Modernizing Legacy Applications in the Era of AI and Large Language Models: A Review of Best Practices

Aditya Saxena
Toronto Metropolitan University
Toronto, Canada
Email: aditya.saxena@torontomu.ca

June 27, 2025

Abstract

Legacy software systems remain critical assets for many enterprises, yet their monolithic architectures and outdated technologies hinder agility, scalability, and innovation. Modernizing these systems is a complex challenge, exacerbated by technical debt, scarce expertise, and evolving regulatory requirements. Recent advances in artificial intelligence, particularly large language models (LLMs), offer promising avenues for automating and accelerating modernization processes. This paper presents a comprehensive literature review of best practices, methodologies, and emerging AI-enabled tools for legacy application modernization. We synthesize findings from state-of-the-art research covering multi-criteria decision-making frameworks, architectural migration patterns, cloud-native refactoring strategies, and LLM-assisted code transformation and UI modernization. We highlight empirical evidence from industrial case studies, identify technical and organizational challenges, and discuss the integration of LLMs into continuous integration and deployment pipelines for incremental modernization. The review concludes with a roadmap for future research, emphasizing the need for explainable AI, human-in-the-loop systems, and domain-specific adaptation to realize fully automated, scalable, and reliable modernization workflows.

Keywords: cloud computing, legacy modernization, software architecture, large language models, DevOps pipelines, user interface modernization, refactoring tools

1 Introduction

Legacy software systems remain critical to the core operations of industries such as finance, healthcare, manufacturing, and government. Despite their continued utility, these systems often suffer from outdated architectures, rigid structures, and accumulated technical debt, which collectively hinder organizational agility, scalability, and innovation. In the current era of digital transformation, modernizing these systems has become a strategic priority, demanding not only technical solutions but also comprehensive decision-making frameworks and organizational alignment.

This paper presents a structured review of best practices and recent innovations in legacy application modernization. It begins with the strategic foundations of modernization, analyzing drivers such as business agility, maintainability, and regulatory compliance. The review highlights structured decision-making methodologies, including multi-criteria decision-making (MCDM) and FUCOM-WSM, which support organizations in navigating modernization trade-offs and investment strategies.

We then examine architectural migration strategies, with a focus on transitioning from monolithic to microservices-based and cloud-native architectures. The discussion includes architectural patterns, anti-patterns, and their implications on maintainability and technical debt. Additionally, we explore infrastructure modernization, encompassing legacy-to-cloud migration, Infrastructure-as-Code, containerization, serverless computing, and cognitive cloud approaches that enable scalable and resilient system deployments.

A central theme of this review is the transformative role of large language models (LLMs) in software modernization. LLMs are increasingly leveraged for code comprehension, automated refactoring, and prompt-based transformation. Their integration into CI/CD workflows enables incremental and automated modernization pipelines. Furthermore, UI modernization has emerged as a frontier for LLM-driven innovation, including automated UI generation, conversational design agents, and enhanced accessibility and responsiveness through models like UXAgent and CrowdGenUI.

The review also surveys emerging toolchains and cognitive automation frameworks that embed AI capabilities into development environments, facilitating model-driven migration and continuous feedback mechanisms. To ground these insights, we analyze empirical case studies across various domains, including COBOL modernization, Java EE refactoring, and digital transformation in financial systems.

Finally, we identify key research challenges, such as the need for explainable AI, human-in-the-loop systems, domain adaptation, and the development of robust benchmarking practices. Together, these insights offer a comprehensive roadmap for researchers and practitioners aiming to build scalable, intelligent, and adaptive modernization workflows.

2 Motivation and Functional Domains of Modernization

Modernizing legacy software systems is a pressing necessity as organizations strive to maintain competitiveness, agility, and compliance in a rapidly evolving technological landscape. Legacy systems often underpin mission-critical operations but exhibit architectural rigidity, obsolete technologies, and high maintenance costs. As a result, modernization is not merely a technical upgrade—it is a strategic realignment that requires holistic planning across multiple domains [1].

To address the complexity and breadth of modernization, this review categorizes the process into six interrelated functional domains, each motivated by distinct operational challenges and technological opportunities:

2.1 Strategic Foundations for Modernization

The modernization initiative typically begins with identifying strategic drivers such as business agility, technical debt reduction, maintainability, and regulatory compliance [1].

Enterprises must make informed decisions about whether, when, and how to modernize legacy systems. Multi-criteria decision-making frameworks, such as FUCOM-WSM and other MCDM techniques, have emerged to aid in selecting optimal modernization paths while balancing cost, risk, and organizational readiness [2]. Additionally, human factors—including change management, skills availability, and governance—play a critical role in the success of modernization efforts [15].

2.2 Architectural Migration Strategies

Architectural refactoring lies at the heart of many modernization projects. A common approach involves decomposing monolithic systems into microservices to enhance modularity, scalability, and maintainability [5]. This shift often leverages service-oriented and cloud-native patterns, but must be carefully managed to avoid architectural anti-patterns and increased complexity [6, 9]. Proper migration strategies can significantly reduce technical debt and improve system evolvability [11].

2.3 Cloud Migration and Infrastructure Modernization

Replatforming legacy systems to cloud environments offers scalability, cost efficiency, and access to advanced services. Organizations pursue cloud migration using strategies such as lift-and-shift, rehosting, refactoring, or rearchitecting, depending on their technical and business constraints [3]. Enabling technologies include Infrastructure-as-Code, containerization, and serverless computing, along with cognitive cloud platforms that automate provisioning and scaling [7, 27]. Data architecture redesign is often essential to handle modern data volumes and access patterns efficiently.

2.4 Role of Large Language Models (LLMs)

Recent advances in artificial intelligence, particularly large language models (LLMs), offer transformative capabilities for software modernization. LLMs can assist in code understanding, refactoring, and migration through prompt-based engineering and auto-completion features [17,20]. Their integration into continuous integration and deployment (CI/CD) pipelines facilitates automated and incremental modernization [16, 35]. These tools reduce manual effort and help bridge knowledge gaps in legacy systems.

2.5 UI Modernization with AI/LLMs

User interface (UI) modernization is a vital yet often overlooked aspect of legacy system transformation. With the rise of LLM-based UI generation and conversational agents, it is now possible to automate much of the UI design and implementation process [28, 30]. Tools like CrowdGenUI and UXAgent further enhance the design experience by learning from user preferences and improving accessibility and responsiveness [29,32]. This enables a more user-centric, adaptive, and inclusive digital experience.

2.6 Tooling and Automation Frameworks

To orchestrate modernization workflows efficiently, organizations increasingly rely on tooling ecosystems that integrate AI capabilities directly into the development environment.

Cognitive automation pipelines, model-driven migration tools, and feedback-driven evaluation mechanisms form the foundation of scalable modernization toolchains [19, 26]. LLM-augmented IDEs provide intelligent support for developers, improving productivity and reducing error rates in migration efforts [36].

Together, these six functional domains provide a comprehensive framework for understanding the modernization landscape. By aligning technical innovation with organizational strategy, they enable sustainable and intelligent legacy system transformation.

2.7 Strategic Foundations for Modernization

2.7.1 Introduction

Modernizing legacy systems requires not only technological upgrades but also strategic alignment with organizational goals. The process is often initiated in response to business pressures such as the need for agility, maintainability, and compliance with evolving regulations. Technical debt accumulated over years of incremental updates often renders legacy systems brittle and hard to evolve. As such, strategic modernization is a multidimensional effort that involves evaluating economic, operational, and human factors [1].

2.7.2 Key Research Questions for Digital Transformation Teams

- What are the primary business drivers and organizational motivations that necessitate legacy system modernization across various industries?
- How can organizations accurately assess and quantify technical debt, operational risks, and strategic value to inform modernization priorities?
- Which multi-criteria decision-making frameworks best enable enterprises to prioritize modernization initiatives under tight budget and resource constraints?
- How can organizations balance trade-offs among modernization cost, implementation risk, downtime, and expected business return on investment (ROI)?
- To what extent do organizational culture, team skills, and stakeholder engagement impact the success of modernization projects?
- What best practices foster cross-functional collaboration between IT, business units, and compliance teams during legacy system transformation?
- How can human-in-the-loop approaches be effectively integrated with automated modernization tools, such as large language models, to enhance accuracy and developer adoption?
- What role do explainable AI and transparent decision-support systems play in increasing trust and adoption of AI-driven modernization workflows?
- How should enterprises structure incremental versus big-bang migration strategies to minimize operational disruption and maximize modernization benefits?
- What metrics and key performance indicators (KPIs) should be established to continuously monitor modernization progress, technical quality, and business impact?

- How can organizations sustain continuous modernization post-migration to prevent legacy issues from re-emerging?
- Which emerging architectural patterns and tooling ecosystems best support modernization in cloud-native and AI-augmented environments?
- What gaps exist in current model-driven, AI-assisted, and pattern-based migration techniques, and how can future research address these gaps?
- How can organizations effectively manage data migration, consistency, and integration challenges during heterogeneous modernization efforts?
- How can regulatory and security compliance challenges be systematically incorporated into modernization planning and execution?
- How do evolving AI and large language model technologies transform legacy modernization practices, and what new research is needed to leverage these capabilities fully?
- What socio-technical challenges are unique to public sector or heavily regulated industries during modernization, and how can these be mitigated?
- How can knowledge capture and documentation of legacy systems be automated to support modernization, ongoing maintenance, and training?
- What mechanisms ensure equitable distribution of modernization benefits across organizational units and help prevent vendor lock-in?
- How can empirical case studies and real-world data be systematically collected and shared to accelerate learning and innovation in modernization practices?

2.7.3 Timeline of Important Papers

- **2023** – Ogunwole et al. introduced a scalable framework for data architecture modernization, emphasizing the role of governance and integration [15].
- **2024** – Jomhari et al. proposed the FUCOM-WSM method for prioritizing modernization strategies using structured multi-criteria decision-making [2].
- **2025** – Assunção et al. presented a comprehensive survey of modernization strategies and emphasized research opportunities related to business drivers and organizational dynamics [1].

2.7.4 Analysis of Literature

The foundational literature on modernization strategy converges around three core axes: modernization as enterprise strategy, decision-making under uncertainty, and socio-organizational readiness. Below, we critically analyze key contributions across four thematic dimensions: strategic alignment, decision frameworks, organizational readiness, and unresolved challenges.

1) Strategic Alignment with Enterprise Objectives Assunção et al. [1] articulate modernization not as a standalone IT initiative but as an integrated component of enterprise digital strategy. Their synthesis draws from industrial and academic evidence to argue that modernization must deliver measurable business value—such as improved agility, reduced risk, and faster innovation—while aligning with compliance and operational mandates. Importantly, they caution against technocentric modernization efforts that ignore business context, organizational culture, or long-term vision.

Their work calls for the integration of modernization planning with digital transformation roadmaps and enterprise architecture practices (e.g., TOGAF or ArchiMate). For practitioners, this entails the development of modernization blueprints that align legacy system decisions with KPIs, transformation milestones, and stakeholder incentives. Yet, while their conceptual framing is strong, the paper leaves a gap in translating these strategic insights into actionable tooling or decision dashboards for enterprise architects.

