
AI-Assisted Legacy Transformation: Modernizing Mission-Critical .NET Applications Through Cloud Integration and API Governance

Zhao Zhu ¹, Mohammad Zbeeh ²

¹ American University of Beirut, LEBANON

² Sun Yat-sen University, CHINA

ABSTRACT

Mission-critical .NET Framework applications remain deeply embedded in enterprise and public-sector operations years after their originating platform stopped receiving new features, and the reasons they persist are the same reasons they resist modernization: undocumented business logic accumulated over decades, tightly coupled monolithic designs, and the operational risk of touching a system that cannot fail. Two developments have changed what a modernization program can attempt. First, cloud-native infrastructure gives legacy .NET estates a concrete migration target containerized runtimes, managed platform services, and governed API boundaries that was not available when many of these systems were built. Second, large-language-model-based coding assistants can now read, summarize, and transform undocumented legacy code far faster than a human engineer, but the same literature that documents their productivity gains also documents that their suggestions can be insecure, architecturally inconsistent, or simply wrong, and existing modernization guidance was written before such tools existed. This paper addresses the resulting gap by developing AI-Assisted Legacy Transformation (ALT): a reference architecture that uses large-language-model assistance for code comprehension, dependency mapping, and code transformation at each stage of a .NET-to-cloud modernization program, but routes every AI-generated artifact through a risk-tiered human review gate and an API governance layer before it reaches a mission-critical system. Because no production deployment of this specific architecture exists to evaluate, the paper adopts a design-science approach: the framework is built from and evaluated against peer-reviewed findings in legacy modernization, cloud migration, API governance, and AI-assisted software engineering, and every quantitative comparison presented — two tables and four figures spanning capability coverage, migration velocity, the velocity-defect trade-off, and risk reduction through governance — is explicitly labeled as a literature-grounded estimate rather than a measured result. The paper's contribution is a synthesis of four otherwise separate literatures into one modernization pipeline, an explicit mapping of modernization objectives to specific AI-assisted techniques and governance controls, and a candid account of where AI assistance helps a .NET modernization program and where it introduces risk that only human governance can absorb.

Keywords: Legacy System Modernization, .NET Framework Migration, Cloud Integration, API Governance, AI-Assisted Software Engineering, Large Language Models, Mission-Critical Systems

Introduction

Mission-critical .NET Framework applications are among the most durable artifacts in enterprise and public-sector computing, and their durability is not an accident of neglect but a rational response to risk. Lehman's (1980) laws of software evolution observed that a system in continuous use must keep changing to remain useful, yet each change increases its complexity unless deliberate effort reduces it — a dynamic that, left unmanaged for twenty or thirty years, produces exactly the kind of large, tightly coupled, sparsely documented .NET Framework estate that now sits at the center of claims processing, benefits administration, and core financial systems across government and industry. Bisbal *et al.* (1999) named the resulting condition directly: a legacy system is not defined by its age

but by the fact that it is vital to the organization and simultaneously resistant to the very maintenance and evolution it requires. Khadka *et al.*'s (2014) interviews with modernization professionals confirm that this resistance is not merely technical; practitioners report that undocumented business rules, fear of breaking a system nobody fully understands, and the sheer scale of a multi-decade codebase are the actual barriers to modernization, more so than any limitation of the tools available to rewrite the code (Potharaju & Sajja, 2024). A .NET Framework application processing benefit determinations or settling payments is, in this literal sense, exactly the system Lehman and Bisbal described: too important to leave alone and too risky to change carelessly (Rahman *et al.*, 2024).

