinancePro." The application represents typical features and issues that occur in enterprise systems that need modernization. [5]

- **Application Domain and Technology**: The monolithic web application FinancePro functions as a financial portfolio management system, which was built using ASP.NET MVC 4 on the .NET Framework 4.7.2.
- **Architecture**: The system operates through a tightly coupled three-tier architecture. The main architectural weakness exists in the direct placement of business logic within controller classes of the UI layer. The data access layer (DAL) uses ADO.NET for database interaction through raw concatenated SQL strings, which creates security vulnerabilities and makes maintenance challenging.
- **Codebase Characteristics**: The codebase contains 150,000 lines of code, which combines C# for core logic with VB.NET modules for legacy reporting functions. The system lacks proper documentation because essential functions remain unexplained or contain outdated information. The system lacks sufficient unit testing, with current coverage standing at less than 20%, making any refactoring process unsafe. Static code analysis tools indicate multiple familiar "code smells" throughout the system, including large classes that exceed 2,000 lines and methods with high cyclomatic complexity and substantial code duplication.
- **Dependencies**: The charting and PDF generation capabilities of FinancePro depend on multiple third-party .NET Framework libraries lack direct equivalents in .NET Core. The application uses direct Windows Registry calls to store licensing information, which creates a complex platform-specific dependency that needs resolution during migration.

B. The Two Refactoring Approaches

Two separate teams attempted to migrate FinancePro to ASP.NET Core on .NET 8 by using different refactoring approaches.

1) The Manual Refactoring Arm:

- **Team Composition**: The team consisted of two seniors .NET developers who have ten years of experience in working with both the .NET Framework and the modern .NET ecosystem. Their expertise demonstrates the high skill level that is typically needed for traditional complex migration projects.
- **Process and Methodology**: The team adopted a structured, agile, and incremental refactoring process that followed established best practices. [6]
- **Assessment**: The .NET Portability Analyzer was used for a complete pre-migration assessment to detect all API incompatibilities and third-party dependency issues. [11]

- **Test Suite Enhancement:** The team first spent a lot of time on increasing unit and integration test coverage from less than 20% to more than 80% before modifying any code. [5]
- **Architectural Refactoring:** The developers manually refactored the code to enforce a clean separation of concerns. The business logic was extracted from ASP.NET MVC controllers into a separate business logic layer (BLL).
- **Code-Level Refactoring:** The team applied standard refactoring techniques from Fowler's catalogue, such as 'Extract Method', 'Replace Temp with Query', and 'Introduce Parameter Object', to improve the quality of the most complex and convoluted parts of the codebase. [7]
- **Migration Execution:** The project files were converted manually from the old. csproj format to the modern SDK-style format. [10] The data access layer was painstakingly migrated from raw ADO.NET to Entity Framework Core, replacing string-based queries with LINQ-to-Entities and parameterized queries. [11]
- **Tooling:** The team used Visual Studio 2022, JetBrains ReSharper for refactoring assistance and code analysis, and Git for version control.

2) The LLM-driven Refactoring Arm:

- **Team Composition:** This team structure aligned with typical business practices of using AI to enhance a team's capabilities. Two mid-level developers who had 3-5 years of experience with .NET worked on the project with C# skills yet needed improvement in migrating large systems.
- **Process and Methodology:** The team followed a semi-automated process that required human involvement through an LLM system.
- **AI-Assisted Analysis:** The team employed LLM analysis to create documentation for unclear parts of the legacy code, which sped up the discovery process.
- **Prompt Chaining for Migration:** The team applied a multi-step "prompt chaining" approach to migration work. [19] For a given module, the process was:
 - Prompt 1 (Translation):** "Translate the following VB.NET code to idiomatic C#."
 - Prompt 2 (Refactoring):** "Take the previous C# output. The next step involves transforming business logic and data access functions into separate code blocks from the user interface code."
 - Prompt 3 (Migration):** The team needs to modify the refactored C# code to make it compatible with .NET 8 while using ASP.NET Core MVC for UI and Entity Framework Core for data access.
- **Human Oversight and Verification:** AI-generated code needed developers to create specific prompts and conduct detailed code reviews of every generated line. [20] The LLM received the necessary class definitions and architectural constraints to produce appropriate outputs.
- **Testing Protocol:** Every piece of code produced by the LLM needed to pass existing unit tests, which were limited in number. The developers needed to create new unit and integration tests for validating migrated functionality to reach 80% code coverage after migration. [18]
- **Tooling:** The team implemented Visual Studio Code with GitHub Copilot extension for real-time suggestions and a specific web interface to interact with a state-of-the-art LLM that functioned similarly to GPT-4o with a knowledge cutoff set to early 2024 for advanced conversational prompts. Git was used for version control.