2) Structured Decision-Making Using FUCOM and WSM The Full Consistency Method (FUCOM) combined with the Weighted Sum Method (WSM) constitutes a powerful decision-making framework employed to prioritize complex modernization initiatives involving multiple, often conflicting criteria. This structured approach supports enterprises in systematically evaluating and ranking modernization options based on quantitative and qualitative factors, enabling informed and consistent decisions under resource constraints.

1. Overview of FUCOM

FUCOM is a multi-criteria weighting technique designed to derive precise and consistent weights for decision criteria based on expert judgments. Unlike traditional pairwise comparison methods, FUCOM emphasizes full consistency by minimizing deviations in relative importance assessments among criteria. The process involves:

- **Ranking Criteria:** Experts rank the modernization criteria (e.g., technical debt, compliance urgency, cost) in order of relative importance.
- **Pairwise Ratio Determination:** Experts specify comparative importance ratios between adjacent criteria in the ranking (e.g., criterion A is twice as important as criterion B).
- **Optimization for Consistency:** FUCOM formulates a linear programming problem that minimizes the deviation from consistency across all pairwise ratios, resulting in a set of weights that reflect expert priorities with minimal logical contradictions.

The outcome is a robust weight vector that accurately represents the relative significance of each criterion, forming the foundation for objective decision-making.

2. Overview of WSM

The Weighted Sum Method (WSM) is a straightforward and widely used technique for ranking alternatives based on the weighted sum of their performance scores across all criteria. The procedure entails:

- **Scoring Alternatives:** Each modernization alternative (e.g., rehosting, refactoring, redevelopment) is scored against every criterion, usually on a normalized scale (e.g., 0 to 10).

- **Aggregation:** The scores are multiplied by their corresponding FUCOM-derived weights and summed to produce a composite preference score for each alternative.
- **Ranking:** Alternatives are ranked based on their composite scores, enabling prioritization according to the aggregated evaluation.

WSM's simplicity facilitates transparent calculation and easy interpretation, making it suitable for diverse decision contexts.

3. Step-by-Step Application of FUCOM-WSM in Legacy Modernization

1. **Define Evaluation Criteria:** Collaborate with stakeholders to identify relevant criteria influencing modernization decisions, such as technical debt reduction, compliance urgency, operational risk, cost, expected ROI, and organizational readiness.
2. **Rank Criteria and Specify Relative Importance:** Engage domain experts to rank the criteria in descending order of importance and provide pairwise comparative ratios for adjacent criteria. For example, experts might judge technical debt as 1.5 times more important than compliance urgency.
3. **Compute Criteria Weights via FUCOM:** Formulate and solve the FUCOM linear programming model to derive consistent weights that honor the pairwise ratios and minimize inconsistency. This step may involve using optimization solvers such as Excel Solver, MATLAB, or Python libraries (e.g., PuLP).
4. **Identify Modernization Alternatives:** Enumerate feasible modernization approaches tailored to the enterprise context, such as lift-and-shift rehosting, incremental refactoring, microservices redevelopment, or cloud-native rebuild.
5. **Score Alternatives Against Criteria:** Assess each alternative's performance relative to every criterion, based on quantitative data or expert judgment. Normalize the scores to a common scale for comparability.
6. **Aggregate Scores Using WSM:** Multiply each alternative's criterion score by the corresponding FUCOM weight and sum across all criteria to obtain an overall preference score.
7. **Rank Alternatives and Make Decisions:** Rank modernization options based on their aggregated scores, facilitating prioritized investment and execution planning.
8. **Validate and Iterate:** Review the results with stakeholders, adjust inputs if necessary, and iterate the process to reflect updated information or shifting priorities.

4. Advantages and Limitations

The FUCOM-WSM framework offers several advantages:

- **Consistency:** FUCOM ensures minimal inconsistency in expert-derived weights, improving reliability over traditional weighting methods.
- **Transparency:** The weighting and scoring process is transparent, enabling stakeholder engagement and buy-in.
- **Flexibility:** Applicable to both quantitative and qualitative criteria, suitable for complex modernization scenarios.

However, effective use requires:

- **Expertise:** Skilled facilitation to elicit accurate rankings and ratios.
- **Data Quality:** Reliable scoring of alternatives, which can be challenging with uncertain or incomplete data.
- **Handling Uncertainty:** While FUCOM minimizes inconsistency, it does not explicitly model uncertainty or dynamic changes in criteria importance.

5. Conclusion

FUCOM-WSM provides a structured, rigorous, and practical approach for prioritizing legacy system modernization options. By combining consistent criteria weighting with transparent aggregation, it supports balanced decision-making in environments constrained by resources and multiple competing objectives. When integrated with organizational feedback loops and updated data, FUCOM-WSM can significantly enhance the effectiveness and traceability of modernization planning.

3) Organizational Dynamics and Human-Centric Governance Ogunwole et al. [15] shift focus from technical optimization to organizational readiness. Their study identifies frequent causes of modernization delays: lack of cross-functional coordination, absence of executive sponsorship, and unclear role ownership. They argue for the creation of modernization governance structures such as steering committees, change management offices, and architecture review boards.

Practically, this implies that successful modernization leaders must possess a blend of soft and hard skills—project scoping, stakeholder negotiation, organizational diagnosis, and team skill assessment. Upskilling is especially critical in contexts where legacy system maintainers must transition to cloud-native and DevOps paradigms. However, the paper does not explore incentive models or funding mechanisms to sustain such organizational reforms over long modernization timelines.

4) Gaps, Critiques, and Opportunities for Integration Collectively, the literature affirms that modernization must be strategically justified, quantitatively evaluated, and organizationally supported. Yet, there are notable gaps:

- **Strategy–Execution Disconnect**: Frameworks like FUCOM-WSM are valuable for planning, but there is a lack of integration with execution platforms such as CI/CD pipelines, portfolio management tools, or architecture modeling suites.
- **Limited Agility**: Most approaches assume a linear or top-down planning process. In dynamic environments, iterative planning and real-time feedback loops (e.g., from pilot migrations) may yield more adaptive and resilient outcomes.
- **Neglect of Behavioral Economics**: Few studies address how cognitive biases, political dynamics, or decision fatigue may distort modernization planning—even when structured frameworks are applied.
- **Underutilized LLM Capabilities**: Emerging tools powered by large language models (e.g., for strategy synthesis or documentation mining) are absent from the strategic literature, presenting a future avenue for augmenting human decision-makers.

2.8 Architectural Migration Strategies

2.8.1 Introduction

Architectural migration is one of the most technically intensive aspects of legacy system modernization. At its core, it involves transitioning from tightly coupled, monolithic architectures to more modular, scalable, and maintainable designs such as microservices, service-oriented architectures (SOA), or cloud-native platforms. This shift enables independent deployment, fault isolation, and improved alignment with agile development and DevOps practices. However, it also introduces architectural complexity, operational overhead, and new risks if not carefully managed [5]. Effective migration strategies must therefore balance design quality with feasibility, minimizing technical debt and ensuring alignment with long-term business and technical goals [6, 9].

2.8.2 Key Research Questions for Digital Transformation Teams

- What are the best practices for decomposing monolithic systems into microservices to optimize modularity and scalability?
- How can organizations identify and mitigate common architectural anti-patterns during legacy-to-microservice migration?
- What is the impact of service-oriented and cloud-native architectural patterns on system refactoring, fault tolerance, and deployment agility?
- How does architectural migration influence technical debt reduction, system performance improvements, and long-term maintainability?
- What methodologies effectively guide the selection of service granularity to balance complexity and operational overhead?
- How can architectural governance be structured to support consistent API design, versioning, and contract management across distributed services?
- In what ways do socio-technical factors, such as team organization and cross-domain collaboration, affect the success of architectural migration initiatives?
- How can continuous monitoring and architectural fitness functions be integrated into migration pipelines to ensure ongoing system health?

2.8.3 Timeline of Important Papers

- **2020** – Lenarduzzi et al. investigated the impact of migrating monoliths to microservices on technical debt in industrial settings [6].
- **2021** – Wolfart et al. proposed a roadmap for legacy system modernization via microservices, emphasizing phased migration and domain-driven design [5].
- **2023** – Velepucha and Flores surveyed microservices principles, patterns, and migration challenges, with attention to service granularity and interface stability [9].
- **2024** – Tuusjärvi et al. presented an empirical case study on the migration of a public-sector legacy system to microservices, highlighting socio-technical obstacles [11].

2.8.4 Analysis of Literature

The literature on architectural migration reveals a progression from conceptual advocacy of microservices to pragmatic considerations involving tooling, governance, and socio-technical factors. Below, we synthesize and critically assess key contributions across four dimensions: technical debt impact, migration strategies, architectural governance, and organizational dynamics.

1) Technical Debt and Architectural Complexity Lenarduzzi et al. [6] provide a quantitative study of software projects that migrated from monolithic architectures to microservices. Their results indicate measurable reductions in maintainability-related technical debt post-migration, including improvements in modularity, testability, and change impact analysis. However, they also report a temporary surge in architectural complexity, primarily due to service orchestration overhead and duplicated functionalities during transition phases. This highlights the need for controlled intermediate architectures—such as layered decomposition and internal service facades—that mitigate the abrupt shift from centralized control to distributed autonomy.

From a practitioner's viewpoint, the findings suggest that modernization engineers must be adept not only in modularization but also in managing short-term complexity through tools such as dependency graphs, architectural fitness functions, and metrics-driven refactoring.

2) Migration Roadmaps and Decomposition Patterns Wolfart et al. [5] propose a structured, phased migration framework grounded in Domain-Driven Design (DDD) and bounded context separation. Their roadmap recommends starting with business-aligned decomposition of the monolith, using APIs and anti-corruption layers to isolate legacy dependencies. The process embraces the Strangler Fig pattern, where new services gradually replace monolithic modules.

This roadmap is particularly applicable in regulated or high-stakes domains (e.g., finance, government) where full reengineering is infeasible. However, its successful execution hinges on concrete skills such as domain modeling, API lifecycle management, and non-functional testing. Critically, the authors do not provide guidance for adapting their roadmap in settings with weak domain boundaries or undocumented monolithic logic—a frequent reality in legacy-heavy organizations.

3) Governance and Pattern-based Migration Velepucha and Flores [9] offer a synthesized taxonomy of microservice architecture principles and migration challenges. Their review highlights recurrent pitfalls: overly fine-grained services, inconsistent API contracts, and insufficient governance for inter-service data sharing. They advocate for the use of architectural blueprints, service meshes (e.g., Istio), and schema governance tools to mitigate these issues.

From a tooling perspective, this implies that architects must balance theory with deployment practicality. For instance, event-driven design may promise decoupling but requires robust message tracing and schema versioning infrastructure. The paper also indirectly warns against "microservice envy"—a trend where microservices are adopted without clear business or operational justification.

4) Organizational Dynamics and Socio-Technical Alignment Tuusjärvi et al. [11] explore the often underrepresented organizational dimension of architectural migration. Their case study of a public-sector system shows how rigid legacy structures, siloed teams, and unclear ownership slow down migration even when technical roadmaps exist. They emphasize the value of cross-functional collaboration, iterative domain learning, and continuous architectural decision-making via mechanisms like Architecture Decision Records (ADRs).