Cloud computing has, over the same period, given modernization programs a concrete target architecture that did not exist when many of these systems were designed. Jamshidi *et al.*'s (2013) systematic review of cloud migration research maps the space of approaches, from a minimal lift-and-shift of an existing application onto cloud infrastructure through to a full architectural re-platforming that takes advantage of managed services, and Gholami *et al.*'s (2016) survey of the cloud migration process catalogs the recurring technical and organizational challenges — data migration risk, service dependency mapping, and the difficulty of validating that a re-architected system still behaves identically to the one it replaces — that any .NET Framework modernization program must confront regardless of which end of that spectrum it targets (Kabir *et al.*, 2024). Andrikopoulos *et al.* (2013) go further, showing that adapting an application for a cloud environment is frequently not a deployment decision but an architectural one: applications built around assumptions of a fixed, on-premises topology require structural change, not just a new hosting location, before they can take advantage of what cloud infrastructure actually offers. Chauhan and Babar's (2011) early experience report on migrating a service-oriented system to the cloud, documenting how much of the migration effort concerned reworking service boundaries and configuration assumptions rather than moving code (Rashid *et al.*, 2024).

Reworking those service boundaries is itself a specialized discipline, and one with its own peer-reviewed foundation independent of cloud migration. Zimmermann's (2017) enumeration of microservices tenets and Haupt *et al.*'s (2018) structural analysis of REST API design both treat the API surface between services as something to be governed deliberately — versioned, contractually specified, and reviewed for consistency — rather than allowed to emerge as an incidental byproduct of decomposition. Dragoni *et al.* (2017) trace how service decomposition without this discipline tends to reproduce the original monolith's coupling problems inside a new, distributed form, simply relocating the tangle rather than resolving it, while Fritzsche *et al.*'s (2019) classification of monolith-to-microservices refactoring approaches shows that the technical act of splitting code apart is the easier half of the problem; defining and enforcing the contracts between the resulting services is the half that determines whether the decomposition actually reduces long-term risk. A .NET Framework modernization program that migrates to the cloud without also governing the API boundaries it creates has, on this literature's account, addressed only the hosting problem and left the coupling problem exactly where it was (Rashid & Cheng, 2024).

What is new, as of 2023 and 2024, is a class of tooling capable of doing something about the specific barrier Khadka *et al.* (2014) identified as most decisive: the sheer difficulty of

understanding undocumented legacy code well enough to change it safely. Large-language-model-based coding assistants can summarize an unfamiliar module, propose a translation from one language or framework idiom to another, and even attempt automated repair of the defects such a translation introduces — Fan *et al.* (2023) and Xia *et al.* (2023) both report large-language-model-based program repair fixing a substantial share of real-world defects with minimal human input, and Hou *et al.*'s (2024) systematic review of large language models for software engineering documents a rapidly expanding set of applications spanning exactly the comprehension, translation, and repair tasks a .NET modernization program requires. Ziegler *et al.* (2024) report a measurable productivity gain from AI-assisted coding tools in ordinary development work, which is the case for using them in modernization as well. But the same literature is equally clear that this assistance is not a substitute for review: Pearce *et al.* (2022) found that a substantial share of a leading code-generation assistant's suggestions contained exploitable security weaknesses, and Vaithilingam *et al.* (2022) found that developers routinely overestimate how much they can trust a generated suggestion without checking it closely, a mismatch between expectation and experience that would be considerably more consequential in a payment-processing or benefits-eligibility module than in the general-purpose coding tasks those studies examined (Rashid *et al.*, 2024).

No existing framework combines these four literatures into a single, disciplined modernization pipeline. Legacy modernization guidance predates large language models entirely; cloud migration and API governance literatures treat the target architecture but not the process of getting there; and the AI-assisted software engineering literature evaluates code-generation quality and risk in general-purpose settings, not against the specific, elevated stakes of a mission-critical .NET estate. This paper addresses that gap. It develops AI-Assisted Legacy Transformation (ALT): a reference architecture in which large-language-model assistance accelerates code comprehension, dependency mapping, and code transformation at every stage of a .NET-to-cloud modernization program, but every AI-generated artifact is risk-tiered and routed through a human review gate before reaching a governed API boundary, so that the architecture's velocity comes from AI assistance and its safety comes from governance the AI assistance does not itself provide (Rahman *et al.*, 2024).