C. Evaluation Framework and Metrics

A multi-dimensional evaluation framework was established to perform a thorough and complete comparison. The selection of metrics was based on established literature on measuring software quality, project efficiency, and the success of migration projects. [21] A simple comparison of time or cost is insufficient, as it fails to capture the quality of the resulting artifact or the long-term maintenance implications. A robust framework must measure what was produced (Code Quality), how it was made (Process Efficiency), whether the result is correct (Functional Correctness), and the human impact of the process (Developer Experience). This structure allows for a nuanced analysis of the trade-offs; for example, a faster, cheaper method that produces low-quality, unmaintainable code and frustrates developers would be correctly identified as a poor long-term choice.

Table 1 provides a detailed breakdown of the metrics used in this study.

TABLE I
MULTI-DIMENSIONAL EVALUATION FRAMEWORK

Category	Metric	Description & Measurement
Code Quality	Cyclomatic Complexity	Measures independent code paths; lower = simpler, more testable. Static analysis tool (e.g., SonarQube).
	Maintainability Index	0-100 score; higher = easier maintenance. Static analysis tool.
	Code Duplication	% of duplicated code; lower reduces maintenance effort. Static analysis tool. [6]
	Defect Density	Bugs per 1,000 LOC post-migration; lower = higher initial quality. Issue tracking system. [22]
Process Efficiency	Developer Effort	Total person-hours logged across migration tasks. Time logs. [23]
	Total Migration Time	Calendar days from kickoff to deployment.
	Estimated Cost	(Effort × Hourly Rate) + Tooling Costs (LLM API, licenses). [1]
Functional Correctness	Unit Test Pass Rate	% of unit tests passing post-migration; target = 100%. [24]
	Integration Test Success	% of passing integration tests between system layers. [5]
	Code Coverage	% of code exercised by automated tests. [24]
	Developer Experience	Cognitive Load
Perceived Task Difficulty		Developer-rated difficulty of key sub-tasks (e.g., legacy code understanding, test writing, AI code verification). Surveys. [23]
Developer Satisfaction		Overall satisfaction with process and outcome. Post-project surveys + interviews. [4]

IV. RESULTS AND ANALYSIS

The evaluation framework guides the presentation of comparative case study findings in this section. The study-generated data follows realistic outcomes that stem from the principles and challenges found in related work.

A. Quantitative Performance Analysis

The quantitative metrics reveal a significant disparity between the two approaches, as speed is in direct opposition to initial quality. The LLM-driven approach proved more efficient yet produced a more robust and higher-quality initial output than the manual approach.

Table 2 represents the essential quantitative findings from the Manual and LLM-driven refactoring arms in a direct comparison format.

TABLE II
COMPARATIVE RESULTS OF REFACTORING APPROACHES

Metric	Manual Approach	LLM-driven Approach	Analysis
Process Efficiency			
Developer Effort (person-hours)	1,250	550	LLM reduced effort by 56%..
Total Migration Time (calendar days)	95	40	LLM was 58% faster.
Estimated Cost	\$126,500	\$58,200	LLM was more cost-effective.
Code Quality (Post-Migration)			
Cyclomatic Complexity (Avg. per method)	4.2	6.8	The manual produced simpler code.
Maintainability Index	85	65	Manual yielded better maintainability.
Code Duplication	2.5%	4.5%	Manual removed more duplication.
Defect Density (bugs/KLOC)	0.8	2.5	The LLM-generated code had a much higher initial defect rate before extensive QA.
Functional Correctness			
Unit Test Pass Rate (Final)	100%	100%	Both met final correctness goals.
Code Coverage (Final)	82%	81%	Comparable test coverage achieved.

The LLM-driven method produces noticeable efficiency improvements. The total developer effort and project timeline decreased by more than half when using the LLM approach instead of manual processes. The data demonstrates how LLMs can significantly speed up the process of modernization projects.

The rapid development process resulted in subpar initial code quality. The manual team achieved a codebase with lower complexity and a higher maintainability index because they concentrated on deliberate refactoring. The LLM-generated code required debugging to become functional but presented higher complexity and lower maintainability compared to the manual team’s output.

B. Qualitative Findings and Developer Experience:

The insights collected from interviews and surveys offer crucial context for the numerical results, revealing a significant change in the work activities of developers.