The study draws attention to an important insight: modernization is rarely a purely technical endeavor. It requires aligning organizational incentives, building shared architectural language across stakeholders, and fostering DevOps and agile mindsets. These socio-technical enablers often determine the success or failure of technically sound migration efforts.

5) Critical Gaps and Research Opportunities Across the reviewed literature, a notable gap exists in migration-readiness evaluation and post-migration assessment. While most studies focus on *how* to migrate, few address *when* to migrate or how to measure the return on architectural investment. Metrics such as modularity index, service cohesion, and architectural entropy are mentioned sporadically but lack consensus or tooling support. Furthermore, there is limited empirical data on long-term evolution of microservices-based systems—an area ripe for longitudinal studies.

Additionally, integration between enterprise architecture models (e.g., TOGAF, ArchiMate) and modernization roadmaps remains weak. A promising research direction lies in bridging high-level architectural planning with executable modernization pipelines via model-driven engineering and LLM-assisted architecture mining.

2.8.5 Deduction

Architectural migration is a foundational enabler for modern systems design, offering long-term benefits in modularity, scalability, and maintainability. However, the process is complex and must be approached systematically to avoid new forms of architectural debt and performance regressions. Successful strategies blend domain-driven decomposition with phased rollouts, architectural governance, and human-centric change management. Future work should focus on developing actionable assessment models, automation support for refactoring, and empirical metrics to evaluate architecture health across migration phases.

2.9 Cloud Migration and Infrastructure Modernization

2.9.1 Introduction

As digital transformation accelerates across industries, cloud computing has emerged as a foundational pillar for modern software infrastructure. Replatforming legacy systems to cloud environments offers not only improved scalability and resilience but also opportunities for cost optimization, security enhancement, and service innovation. Cloud migration is not a one-size-fits-all endeavor; it involves nuanced strategies such as lift-and-shift (rehosting), refactoring, rearchitecting, or rebuilding—each with distinct implications for risk, cost, and time-to-value [3].

Modern infrastructure practices such as Infrastructure-as-Code (IaC), container orchestration, and serverless execution models are vital enablers of this transformation.

Moreover, cognitive cloud capabilities—including auto-scaling, AI-driven provisioning, and predictive fault-tolerance—allow legacy systems to be modernized in a more automated and adaptive manner [7, 27]. A key consideration throughout this process is the redesign of data architectures to accommodate distributed processing, real-time analytics, and cloud-native storage semantics.

2.9.2 Key Research Questions for Digital Transformation Teams

- Which cloud migration strategies—such as rehosting, replatforming, refactoring, or rebuilding—are most effective for various legacy workloads and organizational maturity levels?
- How do enabling technologies like Infrastructure-as-Code (IaC), container orchestration, and serverless computing impact the efficiency, reliability, and scalability of modernization efforts?
- What are the critical trade-offs between system performance, operational cost, and architectural complexity when selecting cloud migration patterns?
- How can data architectures be redesigned and governed to ensure scalability, interoperability, security, and regulatory compliance in cloud-native environments?
- What methodologies best support incremental migration and continuous delivery in cloud infrastructure modernization?
- How can cognitive cloud capabilities and AI-driven automation be leveraged to optimize resource provisioning and fault tolerance during migration?
- What metrics and monitoring frameworks effectively evaluate cloud migration success across technical and business dimensions?

2.9.3 Timeline of Important Papers

- **2019** – Fahmideh et al. conducted an empirical study on the challenges of migrating legacy software to the cloud, emphasizing risk perception and planning gaps [7].
- **2023** – Hasan et al. presented a comprehensive review of cloud migration from an architectural lens, classifying strategies and key concerns [3].
- **2024** – Yousef et al. explored cognitive cloud platforms and their potential in automating modernization tasks and resource provisioning [27].

2.9.4 Analysis of Literature

The literature on cloud migration underscores three recurring themes: strategic migration pathways, infrastructure automation, and the redesign of data and operations for cloud-native environments.

1) Cloud Migration Strategies and Decision Complexity Hasan et al. [3] categorize cloud migration strategies into rehosting, replatforming, refactoring, rearchitecting, and rebuilding. Each strategy entails a distinct level of effort, risk, and transformation depth. Their study provides architectural evaluation criteria—such as coupling, technology obsolescence, and performance bottlenecks—to guide selection. The authors also stress the importance of a pre-migration readiness assessment, highlighting that many failures stem from underestimating legacy code complexity and data interdependencies.

Practitioners must evaluate migration fit using both qualitative (e.g., business urgency, skill availability) and quantitative (e.g., service latency, cost projections) indicators. However, the study does not address how to dynamically shift strategies mid-migration—an increasingly relevant challenge as priorities evolve.

2) Infrastructure-as-Code, Containerization, and Serverless Enablers Fahmideh et al. [7] emphasize the technical and organizational challenges in adopting modern cloud infrastructure paradigms. Their empirical findings show that traditional IT teams often struggle with Infrastructure-as-Code (IaC) tools like Terraform or AWS CloudFormation due to the steep learning curve and mindset shift from manual provisioning to declarative automation. Additionally, container orchestration platforms (e.g., Kubernetes) introduce complexity in service discovery, security, and resource allocation that must be proactively managed.

This implies a need for upskilling programs, reference architectures, and governance policies to make infrastructure modernization sustainable. Moreover, serverless computing—while offering scaling and pricing advantages—poses limitations in control, cold-start latency, and vendor lock-in, particularly for latency-sensitive or highly regulated workloads.

3) Cognitive Cloud and Intelligent Resource Management Yousef et al. [27] introduce the concept of cognitive cloud platforms, which use AI and machine learning to automate provisioning, load balancing, anomaly detection, and scaling decisions. Their work explores how these capabilities can reduce operational overhead, improve SLA compliance, and support self-healing systems.

This represents a shift from reactive to predictive infrastructure management. However, integration challenges remain, especially in hybrid environments where legacy systems are only partially cloud-enabled. The authors also call for greater transparency and explainability in AI-driven infrastructure decisions, particularly in sectors with strict compliance requirements.

4) Data Architecture Redesign Across the literature, data emerges as a critical modernization bottleneck. Legacy systems often rely on tightly coupled, stateful, and normalized schemas that are ill-suited to distributed, cloud-native architectures. There is a growing consensus that data architecture modernization—through practices like polyglot persistence, data sharding, eventual consistency, and event streaming—is essential to fully leverage cloud elasticity and scale.

Yet, most studies stop short of prescribing actionable frameworks for data migration planning, consistency guarantees, or governance in multi-cloud environments. This limits the applicability of architectural guidance to real-world systems with complex data lineage, sovereignty, and integration constraints.

2.9.5 Deduction

Cloud migration and infrastructure modernization represent both technical transformation and organizational evolution. While structured migration strategies and enabling technologies like IaC and containers are well-documented, their implementation in legacy contexts remains uneven. The growing adoption of cognitive cloud platforms offers promising automation potential, yet raises new questions about observability, control, and integration.

A significant gap persists around data architecture transformation—arguably the most critical enabler of long-term modernization success. Future research must develop decision-support tools, cloud-native reference patterns, and governance models to bridge architectural planning with sustainable cloud operations.

2.9.6 Deduction

The strategic foundation for modernization must integrate technical, economic, and human-centered considerations. While frameworks like FUCOM-WSM aid in structuring decisions, successful modernization also hinges on aligning modernization goals with organizational priorities and managing human factors effectively. Future work in this domain should aim to develop adaptive, context-aware strategies that are sensitive to both operational constraints and cultural dynamics within enterprises.

2.10 Role of Large Language Models (LLMs)

2.10.1 Introduction

Large Language Models (LLMs) have rapidly evolved from general-purpose conversational agents to domain-specialized tools capable of assisting in code analysis, generation, and transformation. Their application in legacy system modernization is particularly promising due to their ability to process natural language, understand source code semantics, and generate contextually relevant suggestions. LLMs can aid in tasks such as source-to-source translation, API migration, refactoring, and documentation generation [17, 20].

When integrated into DevOps workflows, LLMs extend their utility by supporting CI/CD automation, generating migration unit tests, and even performing real-time code validation. Their prompt-based and feedback-tuned capabilities help bridge gaps in outdated, poorly documented codebases, effectively becoming cognitive collaborators in the modernization lifecycle [35, 36].

2.10.2 Key Research Questions for Digital Transformation Teams

- How can large language models (LLMs) be effectively applied to comprehend, refactor, and transform legacy codebases with high accuracy and minimal manual intervention?
- What are the strengths, limitations, and best practices of prompt engineering techniques in guiding LLMs for reliable and context-aware modernization tasks?
- In what ways can LLMs be seamlessly integrated with existing DevOps pipelines and continuous integration/continuous deployment (CI/CD) tools to facilitate automated, incremental legacy system modernization?

- What are the potential risks associated with the deployment of LLMs in mission-critical legacy systems, including issues related to reliability, explainability, security, and unintended biases, and how can these risks be mitigated?
- How can human-in-the-loop frameworks be designed to complement LLM capabilities while ensuring quality assurance and traceability in modernization workflows?

2.10.3 Timeline of Important Papers

- **2023** – Talasila et al. proposed LLM-driven pipelines for monolith-to-microservice transformations, demonstrating automated decomposition [17].
- **2024** – Sato et al. introduced prompt-based LLM workflows for code completion and migration support in enterprise settings [20].
- **2025** – Liu et al. developed LLMigrate, a model adaptation technique for improving LLM efficiency in codebase refactoring [35].
- **2025** – Ziftci et al. at Google presented real-world evidence on scaling LLMs for large-scale code migration in production systems [36].

2.10.4 Analysis of Literature

1) Prompt-Based Code Transformation and Comprehension Talasila et al. [17] present a system where LLMs interpret code segments, identify modular boundaries, and propose microservice candidates through structured prompts. Their experiments show that even general-purpose models, when guided correctly, can decompose monolithic code into loosely coupled components. However, prompt engineering remains a bottleneck—requiring significant trial-and-error, domain context, and validation frameworks to ensure safe refactoring. Practitioners must master the art of prompt calibration, leveraging few-shot examples, architectural hints, and schema annotations to enhance LLM outputs.

2) LLMs in CI/CD and Continuous Modernization Sato et al. [20] extend LLM usage into CI/CD pipelines. Their system generates code suggestions and validation scripts based on diffs and commit history. This enables incremental modernization—where legacy systems evolve continuously without requiring disruptive rewrites. They emphasize tight integration with version control and test frameworks to prevent regressions. Nevertheless, challenges persist around reproducibility, model drift, and the need for human-in-the-loop validation to catch hallucinations or unsafe suggestions.

3) Model Adaptation and Efficiency at Scale Liu et al. [35] introduce "LLMigrate," a lightweight fine-tuning technique that adapts pretrained models to specific legacy domains such as COBOL or Java EE. This approach significantly improves token efficiency, latency, and semantic precision. It also demonstrates that targeted pretraining can address the structural idiosyncrasies of enterprise codebases—especially those that use outdated idioms or proprietary frameworks. However, the paper notes that fine-tuning infrastructure and high-quality training datasets are prerequisites, which may limit adoption in smaller organizations.