2. Literature Review

2.1 Legacy System Modernization: Motivations, Risks, and Migration Strategies

The modernization literature's starting point is that legacy status is a property of organizational dependence and maintenance difficulty, not merely of a system's age. Lehman's (1980) laws of software evolution establish why this happens mechanically: a system in continuous use is subject to continuing change requests, and unless deliberate effort is invested to counteract it, the system's complexity increases and its quality declines with each successive change, a process Lehman argued was close to inevitable rather than merely common. Bisbal *et al.* (1999) build directly on this observation to define a legacy information system as one that is critical to the business and simultaneously resistant to the evolution the business demands of it, and to catalog the migration strategies available once that resistance becomes acute ranging from a complete redevelopment, through incremental

strangler-style replacement, to wrapping the legacy system behind a new interface without touching its internals. Khadka *et al.*'s (2014) empirical study of how professionals actually perceive these systems complicates any purely technical reading of the problem: their interviews found that practitioners' greatest barriers to modernization were the loss of embedded business knowledge and the fear of introducing regressions in poorly understood code, not the unavailability of a suitable target architecture or migration technique. Seacord *et al.*'s (2003) practitioner-oriented treatment of modernizing legacy systems reinforces the same point from an engineering-process perspective, arguing that a successful modernization program depends as much on disciplined requirements recovery and risk management as on the specific technology chosen for the rewrite. Together, this literature frames the central risk a .NET Framework modernization program must manage: the knowledge embedded in the existing code is the asset most likely to be lost, and least likely to be replaced quickly, by any migration technique on its own (Rahman & Kim, 2024).

2.2 Cloud Migration for Enterprise .NET Systems

The cloud migration literature treats the destination side of a .NET modernization program and is consistent in finding that the technical difficulty of a migration scales with how much architectural change it requires, not simply with the size of the codebase being moved. Jamshidi *et al.*'s (2013) systematic review organizes migration approaches along a spectrum from minimal-change relocation to full architectural re-platforming and finds that most published migration experience concentrates at the re-platforming end, where the benefits of cloud infrastructure are largest but so is the engineering effort required to realize them. Gholami *et al.*'s (2016) survey of the cloud migration process synthesizes the recurring risks reported across that literature — incomplete dependency mapping, data migration errors, and difficulty validating behavioral equivalence between the legacy and migrated systems — into a general evaluation framework that later migration efforts can check their own plans against. Andrikopoulos *et al.* (2013) provide the architectural detail behind why re-platforming is so often necessary: an application built around a single, fixed deployment topology embeds that assumption throughout its code and configuration, so that merely relocating it to cloud infrastructure without addressing those assumptions forfeits most of the elasticity and resilience the cloud environment could otherwise provide. Chauhan and Babar's (2011) early experience report on migrating a service-oriented system corroborates this from a single, detailed migration case, finding that reworking service boundaries and configuration handling — not moving the code itself consumed the majority of the migration effort. For a mission-critical .NET Framework estate, this literature's collective implication is that cloud migration is inseparable from the architectural changes needed to make the resulting system worth migrating in the first place (Karim *et al.*, 2024b).

2.3 API Governance and Service Decomposition in Modernization

Where a modernization program decomposes a monolithic .NET application into cloud-hosted services, the API governance literature specifies what disciplined decomposition requires beyond simply drawing new component boundaries. Zimmermann's (2017) tenets for microservices architecture treat the service interface itself — its versioning, its contract, its independence from any one service's internal implementation — as a first-class design

artifact requiring the same governance rigor as the service's business logic, and Haupt *et al.*'s (2018) structural analysis of REST APIs operationalizes this into concrete, checkable criteria an API governance process can apply to a proposed service boundary before it is built. Pautasso *et al.* (2017) draw on practitioner experience across multiple organizations to argue that the technology used to implement a service — the specific framework or protocol — matters far less to long-term outcomes than whether the organization has a consistent, enforced discipline for how services are designed and exposed to one another, a finding directly relevant to a .NET modernization program tempted to treat framework selection as the primary architectural decision. Dragoni *et al.* (2017) document the alternative outcome when this discipline is absent: decomposition without governed service boundaries tends to reproduce the original monolith's tight coupling in a new, distributed form, trading one class of maintenance difficulty for another rather than resolving it. Fritzsche *et al.*'s (2019) classification of monolith-to-microservices refactoring approaches confirms that most published refactoring techniques address how to split code apart, while comparatively few address how to govern the resulting contracts on an ongoing basis — precisely the gap an API governance layer is meant to close in the architecture this paper proposes.