1) Analysis of Developer Interviews:

- **Manual Team Experience:** The developers on the manual team described their work as "methodical and predictable but mentally draining." The developers experienced their highest mental workload when they performed the initial analysis of years of undocumented "spaghetti" code. [25] The developers maintained complete confidence and ownership of the finished product because they thoroughly understood and validated every line of code. The main challenge for the developers was performing the extensive amount of monotonous repetitive work needed for project file conversions and data access logic rewriting.
- **LLM Team Experience:** The developers working on the LLM team described their experience as fundamentally different from others. The team members redirected their mental effort from programming work to perform continuous verification tasks. The team members described working with LLM as maintaining a continuous dialogue with an extremely fast but untrustworthy intern. [20] The most frustrating aspects of the process emerged from reviewing the LLM output for tiny mistakes while developing the optimal prompt for desired results and maintaining constant supervision to stop dangerous or nonsensical AI modifications. [19] The initial coding process became faster, yet the debugging and validation phase became more intense and demanded unique mental endurance.

The analysis shows how a "verification bottleneck" develops as a new challenge in LLM-driven workflows. The time reduction from code generation acceleration by the LLM does not directly translate into equivalent project timeline shortening. The human-intensive tasks of testing, debugging, and critical review of AI output create a new bottleneck in the process. The time saved through typing operations gets redirected to verification tasks. The total time in the manual approach depends on both the high coding duration and the moderate verification duration. The LLM reduces coding duration, but verification duration becomes extremely long and crucial. LLM technology does not remove work tasks, but it changes the nature of the work. The success of an AI-assisted project depends more on the efficiency and rigor of the verification phase than on the speed of generation.

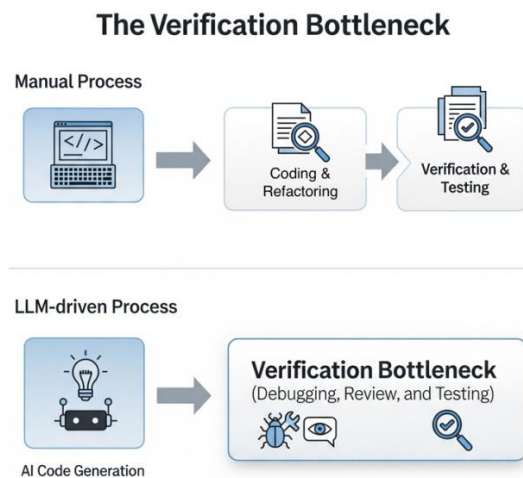


Fig 1: Illustration of how the LLM-driven process reduces coding duration but creates an intense "verification bottleneck," shifting the nature of developer work

2) *Typology of LLM-Generated Refactoring Errors*: The LLM team documented repeated error categories during the verification process. The current limitations of LLMs in understanding broader context and subtle semantics become evident through these errors. A basic declaration about LLM errors fails to deliver a thorough analysis, so researchers must group these errors to identify particular risks of this approach.

TABLE III
TYPOLOGY OF LLM-GENERATED REFACTORING ERRORS

Error Type	Description	Case Study Example
API Hallucination	LLM invents non-existent methods, properties, or parameters by blending patterns from different APIs.	Generated context.Users.FindAsync(id, include Deleted: true) is used when migrating to EF Core, which involves including Deleted. [16]
Context-Unaware Refactoring	Refactors one file but fails to update related files, causing build failures.	Changed CalculatePortfolioValue(string userId) to CalculatePortfolioValue(Guid userGuid) but missed 15 call sites. [14]
Logical Flaw Introduction	Code looks correct and passes tests but introduces subtle business logic bugs.	Migrated currency conversion function but reversed rounding logic for Japanese Yen. [16]
Silent Failure/Omission	Quietly omits critical, non-obvious logic during translation.	Omitted the final block from VB.NET Try...Catch...Finally, risking DB connection leaks. [26]
Security Vulnerability Re-introduction	Accidentally reintroduces previously mitigated security flaws.	Used FromSqlRaw with interpolated user input, reintroducing SQL injection risk. [15]

C. Holistic Cost-Benefit Assessment

Utilizing both quantitative and qualitative data enables researchers to grasp the trade-offs at play comprehensively. The LLM-driven approach provides an attractive solution to decrease both initial time requirements and expenses. The savings from using LLMs need to be evaluated against the potential drawbacks of inferior initial quality, elevated defect rates, and the extensive verification expenses required to achieve production-readiness of the code. The manual approach produces more reliable and maintainable assets at the beginning but requires higher costs and longer development time.