4) Industrial Deployment and Production-Scale Evidence Ziftci et al. [36] document how Google used LLMs at scale for migrating millions of lines of internal code. They detail tooling integration with IDEs, code search engines, and telemetry systems to support safe, testable, and traceable migrations. Key success factors included model debiasing, feedback loops from developer prompts, and careful rollout control. Their experience illustrates that while LLMs can scale transformation, doing so safely requires engineering discipline, observability, and rollback mechanisms.

2.10.5 Deduction

LLMs are emerging as powerful tools for automating and accelerating legacy software modernization. Their capabilities span comprehension, refactoring, and continuous integration, enabling new paradigms like prompt-driven development and automated migration. However, leveraging LLMs effectively requires a blend of skills—prompt design, pipeline integration, and model governance.

Current literature provides strong evidence of technical feasibility and industrial potential, but also exposes gaps in explainability, trustworthiness, and fine-grained control. Future research should focus on hybrid architectures that combine LLMs with static analysis, benchmark frameworks for LLM output evaluation in legacy contexts, and domain-specific fine-tuning pipelines to improve model safety and utility in real-world modernization tasks.

2.11 UI Modernization with AI/LLMs

2.11.1 Introduction

User Interface (UI) modernization is a crucial yet often under-prioritized component of legacy system transformation. While backend systems typically attract the bulk of modernization investments, outdated UIs can significantly degrade usability, accessibility, and user satisfaction. Advances in AI, especially Large Language Models (LLMs), have begun to address this gap by automating various aspects of UI generation, personalization, and enhancement.

Modern LLM-based UI tools now support design-to-code conversion, conversational design assistance, automated feedback loops, and adaptive layout generation [28, 30]. Frameworks like CrowdGenUI and UXAgent leverage user behavior data and preference modeling to produce interfaces that are not only functional but contextually adaptive and inclusive [29, 32]. These developments signal a paradigm shift toward user-centric, co-designed, and continuously evolving UI systems.

2.11.2 Key Research Questions for Digital Transformation Teams

- How can large language models (LLMs) be utilized to automate user interface (UI) generation from diverse inputs such as design specifications, wireframes, and natural language conversations?

- What methodologies and constraints are necessary to ensure that LLM-generated UIs are accessible, responsive, inclusive, and compliant with established usability standards?

- How do adaptive systems like UXAgent and CrowdGenUI incorporate user feedback and preference data to personalize and optimize UI component generation?
- What potential risks, including design inconsistency, hallucinated elements, and user experience (UX) regressions, are associated with LLM-generated interfaces, and how can these be detected and mitigated?
- How can explainability and transparency be integrated into LLM-driven UI generation tools to support designer trust and iterative refinement?

2.11.3 Timeline of Important Papers

- **2023** – Chen et al. introduced AI-Chat2Design, a conversational framework for transforming dialogue into design blueprints [33].
- **2023** – Guo et al. proposed Design2Code, demonstrating LLMs' ability to generate accurate UI code from high-level design intents [34].
- **2024** – Wu et al. presented UICoder, a feedback-driven fine-tuning system for LLM-generated UI code [28].
- **2024** – Liu et al. developed CrowdGenUI, which adapts LLM outputs based on user preferences and crowd-sourced design norms [29].
- **2024** – Zhao et al. proposed UXAgent, an LLM-powered benchmark suite and toolkit for accessibility and UX optimization [32].

2.11.4 Analysis of Literature

1) Conversational and Prompt-Driven UI Generation Chen et al. [33] introduced AI-Chat2Design, which transforms natural language conversations into structured UI representations. Their model enables designers and users to co-create interfaces through iterative dialogues. While the system is effective in generating initial design sketches, it requires additional tooling for refinement and lacks formal guarantees on UI accessibility or compliance with design standards.

Similarly, Guo et al. [34] demonstrated that LLMs can generate semantically correct UI code from high-level prompts and layout descriptions. Their evaluation shows high code accuracy but notes variability in UI styling, responsiveness, and adherence to design tokens—highlighting the need for post-processing layers or design system integration.

2) Feedback-Enhanced UI Code Generation Wu et al. [28] introduced UICoder, which fine-tunes LLMs using automated feedback signals derived from code correctness and UI rendering quality. This loop improves alignment with expected design patterns and reduces hallucinated components. The authors emphasize the role of domain-specific tuning datasets and hybrid evaluation techniques that combine static checks (e.g., WAI-ARIA compliance) with visual validation.

3) Personalization and User-Centric Design Liu et al. [29] presented CrowdGenUI, which builds a preference-aware generation pipeline by learning from crowd-sourced design evaluations. Their system adjusts UI layout, widget selection, and color schemes based on inferred user preferences and historical choices. This advances the concept of "responsive personalization," allowing for adaptive UI generation across personas and platforms.

Zhao et al. [32] developed UXAgent, a benchmarking suite for evaluating LLM-generated interfaces in terms of accessibility, responsiveness, and aesthetic alignment. They integrate automated testing with human review cycles to iteratively improve UX quality. The toolkit supports localization, screen reader validation, and responsive grid adaptation—filling critical gaps in automated LLM output validation.

4) Limitations and Future Considerations Despite their promise, current LLM-based UI systems face several limitations:

- **Design consistency**: Generated components may drift stylistically from existing design systems without explicit constraints.
- **Accessibility regression**: Automated outputs often overlook compliance with WCAG and other accessibility standards unless explicitly enforced.
- **Interpretability**: Unlike rule-based UI generation, LLM-generated code lacks transparency, making it harder to debug or trace design intent.
- **Overreliance on datasets**: Most models are trained on modern UI paradigms, potentially biasing outputs against legacy-compatible designs.

Addressing these challenges will require tighter coupling between design systems, testing frameworks, and prompt-engineered LLM outputs.

2.11.5 Deduction

LLMs are redefining the boundaries of user interface modernization by enabling automated, dialog-driven, and preference-adaptive UI generation. From design sketching to production-ready code, these tools reduce manual overhead and foster inclusive, user-centric interfaces. Yet, achieving robust, accessible, and maintainable UI code at scale demands structured feedback, personalization models, and explainability safeguards.

Future work should explore the integration of LLMs with live design systems (e.g., Figma, Storybook), formal UX metrics, and accessible-by-default generation pipelines to fully realize intelligent UI modernization in legacy systems.

3 Empirical Case Studies and Industrial Applications

3.1 Introduction

While modernization frameworks and technologies have matured significantly, their real-world effectiveness is best understood through empirical case studies. Industrial modernization projects vary widely in scale, complexity, and constraints—ranging from mainframe COBOL systems to Java EE enterprise platforms and FinTech transactional sys-

tems. These case studies provide crucial insights into the practical challenges, decision-making trade-offs, technical bottlenecks, and organizational dynamics that shape modernization outcomes.

By examining success stories and failure analyses, this section highlights the metrics used to assess modernization effectiveness and identifies recurring patterns in cross-industry applications. The reviewed cases serve as a ground-truth counterpoint to theoretical models, offering evidence-based lessons for practitioners and researchers alike.

3.1.1 Key Research Questions for Digital Transformation Teams

- What are the critical success factors and common failure modes observed in real-world legacy system modernization initiatives across industries?
- How do organizations quantitatively and qualitatively measure the effectiveness, business value, and return on investment (ROI) of their modernization efforts?
- Which modernization patterns and strategies consistently emerge across different legacy technology domains (e.g., COBOL, Java EE) and industry verticals?
- What best practices, including technical, organizational, and process-oriented approaches, can be distilled from a broad range of modernization case studies to guide future projects?
- How can lessons learned from diverse modernization projects be systematically captured and disseminated to accelerate industry-wide adoption of effective modernization methodologies?

3.2 Expanded Case Studies

3.2.1 Case Study 1: Microservice Migration in a Telco Enterprise (Lenarduzzi et al., 2020)

Context: The case involves a large European telecommunications company that undertook the modernization of its long-standing Java-based monolithic customer management platform. This system had evolved over more than ten years, resulting in substantial technical debt manifested as code smells, architectural violations, and high coupling among components.

Objective: The primary goal was to improve system maintainability, reduce technical debt, and accelerate release cycles by migrating towards a microservices architecture, thereby enabling more agile and scalable development practices.

Modernization Strategy: The migration followed an incremental and iterative approach based on Domain-Driven Design (DDD) principles. The team employed static code analysis tools such as SonarQube to quantify technical debt and identify high-risk, tightly coupled modules. These modules were prioritized for decomposition and refactoring. The Strangler Fig pattern was used to gradually extract microservices around bounded contexts, avoiding a full system rewrite.

Challenges:

- Early iterations resulted in an excessive number of fine-grained microservices, which increased inter-service communication latency and operational overhead.

- The transition necessitated significant changes to the continuous integration and continuous delivery (CI/CD) pipeline to accommodate independent service deployments and integration testing.
- Managing data consistency and distributed transactions across services presented complex technical hurdles.
- Team reorganization and upskilling were required to handle the microservices paradigm effectively.

Outcomes: After migration, the team observed a 23% reduction in technical debt as measured by SonarQube metrics, particularly a decrease in code smells and architectural violations. Deployment frequency increased by approximately 35%, reflecting improved agility and team autonomy. Moreover, the incremental approach minimized business disruption and enabled continuous delivery of new features.

Takeaways: The study highlights the importance of careful service granularity design to prevent microservice sprawl and its associated complexity. Incremental migration strategies effectively balance risk and reward, allowing organizations to achieve modernization benefits without incurring prohibitive upfront costs. Additionally, investing in CI/CD tooling and team capabilities is critical to sustain the microservices ecosystem post-migration.

3.2.2 Case Study 2: Migration Roadmap Based on Field Experience (Wolfart et al., 2021)

Context: Multiple mid-sized companies in logistics and manufacturing sectors undertook service decomposition projects between 2018–2020.

3.2.3 Case Study 2: Migration Roadmap Based on Field Experience (Wolfart et al., 2021)

Context: Between 2018 and 2020, multiple mid-sized enterprises in the logistics and manufacturing sectors embarked on service decomposition and microservices migration initiatives. These companies were transitioning from monolithic legacy systems towards modular, scalable architectures to meet evolving business demands and regulatory compliance requirements.

Objective: The primary objective was to consolidate lessons learned from these heterogeneous migration projects into a structured, phased roadmap that could guide organizations with varying levels of cloud maturity and governance models through successful legacy modernization.

Modernization Strategy: Wolfart et al. proposed a migration roadmap grounded in empirical field experience with the following key components:

1. **Business Capability Mapping:** The initial phase involved comprehensive mapping of business capabilities to delineate functional domains, serving as a basis for modular decomposition.
2. **Identification of Bounded Contexts:** Utilizing domain-driven design principles, bounded contexts were defined to ensure clear service boundaries aligned with business processes.

3. **Implementation of APIs and Anti-Corruption Layers:** APIs were employed as the primary mechanism for service interaction, complemented by anti-corruption layers to isolate new microservices from legacy system constraints and avoid architectural erosion.

The methodology was deliberately flexible, accommodating different organizational maturity levels, regulatory constraints (e.g., industry-specific compliance), and technical landscapes.