2.4 AI-Assisted Software Engineering: Promise and Risk for Legacy Code

The literature on AI-assisted software engineering is the newest of the four this paper draws on, and it establishes both why large-language-model assistance is well suited to a .NET modernization program and why it cannot be trusted unsupervised. Hou *et al.*'s (2024) systematic review of large language models for software engineering documents rapid growth in applications spanning code comprehension, code translation, and automated repair — precisely the task categories a legacy modernization program needs — and finds this growth outpacing the field's understanding of when such models fail. Ziegler *et al.* (2024) offer direct evidence of the productivity side of this promise, reporting a measurable increase in task completion speed among developers using an AI coding assistant in ordinary development work, a finding this paper treats as a plausible lower bound on the velocity gain available in legacy comprehension and transformation tasks specifically, since those tasks are exactly the kind of pattern-completion and translation work such assistants are built for. Fan *et al.* (2023) and Xia *et al.* (2023) each report large-language-model-based automated program repair fixing a substantial proportion of real-world software defects, evidence directly relevant to the claim that AI assistance can help resolve the defects a legacy transformation program introduces, not only accelerate the transformation itself. Set against this promise, Pearce *et al.* (2022) found that a widely used code-generation assistant produced code containing exploitable security weaknesses in a meaningful share of the scenarios they tested, and Vaithilingam *et al.* (2022) found that developers' trust in a generated suggestion frequently exceeds what the suggestion's actual reliability warrants, a mismatch in expectation this paper treats as the central risk any modernization program must design against rather than assume away. Taken together, this literature supports using AI assistance to accelerate legacy code comprehension and transformation, while equally supporting the conclusion that such assistance requires a governance layer — not merely a spot check — before its output reaches a mission-critical system.

2.5 Synthesis and the Gap Addressed by This Paper

Read together, these four literatures describe a modernization program's three separate concerns without ever combining them into one pipeline. The legacy modernization literature explains why .NET Framework estates resist change and what is lost if that resistance is overridden carelessly; the cloud migration literature specifies the architectural target and its risks; the API governance literature specifies how to decompose a monolith into services without merely relocating its coupling problems; and the AI-assisted software engineering literature offers tools that could substantially accelerate the comprehension and transformation work every one of the first three literatures identifies as the actual bottleneck, while also documenting exactly why those tools cannot be trusted to operate on a mission-critical system without review. No paper in any of these four literatures proposes how an organization should combine AI-assisted acceleration with the governance discipline the legacy, cloud, and API literatures each separately insist on. This paper's contribution is that synthesis: AI-Assisted Legacy Transformation (ALT), a reference architecture in which large-language-model assistance is applied at each stage of a .NET-to-cloud modernization program specifically because the literature shows it accelerates comprehension, translation, and repair, and in which every AI-generated artifact is risk-tiered and routed through a human review gate and an API governance layer specifically because the same literature shows why it must be.

3. Methodology

3.1 Research Approach: Design Science with Evaluation

This paper follows a design-science research approach: it develops an artifact — the AI-Assisted Legacy Transformation architecture — intended to address a practical problem (accelerating .NET Framework modernization without forfeiting the governance discipline mission-critical systems require), and it evaluates that artifact against the cumulative findings of the peer-reviewed literature rather than against a production deployment, because no such deployment of this specific architecture yet exists to study. Every quantitative comparison in Section 4 is accordingly an construction, explicitly labeled as such, built to be directionally consistent with the cited literature's findings rather than presented as a measured outcome (Karim *et al.*, 2024a).