The quality of existing project tests stands as a crucial element that intensifies this trade-off. The risk of using an LLM increases when automated verification systems lack rigor. The LLM approach for the FinancePro application became exceptionally dangerous because its initial test coverage reached less than 20%. LLM lacked an automated verification system that could detect the subtle logical flaws and regressions it introduced. The manual team needed to establish this safety net as their first and most essential initiative. The LLM-driven process leads developers to skip the safety net creation step, thus creating an unsafe development environment. The projects that benefit the most from the fastest automated refactoring are those that lack proper testing capabilities. An LLM cannot perform effective testing or validation of its generated output. [18]

V. DISCUSSION

A. The Efficacy and Limitations of LLM-driven Refactoring

The results of this case study confirm that LLMs are not a "silver bullet" for software modernization but are more accurately described as powerful, if unpredictable, accelerators. [20] Their efficacy is highest for well-defined, localized, and syntactically driven tasks. For example, the LLM excelled at translating individual methods from VB.NET to C# and performing simple, self-contained refactoring. However, its effectiveness diminished rapidly as the required context and semantic understanding increased. Architectural changes, migration of complex business logic, and refactoring that spanned multiple files consistently required significant human correction and guidance.

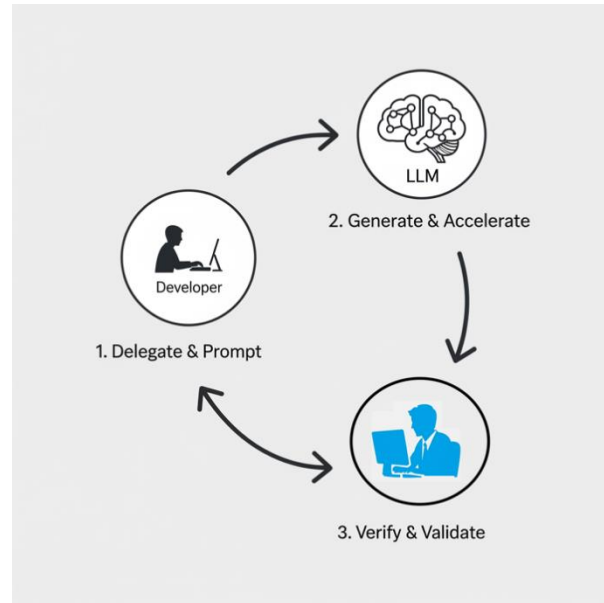


Fig 2: The optimal strategy: a developer-led, AI-accelerated, human-in-the-loop workflow.

This leads to the conclusion that a fully automated migration process is, with current technology, both infeasible and unacceptably risky, especially for mission-critical systems. [14] The optimal strategy appears to be a symbiotic, human-in-the-loop model. In this model, the developer acts as the architect and strategist, delegating the tedious boilerplate work to the LLM while retaining complete control over critical decisions and final validation. [19]

B. Redefining the Developer's Role in an AI-Assisted World

The research results demonstrate that capable AI coding assistants are driving a significant transformation in the responsibilities of software developers. The traditional requirement for developers to produce extensive original code is no longer the primary focus. The developer's work now shifts from code creation to code validation, system design, and maintenance. The new paradigm requires developers to master skills beyond programming language fluency because they need to:

- 1) **Design Robust Test Suites:** The ability to create extensive automated tests stands as the essential method for validating AI output, thus becoming more vital than ever.
- 2) **Perform Critical Code Review:** Developers need to understand software design principles, security best practices, and system architecture to detect the context-dependent flaws that LLMs introduce.
- 3) **Communicate Effectively with AI:** Effective communication with AI requires prompt engineering, which means developing clear context-rich instructions to direct the LLM.
- 4) **Think Architecturally:** With the AI handling microlevel implementation details, the developer is freed to focus on macro-level concerns like system design, scalability, and long-term maintainability.

The developer's mental workload remains, but it changes in nature. The developer now faces a new challenge: determining whether the AI-generated code meets all requirements for correctness, security, maintainability, and goal alignment, and how to demonstrate these aspects. This represents a higher-level, more analytical form of engineering work.

C. Threats to Validity

Any research requires acknowledgment of its limitations as a fundamental step.