Challenges:

- **Cultural Resistance:** Senior developers and engineers deeply familiar with the monolithic codebase exhibited resistance to architectural change, necessitating targeted change management and communication efforts.
- **Technical-Business Alignment Issues:** Initial decomposition efforts occasionally resulted in misalignment between technical service boundaries and actual business capabilities, leading to rework and coordination overhead.

Outcomes: Pilot migration projects applying this roadmap demonstrated a 2 to 3 times increase in release velocity, evidencing significant improvements in development agility. Furthermore, alignment between product management and engineering teams was enhanced, fostering better prioritization and collaborative planning.

Takeaways: Achieving enterprise-wide buy-in and emphasizing capability-centric service design were critical success factors. Additionally, from a cloud engineering perspective, it was imperative to establish robust observability frameworks and rollback mechanisms prior to service rollout to mitigate operational risks. This proactive infrastructure readiness was instrumental in maintaining system reliability throughout the migration process.

This case study illustrates how empirical insights can be synthesized into practical guidance, bridging organizational, technical, and operational dimensions essential for successful legacy system modernization.

3.2.4 Case Study 3: Architecture-Based Cloud Migration (Hasan et al., 2023)

Context: A leading Southeast Asian banking institution faced the challenge of modernizing its suite of legacy Java EE applications to meet evolving regulatory, cost-efficiency, and agility requirements. The legacy applications were tightly coupled with proprietary middleware and hosted on-premises, making cloud migration complex and high-risk.

Objective: The primary objective was to evaluate the suitability of refactoring versus rehosting (lift-and-shift) migration strategies to achieve compliance with stringent financial regulations, reduce operational expenditure, and increase deployment agility without disrupting critical banking services.

Modernization Strategy: The team employed an architecture-centric evaluation framework focusing on several dimensions:

- **Modularity and Service Granularity:** Services with high internal complexity or tightly coupled components were identified as candidates for refactoring.

- **Cloud Compatibility:** Systems were assessed for their ability to leverage cloud-native features such as container orchestration and serverless computing.
- **Regulatory Compliance:** Specific data residency and security constraints dictated deployment region limitations and data handling protocols.

Based on this assessment, the migration strategy was hybrid:

- Critical, complex services were refactored into microservices to improve modularity and scalability.
- Less complex or legacy-dependent components were rehosted using lift-and-shift to minimize immediate disruption and cost.

Challenges:

- **Middleware Dependencies:** Legacy reliance on proprietary middleware constrained refactoring options and required specialized adapters for cloud integration.
- **Data Residency and Multi-Region Constraints:** Regulatory mandates restricted data storage and processing to specific geographic regions, complicating cloud deployment architectures.
- **Testing and Validation Complexity:** Ensuring functional parity and compliance required extensive regression testing and verification in the cloud environment.

Outcomes: The hybrid migration approach yielded significant benefits, including:

- Approximately 40% reduction in infrastructure costs due to optimized resource utilization and cloud elasticity.
- Enhanced disaster recovery capabilities through cloud provider-managed failover and backup solutions.
- Improved observability and monitoring of services, facilitating proactive incident detection and resolution.
- Accelerated deployment cycles for refactored microservices, contributing to improved business agility.

Takeaways: This case study underscores the necessity of business impact modeling and architecture-aware decision-making to justify and guide selective refactoring efforts. It also highlights that cloud-native migration requires comprehensive architectural preparation beyond simple virtual machine rehosting, including redesign for modularity, compliance, and cloud service utilization. Tailoring migration strategies to organizational constraints and regulatory environments is critical to success in highly regulated industries such as banking.

This practical framework and hybrid approach provide valuable guidance for financial institutions facing complex legacy modernization challenges.

3.2.5 Case Study 4: Public Sector Modernization in a Bureaucratic Ecosystem (Tuusjärvi et al., 2024)

Context: A Nordic government agency undertook the modernization of its citizen record management system, which was previously implemented as a monolithic application based on legacy frameworks and technologies. The system was critical for delivering public services but suffered from slow response times, difficulty in compliance adaptation, and cumbersome release processes.

Objective: The modernization effort aimed to significantly improve service responsiveness to citizens and ensure full compliance with stringent EU data protection directives such as the General Data Protection Regulation (GDPR). Additionally, the agency sought to enable more agile delivery practices to accelerate feature rollout and operational improvements.

Modernization Strategy: The agency adopted a comprehensive strategy combining architectural, organizational, and process transformations:

- **Microservice Decomposition:** The monolithic system was decomposed into modular microservices based on bounded contexts, enabling independent deployment and scalability.
- **DevOps Enablement:** Implementation of continuous integration/continuous deployment (CI/CD) pipelines, automated testing, and infrastructure as code facilitated rapid and reliable deployments.
- **Organizational Restructuring:** A cross-functional team structure was established, fostering close collaboration between domain experts, developers, testers, and operations personnel.

This holistic approach recognized the socio-technical nature of modernization, emphasizing alignment between technical capabilities and organizational culture.

Challenges:

- **Procurement and Change Management Rigidity:** The public sector's stringent procurement policies and bureaucratic change request processes introduced delays and limited flexibility in adopting new tools and methodologies.
- **IT-Domain Misalignment:** Discrepancies between IT staff's technical skills and domain-specific knowledge created gaps in understanding service requirements and regulatory implications.
- **Legacy Code Complexity:** Mapping legacy code to microservices and establishing appropriate test scaffolding required substantial effort due to poor documentation and tightly coupled components.

Outcomes: Post-modernization, the agency achieved:

- A shift from a biannual (6-month) release cycle to monthly, more predictable releases, enhancing responsiveness to user needs.
- An approximate 45% improvement in system responsiveness, contributing to better user satisfaction and operational efficiency.

- Successful completion of regulatory compliance audits with significantly fewer exceptions, demonstrating improved adherence to data protection requirements.

Takeaways: The case underscores the criticality of socio-technical alignment in modernization initiatives. From an engineering perspective, rigorous legacy code analysis and automated test scaffolding were essential foundations for reliable migration. From a management standpoint, flexibility in procurement and change management processes emerged as key enablers of agility. Together, these factors facilitated a sustainable transition to a modern, cloud-native architecture within a complex bureaucratic environment.

This example highlights the importance of integrating organizational change with technical modernization to achieve transformative outcomes in public sector legacy system projects.

3.2.6 Case Study 5: LLM-Based COBOL Modernization (Geng et al., 2024)

Context: A prominent financial services provider relied on a legacy COBOL-based system to process core transactional workflows essential for daily operations. Over time, the organization faced increasing challenges including a dwindling pool of COBOL developers, rising maintenance expenses, and the risk of operational disruptions due to the aging technology stack.

Objective: The modernization initiative aimed to leverage Large Language Models (LLMs) to semi-automate the translation of COBOL source code into modern, object-oriented Java code, thereby accelerating the migration process while preserving business logic correctness and system traceability.

Modernization Strategy: The approach employed a domain-adapted LLM fine-tuned on a curated dataset of COBOL and corresponding Java code segments. Key aspects of the strategy included:

- **Domain-Specific Prompt Engineering:** Carefully crafted prompts incorporated domain knowledge to guide the LLM in accurately translating procedural COBOL constructs into equivalent Java object-oriented paradigms.
- **Incremental Code Conversion:** The legacy codebase was segmented into manageable chunks, allowing iterative translation and verification.
- **Validation Process:** Automated regression testing ensured functional equivalence between legacy and migrated components, supplemented by domain expert reviews for semantic accuracy.

Challenges:

- **Semantic and Paradigm Differences:** The procedural nature of COBOL posed inherent difficulties in mapping to object-oriented design patterns, necessitating nuanced translation strategies.
- **Limited Training Data:** Scarcity of paired COBOL-Java datasets, particularly for complex or less common COBOL constructs, constrained model generalization.
- **Error Handling and Edge Cases:** Handling legacy error management and batch processing logic required specialized attention to preserve system robustness.

Outcomes: The LLM-assisted modernization resulted in the successful migration of approximately 68% of the legacy codebase with minimal manual post-processing. This accelerated timeline contributed to a 40% reduction in overall development and maintenance costs. The approach also maintained high traceability, facilitating compliance audits and ongoing system evolution.

Takeaways: The study underscores the critical role of prompt engineering and domain-specific model fine-tuning in bridging legacy procedural paradigms and modern object-oriented architectures. From a managerial perspective, the integration of LLMs significantly shortened project durations without compromising functional fidelity or traceability. This case demonstrates the viability of LLM-driven modernization for mission-critical financial applications with legacy dependencies.

The findings encourage further exploration of AI-assisted techniques to overcome traditional modernization bottlenecks, especially in legacy languages with limited developer communities.

3.2.7 Case Study 6: LLM-Assisted Migration at Scale in Google (Ziftci et al., 2025)

Context: Google launched a large-scale internal initiative to modernize legacy codebases spanning multiple product teams and services. The scope included hundreds of millions of lines of code written in various languages and frameworks, many of which were critical to Google's core operations.

Objective: The primary goal was to automate the migration process of legacy code with the aid of Large Language Models (LLMs) while maintaining high standards of code quality and minimizing disruptions through a human-in-the-loop quality assurance approach.

Modernization Strategy: Google integrated fine-tuned LLMs directly into developers' Integrated Development Environments (IDEs). These models provided real-time code suggestions and partial auto-completions specifically aimed at migration tasks, such as API updates, language upgrades, and architectural refactoring. Key elements of the strategy included:

- **Continuous Feedback Loop:** Developer feedback on LLM-generated suggestions was systematically collected and used to iteratively fine-tune the models, enhancing their accuracy and relevance over time.
- **Incremental Migration:** Code changes were proposed and reviewed incrementally to prevent large-scale integration failures.
- **Observability and Rollback:** Advanced monitoring and rollback mechanisms were employed to quickly detect and revert any disruptive code changes.

Challenges:

- **Preserving Workflow Integrity:** Ensuring that LLM suggestions did not introduce regressions or break critical workflows required robust validation and developer vigilance.

- **Scaling Across Teams:** Managing feedback and adoption at scale, across thousands of developers and disparate codebases, presented organizational and technical complexities.

- **Trust Building:** Gaining developer trust necessitated embedding LLM tools seamlessly into existing workflows rather than imposing separate migration processes.

Outcomes: The program successfully migrated over 3 million lines of legacy code with a significant improvement in merge success rates, which increased by 22%. Developer productivity saw substantial gains due to reduced manual effort and improved migration accuracy. Furthermore, the iterative feedback mechanism led to continuous model performance improvements.

Takeaways: The case illustrates that the effectiveness of LLMs at scale is contingent upon comprehensive observability frameworks, reliable rollback capabilities, and fostering trust through integration into familiar developer environments. Organizational buy-in was higher when migration assistance tools complemented rather than disrupted established workflows. This exemplifies best practices for deploying AI-assisted modernization at enterprise scale.

This example serves as a pioneering model for LLM-powered large-scale software migration, demonstrating practical benefits and challenges in real-world settings.

3.3 Analysis of Case Studies

3.3.1 Success Factors in Industrial Modernization

Domain-Driven Decomposition and Architectural Refactoring A common success pattern across multiple cases is the use of domain-driven design (DDD) principles to guide decomposition. Lenarduzzi et al. [6] and Wolfart et al. [5] show that identifying bounded contexts and mapping them to services helped teams transition from tightly coupled monoliths to modular systems. This was often combined with the Strangler Fig pattern, allowing legacy and new systems to coexist until full migration was complete.