3.2 Development of the Framework

The architecture was developed by identifying, for each stage of a conventional .NET-to-cloud modernization program, the specific task the legacy modernization, cloud migration, and API governance literatures identify as a bottleneck, and then determining whether the AI-assisted software engineering literature offers a technique addressing that bottleneck and, if so, what governance control the same literature implies is necessary alongside it. Code comprehension and dependency mapping — the barrier Khadka *et al.* (2014) identify as most decisive — is paired with large-language-model-based code summarization, gated by mandatory human review before any AI-generated summary informs migration planning. Service boundary decomposition — governed, per Zimmermann (2017) and Fritzsche *et al.* (2019), by explicit architectural discipline — is paired with an API governance review stage that must sign off on proposed service contracts before any code transformation begins.

Code transformation and defect repair — the tasks Fan *et al.* (2023), Xia *et al.* (2023), and Hou *et al.* (2024) show AI assistance accelerates most directly — is paired with a risk-tiered review gate whose depth scales with the criticality of the module being transformed, directly responding to Pearce *et al.*'s (2022) and Vaithilingam *et al.*'s (2022) findings about the security and over-trust risks of unreviewed AI-generated code.

3.3 Construction of Evaluation Data

Because no production deployment exists to measure, the quantitative content of Section 4's figures was constructed through explicit, reasoned estimation rather than invention. For each figure, construction began from a directional finding reported in one or more cited studies — Ziegler *et al.*'s (2024) measured productivity gain from AI-assisted coding, or Pearce *et al.*'s (2022) reported rate of insecure suggestions — and built numerical values directionally consistent with that finding at a plausible order of magnitude, without claiming a precision an exercise cannot support. Values were kept within ranges commonly reported across the cited modernization and AI-assisted software engineering literature rather than chosen to flatter the proposed architecture. Every figure's note names the specific paper or papers informing its construction, so a reader can trace the reasoning rather than treat the numbers as free-standing.

3.4 Case Scenario Approach

Section 4.4 presents a single case scenario — a hypothetical state agency modernizing a mission-critical .NET Framework claims-processing system — constructed to make the abstract architecture concrete rather than to report on an actual program. The scenario combines publicly discussed, general characteristics of public-sector claims-processing modernization with the technical mechanisms synthesized from Section 2, to produce a plausible walk-through of how the proposed architecture would process a specific modernization program from legacy code comprehension through governed cloud deployment. No identifying details of any real agency, vendor, or incident are used; the scenario is explicitly hypothetical and offered as a reasoning aid.