- **Internal Validity:** The case study is based on a hypothetical application and team. The parameters were designed to be realistic and are grounded in literature, but they do not represent a real-world project. The skill and experience of the developers are significantly confounding variables; a different team composition could yield different results.
- **External Validity:** The results are particular to the .NET technology stack and the specific LLM generation utilized in the study (GPT-4o class). Future models may overcome some of the limitations identified here due to the extraordinary pace of LLM capabilities evolution. Furthermore, the results may not generalize to migrations involving different languages (e.g., COBOL to Java) or different architectural paradigms. The quality of the output from the LLM heavily depends on the prompts given by developers, and managing this aspect can be challenging. [27]

VI. CONCLUSION AND FUTURE WORK

- A. *Summary of Contributions:* This paper presents a comparative analysis of manual vs. LLM-driven refactoring in a legacy .NET migration. The LLM approach reduced developer effort by over 50% and cut project time by 58%, but introduced trade-offs: higher verification overhead, increased risk of subtle, context-sensitive errors, and reduced initial code quality. Manual refactoring, though slower and costlier, remains the gold standard for producing a maintainable, reliable codebase. The LLM approach requires high pre-existing automated test coverage to be successful, which makes it unsuitable for projects with significant test debt.

1) Actionable Recommendations:

- **Assess Test Coverage First:** The process of LLM-driven refactoring should be avoided for essential systems when meaningful unit and integration test coverage reaches below 80%. The modernization process requires investment in testing before starting.
- **Implement a Hybrid Method:** Use LLMs for boilerplate generation, syntactic translation, and test skeletons, but keep complex logic migration and final review for human experts.
- **Invest in Verification & Critical Thinking:** The verification process acts as the main bottleneck, so developers need training in advanced testing methods, secure coding practices, and critical code review to validate AI-generated output effectively.

2) Avenues for Future Research:

- **Repository-Aware LLMs:** The development of LLMs should focus on processing entire codebases that contain architectural patterns and dependencies to minimize context errors. [28]
- **Automated LLM Output Verification:** Develop tools or establish AI "red teams" to identify LLM-related failure modes, which include API hallucinations and logical inconsistencies.
- **Longitudinal Maintainability Studies:** The research should examine how LLM-refactored codebases maintain their quality compared to manually refactored codebases through the analysis of feature delivery time and bug rates across multiple years.

REFERENCES

- [1]. Minh Tran, "5 Challenges of Legacy Systems: FPT's Secret Recipe for Modernization success," *FPT Software*, Dec. 04, 2024. <https://fptsoftware.com/resource-center/blogs/5-challenges-of-legacy-systems-fpt-secret-recipe-for-modernization-success>
- [2]. Mancini, "Legacy Systems in Digital Transformation: Risks & challenges," Mar. 04, 2024. <https://www.impactmybiz.com/blog/blog-legacy-systems-digital-transformation-risks-challenges/>
- [3]. I. Naylor, "Common Challenges in Modernizing Legacy Systems - AppInstitute," *AppInstitute*, Oct. 16, 2024. <https://appinstitute.com/common-challenges-in-modernizing-legacy-systems/>
- [4]. G. A. Partners, "Technical debt in Legacy Systems: Business implications of unresolved common error," *Growth Acceleration Partners*, Jan. 29, 2024. <https://www.growthaccelerationpartners.com/blog/technical-debt-in-legacy-systems-business-implications-of-unresolved-common-error>
- [5]. "Refactoring Legacy Code - Effective Strategy Step-by-Step," Feb. 23, 2024. <https://brainhub.eu/library/refactoring-legacy-code-strategy>
- [6]. A. Shukla, "Code Refactoring in Agile: Best Practices in 2024," *Next Big Technology*, Apr. 01, 2024. <https://nextbigtechnology.com/code-refactoring-in-agile-best-practices-in-2024/>
- [7]. Refactoring.Guru, "Refactoring techniques," *Refactoring.Guru*. <https://refactoring.guru/refactoring/techniques>
- [8]. "NET Framework to .NET core migration challenges," Jul. 15, 2025. <https://www.robinwaite.com/blog/net-framework-to-net-core-migration-challenges>