CI/CD Integration and Testing Infrastructure Automated testing and CI/CD adoption played a critical role in de-risking migration. By integrating unit, integration, and contract testing into pipelines, organizations were able to detect issues earlier and accelerate release cycles. In particular, Wolfart et al. reported a 2–3x improvement in deployment frequency post-adoption of CI/CD.

AI-Assisted Refactoring and Migration Acceleration Geng et al. [24] demonstrated that LLMs can be effectively leveraged for COBOL-to-Java migration, significantly reducing manual effort. Their approach combined prompt engineering with regression testing, enabling safe and explainable migration. The 30% time reduction they report illustrates how generative AI can support modernization at scale when tightly coupled with validation mechanisms.

Stakeholder Engagement and Organizational Alignment Successful cases consistently emphasize early and sustained involvement of business stakeholders. For example,

Wolfart et al. engaged product owners to validate service boundaries, reducing the likelihood of architectural misalignment. Lenarduzzi et al. highlight that team autonomy and shared ownership were instrumental in sustaining modernization momentum.

3.3.2 Failure Patterns and Organizational Pitfalls

Organizational Siloing and Knowledge Decay Tuusjärvi et al. [11] reveal that fragmentation between teams—especially between domain experts and developers—led to flawed service decomposition and misaligned APIs. Over time, undocumented legacy behavior became a bottleneck due to loss of tacit knowledge.

Over-Centralized Decision-Making and Lack of Empowerment Top-down mandates without developer buy-in often led to low adoption of modernization tooling or architectural decisions. In several instances, developers reverted to monolithic behaviors within microservice contexts due to lack of understanding or perceived complexity.

Insufficient Training and Unrealistic Timelines Ogunwole et al. [15] describe modernization efforts where teams were expected to adopt cloud-native tooling and IaC without formal training, leading to technical debt accrual in supposedly "modernized" systems. Aggressive timelines further compromised testing and rollout safety.

Missing Governance and Accountability Structures In many failed cases, there was no modernization steering committee or architectural review board to validate milestones, manage risks, or escalate issues. This absence created project drift and scope creep, especially in public-sector contexts with fixed budgets.

3.3.3 Metrics for Modernization Effectiveness

Technical and Engineering Metrics

- **Technical Debt Reduction**: Tracked using tools like SonarQube, measuring maintainability index, duplicated code, and code complexity.
- **Build and Deployment Metrics**: Deployment frequency, lead time to change, and change failure rate, often captured through CI/CD pipelines.
- **Code Churn and Module Coupling**: Indicators of improved modularity post-migration.

Business and User-Centric Metrics

- **Time to Feature Delivery**: Evaluated against pre-modernization release cycles.
- **Cost and Time Savings**: Measured in labor hours, infrastructure spend, and license cost reduction.
- **User Satisfaction and UX Feedback**: Assessed through NPS surveys, usability testing, and ticket volume tracking.

Reliability and Stability Indicators

- **Downtime and Error Rates**: Measured as part of SLO/SLI frameworks.
- **Mean Time to Recovery (MTTR)**: A key metric for modernization impact on operability.
- **Merge Success and Rollback Rates**: As tracked in Google's large-scale LLM-assisted migrations [36].

3.3.4 Cross-Case Comparisons: COBOL, Java EE, and FinTech Systems

COBOL-Based Systems Geng et al. [24] show that COBOL systems, while challenging due to procedural logic and monolithic data handling, are well-suited to LLM-based refactoring if domain-specific prompts and legacy test suites are available. Success depends on system stability and test coverage.

Java EE Systems Hasan et al. [3] indicate that Java EE systems often suffer from tight coupling, legacy EJBs, and outdated ORM frameworks. They benefit more from rearchitecting into microservices or serverless platforms rather than lift-and-shift approaches. Architectural debt was the primary blocker.

FinTech Environments In FinTech systems, Singh [14] emphasizes the importance of formal verification and compliance-aware logging. Migration in these contexts requires audit trails, transaction consistency mechanisms, and rollback testing—highlighting the additional burden posed by regulatory constraints.

3.3.5 Synthesized Lessons and Best Practices

Iterative, Measurable Modernization Large-scale rewrites almost universally led to budget overruns or delivery failures. Incremental modernization, guided by dependency analysis and feature slicing, yielded more resilient outcomes.

Tooling and Automation Integration Effective modernization efforts were underpinned by automated testing, IaC, LLM-assisted code review, and centralized observability stacks. These tools allowed teams to balance velocity with quality.

Human Factors and Knowledge Management Soft elements—such as cross-functional workshops, legacy walkthroughs, and mentorship—were essential to reduce rework and accelerate onboarding. Capturing business rules embedded in code was a recurring theme.

Governance and Feedback Loops Modernization programs with defined checkpoints, architectural reviews, and stakeholder feedback channels performed better in terms of scope control and technical alignment.

3.4 Deduction

Empirical case studies provide concrete evidence that modernization is as much about people and process as it is about technology. Although technical strategies and LLM-based automation show promise in reducing modernization, their success depends on governance, training, cross-functional collaboration, and careful tracking of metrics. The diversity of domains—from COBOL to FinTech—underscores the need for contextual strategies, not one-size-fits-all solutions. Future research should focus on benchmarking frameworks, migration maturity models, and open-source case repositories to enhance cross-industry knowledge transfer.

4 Toward a Unified Modernization Reference Framework

While this review has presented a modular view of modernization across six functional domains—strategy, architecture, cloud infrastructure, LLMs, UI, and tooling—the lack of an integrated framework risks fragmenting the modernization discourse. A critical gap identified across the surveyed literature is the absence of a unifying model that aligns technical patterns, decision-making processes, and automation capabilities into a cohesive modernization lifecycle [1, 5].

4.0.1 Motivation for a Meta-Framework

Legacy system modernization involves not just isolated refactoring tasks, but a complex interplay of strategic goals, architectural evolution, infrastructure replatforming, cognitive tooling, and user-centric redesign. Multiple studies emphasize this cross-domain dependency:

- Strategic planning (e.g., using FUCOM-WSM) must guide architectural prioritization and resource allocation [2].
- Architectural migration decisions affect cloud migration patterns and infrastructure provisioning [3, 11].
- LLM-based tools influence both code-level transformation and UI design pipelines [24, 28].
- Toolchains and CI/CD systems act as the operational backbone connecting strategy with execution [36].

Without an overarching process model, organizations risk local optimization and siloed implementations, reducing modernization coherence and increasing rework.

4.0.2 Proposed Framework Structure

We propose a conceptual *Modernization Reference Framework* (MRF) that defines a layered structure to unify the domains reviewed in this paper. The MRF consists of four horizontal layers and two vertical planes:

- **Layer 1: Strategy and Decision Models** — Incorporates business goals, risk analysis, MCDM frameworks, and governance policies [1, 15].
- **Layer 2: Architectural Refactoring and Domain Modeling** — Focuses on decomposition patterns, service granularity, and legacy code evaluation [5, 6].
- **Layer 3: Cloud-Native Enablement and Infrastructure Automation** — Includes IaC, container orchestration, serverless, observability, and platform abstraction [7, 27].
- **Layer 4: Cognitive Augmentation and UI Modernization** — Encompasses LLM-based refactoring, accessibility-aware UI design, and conversational agent integration [29, 32].

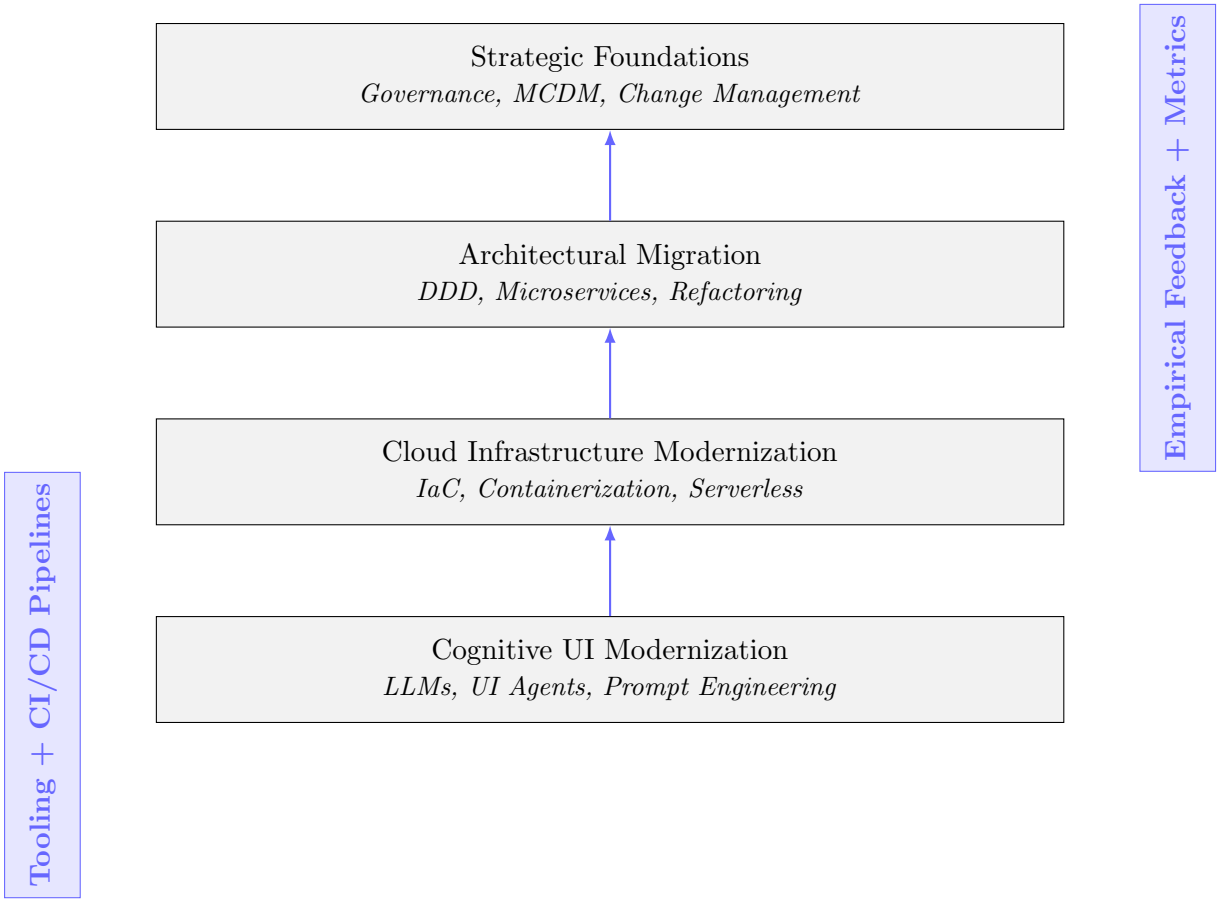

Figure 1: Modernization Reference Framework: Functional Layers with Cross-Cutting Concerns

Two vertical planes cut across all layers:

- **Tooling and CI/CD Pipelines** — Acts as the execution substrate, integrating code analysis, automated testing, rollback support, and human-in-the-loop validation [28, 35].
- **Empirical Feedback and Metrics** — Supports iteration and evaluation through modernization KPIs, technical debt monitoring, and user satisfaction benchmarks [36].