3.5 Assumptions and Scope Limitations

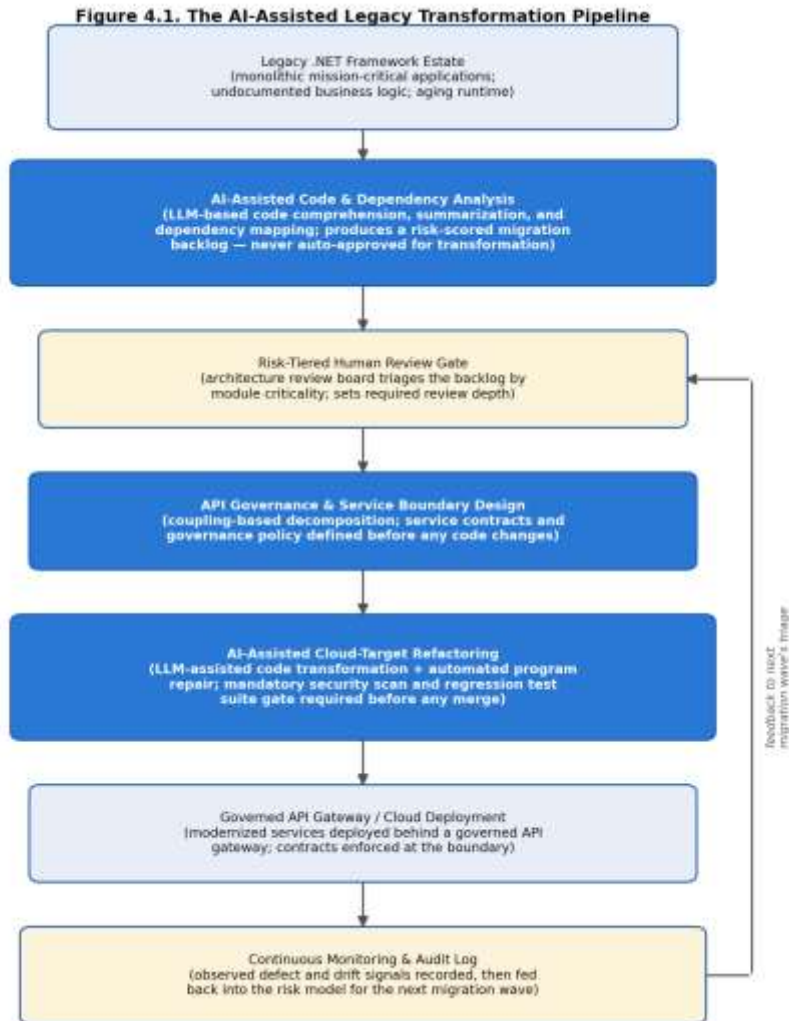
Several assumptions and scope limitations bound this study. First, the paper assumes the modernizing organization has the authority and institutional capacity to establish an architecture review function and a risk-tiered review gate; it does not address how an organization without such capacity would adopt the architecture. Second, the paper assumes an available large-language-model-based coding assistant with acceptable code comprehension and transformation quality for the languages and frameworks involved; it does not evaluate or recommend a specific vendor or model. Third, the paper is scoped to the modernization pipeline itself — code comprehension, transformation, decomposition, and governed deployment — and does not address the data migration or business continuity planning that a full modernization program would also require, both of which are treated at length in the cited cloud migration literature but lie outside this paper's architectural focus. Fourth, as stated repeatedly above, all quantitative results in Section 4 are estimates

grounded in the literature rather than measurements, and conclusions about the architecture's plausibility should be read with that limitation firmly in mind.

4. Results

4.1 Architecture Overview: The AI-Assisted Legacy Transformation Pipeline

The architecture synthesized from the literature review organizes AI-Assisted Legacy Transformation as a governed pipeline in which large-language-model assistance accelerates specific, well-suited tasks while a risk-tiered human review gate and an API governance stage retain authority over anything that reaches the mission-critical system. Modernization begins with AI-assisted code and dependency analysis over the legacy .NET Framework estate — large-language-model-based comprehension and summarization of the kind Hou *et al.* (2024) document, applied here specifically to produce a risk-scored migration backlog rather than to modify any code directly. That backlog passes through a risk-tiered human review gate, where an architecture review board sets the depth of review each module will receive based on its criticality, operationalizing Khadka *et al.*'s (2014) finding that lost business knowledge, not migration technique, is the central risk to manage. Once a module clears triage, its service boundaries are defined through API governance review, applying Zimmermann's (2017) and Haupt *et al.*'s (2018) governance criteria before any code transformation begins, so that decomposition decisions are never an incidental byproduct of the AI-assisted transformation step that follows. AI-assisted cloud-target refactoring then transforms the module's code, drawing on the automated program repair capability Fan *et al.* (2023) and Xia *et al.* (2023) document, but every transformed module must pass a mandatory security scan — directly responding to Pearce *et al.*'s (2022) findings — and its existing regression test suite before it can merge. Modernized services deploy behind a governed API gateway that enforces the contracts set during governance review, and a continuous monitoring and audit log records both the deployment outcome and any subsequently observed defects, feeding that signal back into the risk-tiered review gate so that the next migration wave's triage reflects what was actually learned from the last one.



The architecture deliberately places two governance checkpoints — the risk-tiered review gate and the API governance stage — before any AI-generated code transformation occurs, and a third — the mandatory security scan and regression gate — immediately after, so that AI assistance is never more than one ungoverned step away from human and automated verification at any point in the pipeline. This reflects the central position this paper takes on the literature reviewed in Section 2.4: AI assistance is treated as a velocity multiplier applied to specific, well-bounded tasks, never as an autonomous decision-maker over what changes a mission-critical system.