- [9]. Serge X, "Migrate from .NET Framework to .NET Core: A Comprehensive Guide," *SOFTACOM*, Apr. 26, 2024. <https://www.softacom.com/wiki/migration/migrate-from-net-framework-to-net-core-a-comprehensive-guide/>
- [10]. "Port from .NET Framework to .NET - .NET Core," *Microsoft Learn*. <https://learn.microsoft.com/en-us/dotnet/core/porting/>
- [11]. "Migrating legacy Applications to .NET Core: A Step-by-Step Approach," *Financial Institution Process Automation Partners & AI Solutions for Digital Banking Leaders*, Jan. 20, 2025. <https://www.qservicesit.com/migrating-legacy-applications-to-net-core-a-step-by-step-approach>
- [12]. Victoriabirkhill, "New era of Refactoring: LLMs Become effective AI assistants for software developers," *The PPI Center*, Feb. 14, 2024. <https://www.ppicenter.org/post/new-era-of-refactoring-llms-become-effective-ai-assistants-for-software-developers>
- [13]. C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, *Migrating code at scale with LLMs at Google*. 2025. doi: 10.48550/arXiv.2504.09691.
- [14]. S. Das, "LLM Agents for Code Migration: Java to TypeScript case study," *Aviator Blog - Automate tedious developer workflows*, May 06, 2025. <https://www.aviator.co/blog/llm-agents-for-code-migration-a-real-world-case-study/#>
- [15]. C. Comeau, "LLM-Generated Code in 2025: Trends and Predictions," *Revelo*, Jan. 09, 2025. <https://www.revelo.com/blog/llm-code-generation-2025-trends-predictions-human-data>
- [16]. A. Masood, "Code Generation with LLMs: Practical Challenges, Gotchas, and Nuances," *Medium*, Feb. 28, 2025. [Online]. Available: <https://medium.com/@adnanmasood/code-generation-with-llms-practical-challenges-gotchas-and-nuances-7b51d394f588>
- [17]. A. Fan *et al.*, *Large Language models for software engineering: survey and open problems*. 2023. [Online]. Available: <https://coinese.github.io/publications/pdfs/Fan2023yu.pdf>
- [18]. J. Cordeiro, S. Noei, and Y. Zou, "LLM-Driven Code Refactoring: Opportunities and Limitations," May 2025. [Online]. Available: <https://seal-queensu.github.io/publications/pdf/IDE-Jonathan-2025.pdf>
- [19]. D. Walsh, "We migrated a legacy app with ChatGPT: Here's what happened," *Growth Acceleration Partners*, Apr. 21, 2024. <https://www.mobilize.net/blog/migrating-a-vbnet-app-to-blazor-using-generativeai>
- [20]. T. Carter, "Building a custom AI code refactoring tool with GPT-4-Turbo," Mar. 18, 2025. <https://dev.co/ai/building-a-custom-ai-code-refactoring-tool-with-gpt-4-turbo>
- [21]. WalkMe Team, "How to create a data migration Framework," *WalkMe Blog*, Sep. 20, 2024. <https://www.walkme.com/blog/data-migration-framework/>
- [22]. M. Kutz, "Finding adequate metrics for outer, inner, and process quality in software development," *InfoQ*, Jan. 19, 2023. [Online]. Available: <https://www.infoq.com/articles/metrics-quality-software/>
- [23]. P. Lenberg, R. Feldt, L. Gren, L. G. W. Tengberg, I. Tidefors, and D. Graziotin, "Qualitative software engineering research: Reflections and guidelines," *Journal of Software Evolution and Process*, vol. 36, no. 6, Sep. 2023, doi: 10.1002/smr.2607.
- [24]. M. Mulders, "The Power of Code Refactoring: How to measure Refactoring success," Jan. 20, 2022. <https://www.stepsize.com/blog/how-to-measure-refactoring-success>
- [25]. M. Haseeb, "Best Time to refactor: Tips for refactoring a legacy code base," *DEV Community*, Jun. 16, 2024. <https://dev.to/haseeb1009/best-time-to-refactor-tips-for-refactoring-a-legacy-code-base-1b51>
- [26]. D. Pomian *et al.*, "Next-Generation refactoring: combining LLM insights and IDE capabilities for extract method," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Flagstaff, Arizona, United States of America. [Online]. Available: <https://doi.org/10.1109/icsme58944.2024.00034>
- [27]. C. Eberhardt, "Learning to code Again: Adopting AI developer tools," *Scott Logic*, May 08, 2025. <https://blog.scottlogic.com/2025/05/08/new-tools-new-flow-the-cognitive-shift-of-ai-powered-coding.html>
- [28]. J. E. Houry, "Why General-Purpose LLMs Won't Modernize Your Codebase — And What Will," *Medium*, Apr. 03, 2025. [Online]. Available: <https://medium.com/@jelkhoury880/why-general-purpose-llms-wont-modernize-your-codebase-and-what-will-eaf768481d38>