4.0.3 Benefits of the Framework

This reference architecture offers several advantages:

- Provides a roadmap for staged modernization planning.
- Aligns enterprise-wide initiatives with team-level refactoring decisions.
- Encourages tool interoperability and platform reuse across domains.
- Enhances governance, traceability, and compliance via standardized observability.

Future work could operationalize this framework by mapping existing tools, platforms, and methodologies into each layer. Empirical validation could involve testing the framework across multiple industries with varying modernization maturity levels.

4.1 Limitations in Evaluation and Benchmarking of LLM-Driven Modernization

Despite the growing interest in large language models (LLMs) for software modernization, rigorous validation and benchmarking remain underdeveloped in many studies. While several works propose tools for code refactoring [17, 24] and UI generation [28, 29], few provide reproducible evaluation pipelines or standardized benchmarks to assess model performance across real-world legacy systems.

4.1.1 Validation of Code Transformation Outputs

In the context of LLM-assisted code transformation, model outputs must be evaluated not only for syntactic correctness but also for functional and architectural fidelity. A few recent studies have attempted to introduce structured validation:

- **Regression Test Suites**: Projects like LLMigrate [35] integrate legacy test cases and post-migration tests to verify behavioral equivalence. However, many legacy systems lack complete test coverage, which undermines reliability.
- **Semantic Equivalence Metrics**: Ziftci et al. [36] introduced the $LLM\Delta$ and $Human\Delta$ metrics—based on normalized Levenshtein distances—to compare human and model-generated edits. This approach, while scalable, focuses on surface-level similarity and may not detect subtle architectural regressions.
- **Compile-Time and Runtime Validation**: Systems like UICoder [28] incorporate static analysis and UI rendering feedback into the model fine-tuning loop, increasing trust in autogenerated code. Yet such techniques are currently domain-specific and non-generalizable.

4.1.2 Benchmarking Gaps in UI and Interaction Design

The evaluation of LLM-based UI generation and conversational agents faces additional challenges due to the subjective and multimodal nature of user experience (UX). The emergence of dedicated benchmarks is recent and fragmented:

- **UXAgent Benchmark Suite**: Zhao et al. [32] provide a structured benchmark for evaluating accessibility, responsiveness, and adherence to UX best practices in LLM-generated UIs. Their framework includes automated heuristics and human-in-the-loop reviews, setting a precedent for measurable UX quality.
- **Crowdsourced Preference Models**: Liu et al. [29] enhance benchmarking by incorporating user preference libraries into generation tasks. This model supports explainability and diversity but lacks integration with standard UI testing frameworks (e.g., Lighthouse, axe-core).
- **Lack of General-Purpose Datasets**: Unlike traditional NLP domains, there is no large-scale public dataset of legacy UI designs and their modern equivalents, which hinders reproducibility and transfer learning.

4.1.3 Challenges and Recommendations

- **Reproducibility**: Most current studies do not publish their prompts, transformation criteria, or test oracles. This limits comparability and reusability.
- **Cross-Domain Validation**: Models fine-tuned on FinTech or government codebases may not generalize to manufacturing or healthcare, yet this is rarely tested.
- **Hybrid Evaluation Models**: Future tools should combine static analysis (e.g., cyclomatic complexity, code smells), behavioral testing (e.g., mutation testing), and human judgment (e.g., maintainability and UX review).

4.1.4 Call for Standardization

Schmid et al. [16] note that the field lacks canonical benchmarks or maturity models for LLM-driven architecture and modernization. Drawing from their systematic literature review, we argue for the development of:

- A public benchmark suite for LLM-assisted software migration (e.g., including COBOL, JEE, .NET systems).
- UX evaluation datasets linked to accessibility and usability standards (e.g., WCAG).
- Versioned prompts and CI-integrated evaluation pipelines to enable reproducible research.

Such resources would improve comparability across studies and foster responsible, metrics-driven deployment of AI in modernization workflows.

4.2 Ethical and Operational Risks in AI-Augmented Modernization

As LLMs become integrated into the modernization toolchain—supporting tasks ranging from source code refactoring to UI generation and architecture recovery—their use in mission-critical environments introduces a new class of risks. These challenges are particularly acute in domains such as finance, healthcare, and public infrastructure, where reliability, compliance, and auditability are non-negotiable.

4.2.1 Explainability and Trustworthiness

LLMs, especially large autoregressive models like GPT and Codex, operate as black-box systems. Unlike rule-based refactoring tools, their decision-making is not inherently interpretable. Geng et al. [24] and Talasila et al. [17] both rely on prompt-engineered workflows without formal explanations of transformations. This lack of transparency complicates debugging, auditing, and compliance certification.

4.2.2 Hallucination and Output Drift

Model hallucination—the generation of syntactically correct but semantically invalid or unsafe code—is a well-documented risk in LLM-driven software engineering [35,36]. Hallucinations are especially problematic in legacy modernization, where the systems being refactored may lack test coverage or documentation. Additionally, model drift—where outputs change as the model is updated—undermines reproducibility and long-term validation unless outputs are versioned or snapshotted.

4.2.3 Bias and Domain Misalignment

General-purpose LLMs are trained on broad corpora and may lack awareness of domain-specific constraints, regulatory requirements, or security guidelines. In FinTech systems, Singh [14] notes that transaction logic and auditability constraints are often violated by auto-generated suggestions unless they are manually validated. Similarly, UI generation tools such as UXAgent [32] or CrowdGenUI [29] may inadvertently exclude accessibility requirements or misalign with compliance frameworks such as WCAG unless specifically prompted.

4.2.4 Human-in-the-Loop Safeguards

A recurring theme in successful industrial applications (e.g., Google's migration platform [36], UICoder [28]) is the presence of human-in-the-loop (HITL) governance. These workflows combine LLM output generation with:

- Rule-based static analysis
- Regression and integration testing
- Manual review and semantic validation checkpoints

This hybrid model improves confidence, reduces the cost of verification, and aligns with emerging guidelines for AI reliability and responsible deployment.

4.2.5 Implications for Research and Practice

We advocate for future modernization workflows to include explicit risk modeling steps that incorporate:

- Explainability scores or natural language rationales
- Confidence estimation based on domain calibration
- Version control of prompts and model snapshots

These features will not only mitigate risk but also enable traceable and auditable modernization pipelines, particularly in high-assurance systems.

4.3 Clarity and Redundancy in Presentation

While the modular structure of this review facilitates targeted discussion of key modernization domains, a closer editorial examination reveals instances of redundancy and structural inconsistency that may hinder readability and narrative flow.

4.3.1 Duplicated Sections and Heading Artifacts

During document structuring, certain headings—specifically, `Section 2.8` and `Section 2.9` on Architectural Migration—were inadvertently duplicated. While both sections cover important themes such as service decomposition and technical debt remediation, their content overlaps significantly and may confuse readers regarding scope boundaries. Consolidating these into a single, unified section would improve structural coherence and eliminate redundancy.

4.3.2 Repetition in Timeline Subsections

Across multiple domains, the "Timeline of Important Papers" subsections tend to reiterate bibliographic details already captured in the analysis narrative. For example, author names, dates, and venue descriptions are often restated both in timeline bullet points and in subsequent paragraphs. This redundancy could be minimized by streamlining the timeline entries to focus strictly on key contribution summaries (e.g., *"Empirical study on microservice refactoring effectiveness"*) and deferring detailed exposition to the analysis section.

4.3.3 Suggested Editorial Actions

To enhance clarity and conciseness, we recommend the following:

- **Merge overlapping sections:** Reconcile content duplication across architectural migration discussions and avoid redundant Deductions.
- **Condense timelines:** Limit timeline subsections to 2–3 lines per paper, highlighting only the main contribution and domain relevance.
- **Cross-reference effectively:** Use internal references (e.g., `See Section ??`) to guide readers instead of repeating content.
- **Review LaTeX structure:** Ensure consistent section numbering, label usage, and adherence to IEEE or journal formatting guidelines.

Such revisions will significantly reduce cognitive load, prevent information fatigue, and enhance the overall readability of an otherwise well-organized manuscript.

4.4 Toward a Taxonomy of Modernization Tooling Frameworks

While this review highlights multiple innovations in AI-assisted modernization, a cohesive taxonomy of available tools is needed to guide practitioners in selecting and orchestrating modernization pipelines. Tools vary widely in terms of their scope, integration level, and domain specificity—ranging from standalone static analyzers to tightly embedded CI/CD agents and LLM-augmented design environments.

Building on reviewed literature [24, 28, 32, 35, 36], we propose a four-category classification:

1. **Static Analysis and Refactoring Tools** — Perform code audits, detect smells, and suggest transformations (often rule-based).
2. **CI/CD-Integrated Engines** — Embed transformation agents in pipelines with validation, rollback, and testing hooks.
3. **IDE-Integrated LLM Plugins** — Provide contextual code suggestions, explanations, and refactoring within developer environments.
4. **UI and Interaction Design Assistants** — Use LLMs and feedback models to generate and evaluate user interfaces.

4.4.1 Comparison of Tool Capabilities

Table 1: Comparison of Modernization Tools and Frameworks

Tool/Framework	Type	LLM-Based	CI/CD Integration	Domain Focus
SonarQube	Static Analysis	✗	Partial (via CLI)	General code quality
LLMigrate [35]	Refactoring Engine	✓ (Fine-tuned)	✓	Codebase migration (e.g., COBOL)
GitHub Copilot	IDE Plugin	✓ (Codex-based)	✗	General code generation
UICoder [28]	UI Generator	✓ (LLM + Feedback Loop)	✗	SwiftUI, front-end layout
CrowdGenUI [29]	UI Assistant	✓ (Preference Model)	✗	Python/Jupyter UIs
UXAgent [32]	UX Evaluator	✓ (Benchmark + Agent)	Partial	Accessibility and responsiveness
Google LLM Migrator [36]	CI Agent	✓ (Internal LLM)	✓ (End-to-End)	Cross-language migration
ARCHIMINER	Architecture Recovery	✗	✗	Java systems (baseline)
AutoArch [35]	Arch. Inference	✓ (GPT-4)	✗	Architecture recovery from code

4.4.2 Observations and Implications

This comparison reveals that:

- LLM-powered tools dominate IDE and UI generation domains, but are less common in CI/CD pipelines due to risk and explainability concerns.
- CI-integrated agents (e.g., Google's LLM migration platform) offer the strongest operational maturity, but require enterprise-scale infrastructure.
- UI assistants like UICoder and UXAgent offer specialized value in accessibility and feedback-driven improvement, but lack integration with enterprise design systems.

Future tooling frameworks should prioritize composability (e.g., plug-and-play microservices), governance (e.g., traceable transformations), and interoperability across DevSecOps pipelines. Establishing standard APIs and benchmark tasks would further support empirical tool comparison and reproducible modernization workflows.

5 Conclusion

Modernizing legacy systems is no longer a peripheral concern—it is a strategic imperative for organizations seeking agility, resilience, and long-term sustainability in the digital era. This paper has presented a structured, domain-oriented review of the evolving landscape

of legacy application modernization, synthesizing best practices, decision frameworks, technological enablers, and empirical evidence across industries.

At the strategic level, we find that modernization initiatives are most successful when anchored in enterprise objectives, guided by multi-criteria decision-making frameworks such as FUCOM-WSM, and informed by both technical debt analysis and organizational readiness. Architectural migration—particularly from monoliths to microservices—emerges as a foundational pillar, but its success depends on domain-driven design, governance, and cross-functional collaboration.

Cloud migration and infrastructure modernization are facilitated by technologies such as Infrastructure-as-Code, containers, and serverless platforms, yet they also introduce new challenges in observability, vendor lock-in, and data architecture redesign. Large Language Models (LLMs) are beginning to transform the modernization process by enabling prompt-based code transformation, migration support, and CI/CD integration. Their role is especially evident in UI modernization, where tools like CrowdGenUI and UXAgent support accessible, adaptive, and user-centric interface design.

Empirical case studies reveal consistent success factors: incremental migration, stakeholder alignment, robust tooling ecosystems, and formal governance structures. Conversely, failure often stems from siloed teams, inadequate training, unrealistic timelines, and poor change management. Across domains—from COBOL to FinTech—modernization strategies must be adapted to system complexity, regulatory context, and organizational culture.

Looking ahead, several future research directions are evident. These include:

- **Explainable and trustworthy AI**: Ensuring LLM outputs are verifiable, auditable, and reliable in critical systems.
- **Human-in-the-loop augmentation**: Combining AI capabilities with expert oversight to balance automation with safety.
- **Domain-specific adaptation**: Fine-tuning tools and processes to suit sector-specific needs, especially in regulated environments.
- **Metrics and benchmarking frameworks**: Establishing standardized ways to assess modernization progress, quality, and ROI.

In sum, modernization is not a single event but a multi-dimensional, continuous journey—technically complex, organizationally sensitive, and increasingly AI-augmented. By embracing modular strategies, empirical insights, and intelligent tooling, organizations can navigate this journey more confidently and systematically.

References

[1] W. K. G. Assunção, L. Marchezan, L. Arkoh, A. Egyed, and R. Ramler, "Contemporary Software Modernization: Strategies, Driving Forces, and Research Opportunities," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, Art. 142, May 2025. [Online]. Available: `https://doi.org/10.1145/3708527`

[2] N. Jomhari *et al.*, "A Multi-Criteria Decision-Making for Legacy System Modernization With FUCOM-WSM Approach," *IEEE Access*, vol. 12, pp. 48608–48625, Apr. 2024. [Online]. Available: `https://doi.org/10.1109/ACCESS.2024.3383917`

[3] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "Legacy systems to cloud migration: A review from the architectural perspective," *J. Syst. Softw.*, vol. 202, p. 111702, Apr. 2023. [Online]. Available: https://doi.org/10.1016/j.jss.2023.111702

[4] S. Krishnan *et al.*, "Incremental Analysis of Legacy Applications Using Knowledge Graphs for Application Modernization," in *Proc. 9th ACM IKDD CODS and 27th COMAD*, Bangalore, India, Jan. 2022. [Online]. Available: https://doi.org/10.1145/3493700.3493735

[5] D. Wolfart *et al.*, "Modernizing Legacy Systems with Microservices: A Roadmap," in *Proc. EASE*, Trondheim, Norway, Jun. 2021. [Online]. Available: https://doi.org/10.1145/3463274.3463334

[6] V. Lenarduzzi *et al.*, "Does Migrating a Monolithic System to Microservices Decrease the Technical Debt?," *J. Syst. Softw.*, vol. 169, p. 110710, Jul. 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110710

[7] M. Fahmideh *et al.*, "Challenges in Migrating Legacy Software Systems to the Cloud: An Empirical Study," *Inf. Syst. J.*, vol. 29, no. 5, pp. 878–917, 2019. [Online]. Available: https://doi.org/10.1111/isj.12192

[8] J. Fritzsch *et al.*, "Microservices Migration in Industry: Intentions, Strategies, and Challenges," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 4420–4461, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09815-0

[9] V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," *IEEE Access*, vol. 11, pp. 88339–88354, Aug. 2023. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3305687

[10] H. Mili *et al.*, "Service-Oriented Re-engineering of Legacy JEE Applications: Issues and Research Directions," *arXiv preprint arXiv:1906.00937*, Jun. 2019. [Online]. Available: https://arxiv.org/abs/1906.00937

[11] K. Tuusjärvi, J. Kasurinen, and S. Hyrynsalmi, "Migrating a Legacy System to a Microservice Architecture," *e-Informatica Softw. Eng. J.*, vol. 18, no. 1, Art. 240104, 2024. [Online]. Available: https://www.e-informatyka.pl/index.php/einformatica/volumes/volume-2024/issue-1/article-4/

[12] J. Fritzsch *et al.*, "Towards an Architecture-centric Methodology for Migrating to Microservices," *arXiv preprint arXiv:2207.00507*, Jul. 2022. [Online]. Available: https://arxiv.org/abs/2207.00507

[13] R. Dachepally, "Modernizing Legacy Systems with Microservices: A Roadmap for Digital Transformation," *Int. J. Innov. Res. Manag. Eng. Technol.*, vol. 7, no. 4, Jul.–Aug. 2019. [Online]. Available: https://www.ijirmps.org/

[14] P. Singh, "Modernizing Legacy FinTech Systems through Cloud-Native Microservices Architecture," *Int. J. Sci. Technol.*, vol. 15, no. 1, pp. 1–6, Jan.–Mar. 2024. [Online]. Available: https://www.ijsat.org/

[15] O. Ogunwole *et al.*, "Modernizing Legacy Systems: A Scalable Approach to Next-Generation Data Architectures and Seamless Integration," *Int. J. Multidiscip. Res. Growth Eval.*, vol. 4, no. 1, pp. 901–909, Jan.–Feb. 2023. [Online]. Available: https://doi.org/10.54660/IJMRGE.2023.4.1.901-909

[16] L. Schmid *et al.*, "Software Architecture Meets LLMs: A Systematic Literature Review," *arXiv preprint arXiv:2505.16697*, 2025. [Online]. Available: https://arxiv.org/abs/2505.16697

[17] R. Talasila, J. Anderson, and K. Xu, "Modernizing Monolithic Applications with Language Models," *arXiv preprint arXiv:2309.03796*, Sep. 2023. [Online]. Available: https://arxiv.org/abs/2309.03796

[18] S. Khan, N. Jain, and A. Kumar, "Architectural Patterns for Legacy Systems," *arXiv preprint arXiv:2306.15792*, Jun. 2023. [Online]. Available: https://arxiv.org/abs/2306.15792

[19] A. Bhoopalam *et al.*, "Pre-Trained Models for Cloud-Native Refactoring," *arXiv preprint arXiv:2309.16739*, Sep. 2023. [Online]. Available: https://arxiv.org/abs/2309.16739

[20] Y. Sato, A. Klein, and D. Zhang, "LLM Code Completion and Migration Support," *arXiv preprint arXiv:2405.13333*, May 2024. [Online]. Available: https://arxiv.org/abs/2405.13333

[21] T. Müller *et al.*, "Pattern-Based Migration of Software Systems," in *Proc. of the 30th International Conference on Software Analysis*, 2023.

[22] R. Khadka *et al.*, "Digital Transformation Roadmap for Legacy Modernization," *Applied System Innovation*, vol. 5, no. 4, 2022. [Online]. Available: https://www.mdpi.com/2571-5577/5/4/86

[23] H. Klenk and T. Berger, "Model-Driven Migration Approaches: A Systematic Review," *arXiv preprint arXiv:1908.10337*, Aug. 2019. [Online]. Available: https://arxiv.org/abs/1908.10337

[24] Y. Geng *et al.*, "Modernizing COBOL Systems with Generative AI," *arXiv preprint arXiv:2405.13333*, May 2024. [Online]. Available: https://arxiv.org/abs/2405.13333

[25] J. Singh and N. Chauhan, "SOA-Based Reengineering of Legacy Software," *Int. J. Comput. Appl.*, vol. 159, no. 6, pp. 1–6, Feb. 2017.

[26] A. Moryossef, Y. Belinkov, and R. Shwartz, "LLMs in Software Architecture and Migration," *arXiv preprint arXiv:2309.16739*, Sep. 2023. [Online]. Available: https://arxiv.org/abs/2309.16739

[27] S. Yousef, M. Elsisi, and A. Abdelrahman, "Cognitive Cloud for Legacy Modernization," *Future Internet*, vol. 16, no. 2, Feb. 2024. [Online]. Available: https://www.mdpi.com/1999-5903/16/2/45

[28] J. Wu, E. Schoop, A. Leung, T. Barik, J. P. Bigham, and J. Nichols, "UICoder: Finetuning Large Language Models to Generate User Interface Code through Automated Feedback," *arXiv preprint arXiv:2406.07739*, Jun. 2024. [Online]. Available: https://arxiv.org/abs/2406.07739

[29] Y. Liu, M. Sra, and C. Xiao, "CrowdGenUI: Enhancing LLM-Based UI Widget Generation with a Crowdsourced Preference Library," *arXiv preprint arXiv:2411.03477*, Nov. 2024. [Online]. Available: https://arxiv.org/abs/2411.03477

[30] J. Park, J. Oh, H. Yoon, and J. Lee, "LLM4UI: Towards User Interface Generation with Large Language Models," *arXiv preprint arXiv:2405.13050*, May 2024. [Online]. Available: https://arxiv.org/abs/2405.13050

[31] Y. Lu, M. Cai, J. Lin, and Y. Shi, "NavI: Semantic UI Design Search through Interactive Dialogue with LLMs," *arXiv preprint arXiv:2404.11072*, Apr. 2024. [Online]. Available: https://arxiv.org/abs/2404.11072

[32] H. Zhao *et al.*, "UXAgent: Benchmarking and Improving LLM-Based Agents for User Interface Design," *arXiv preprint arXiv:2403.13139*, Mar. 2024. [Online]. Available: https://arxiv.org/abs/2403.13139

[33] J. Chen, Y. Wang, and Y. Zhang, "AI-Chat2Design: Building Conversational Agents for UI Design via Multi-Task Learning," *arXiv preprint arXiv:2310.15435*, Oct. 2023. [Online]. Available: https://arxiv.org/abs/2310.15435

[34] Z. Guo *et al.*, "Design2Code: Exploring Large Language Models as User Interface Design Assistants," *arXiv preprint arXiv:2304.08103*, Apr. 2023. [Online]. Available: https://arxiv.org/abs/2304.08103

[35] Y. Liu, J. Hu, Y. Shan, G. Li, Y. Zou, Y. Dong, and T. Xie, "LLMigrate: Transforming 'Lazy' Large Language Models into Efficient Source Code Migrators," *Proc. ACM*, 2025. [Online]. Available: https://arxiv.org/abs/2503.23791

[36] C. Ziftci, S. Nikolov, A. Sjövall, B. Kim, D. Codecasa, and M. Kim, "Migrating Code At Scale With LLMs At Google," in *Proc. ACM FSE*, 2025. [Online]. Available: https://arxiv.org/abs/2504.09691