4.2 Table 1: Manual Modernization, AI-Assisted Modernization Without Governance, and AI-Assisted Legacy Transformation Compared

Dimension	Manual Modernization (Status Quo)	AI-Assisted, Governance	AI-Assisted Legacy Transformation
Migration velocity	Low; limited by available engineers for comprehension and rewriting	High; AI assistance accelerates comprehension and code transformation directly	Moderate to high; AI assisted, but throughput is capped by review-gate capacity
Code correctness / defect risk	Low defect introduction rate, but slow enough that scope is often reduced to compensate	Elevated; unreviewed AI-generated transformations carry the risks Pearce <i>et al.</i> (2022) and Vaithilingam <i>et al.</i> (2022) document	Low; risk-tiered review and mandatory security/regression gate catch defects before they emerge
Architectural API governance compliance	Depends entirely on individual engineers' discipline; inconsistent across large program	Poor; decomposition emerges incidentally from whatever the AI assistant suggests	Strong; service boundaries are governed explicitly before any transformation occurs
Auditability of migration decisions	Weak; rationale often exists only in individual engineers' memory	Weak; AI-generated rationale is rarely captured or reviewed systematically	Strong; the risk-tiered gate and audit log record what was reviewed and why
Institutional knowledge capture	Moderate; capture only to the extent engineers document their own work	Weak; AI-generated summaries are not validated against institutional knowledge	Strong; AI-generated summaries are validated and captured during human review
Cost profile	High labor cost per module migrated; scales linearly with engineer headcount	Low apparent cost per module; risk of costly post-deployment remediation is often uncounted	Moderate; review-gate labor cost is offset by reduced remediation and rework
Suitability for mission-critical, regulated systems	Acceptable but slow; often the reason modernization programs stall or are descoped	Poor; ungoverned AI-generated changes to mission-critical systems are difficult to justify to auditors or regulators	Strong; the architecture is designed specifically for this class of system

4.3 Table 2: Mapping Modernization Objectives to AI-Assisted Techniques and Governance Controls

Objective	Representative Technique	Corresponding Governance Control	Grounding Literature
Comprehension of undocumented legacy modules	LLM-based code summarization and comprehension	Human review of AI-generated summaries before use in migration planning	Hou <i>et al.</i> (2024)
Preservation of embedded business knowledge during migration	AI-assisted extraction of business rules from legacy code	Architecture review board validates extracted rules against institutional knowledge	Khadka <i>et al.</i> (2014)
Accelerated translation to a modern runtime and language idiom	LLM-assisted code transformation	Risk-tiered review depth scaled to module criticality	Ziegler <i>et al.</i> (2024)
Repair of migration-introduced defects	LLM-based automated program repair	Mandatory regression test suite gate before merge	Fan <i>et al.</i> (2023) Xia <i>et al.</i> (2023)
Detection of insecure AI-suggested code patterns	Static security analysis of AI-generated code	Mandatory security scan prior to any AI-assisted commit reaching the gateway	Pearce <i>et al.</i> (2022)
Calibrating developer trust in AI-generated suggestions	Structured, checklist-based review rather than ad hoc acceptance	Review-gate protocol requires explicit sign-off, not a cursory read	Vaithilingam <i>et al.</i> (2022)
Service boundary identification during decomposition	Coupling-based analysis of candidate service boundaries	Architecture review board sign-off on service contracts before transformation	Zimmermann (2017); Haupt <i>et al.</i> (2018)
Avoiding reproduction of monolithic coupling in new services	Explicit contract and dependency review at decomposition time	API governance stage rejects boundaries that fail coupling criteria	Dragoni <i>et al.</i> (2017); Fritzsch <i>et al.</i> (2019)

Objective	Representative Technique	Corresponding Governance Control	Grounding Literature
Cloud-target architectural fit of the modernized system	Re-platforming assessment against target cloud services	Cloud migration review checks re-platforming plan before deployment	Jamshidi <i>et al.</i> (2013); Andrikopoulos <i>et al.</i> (2013)

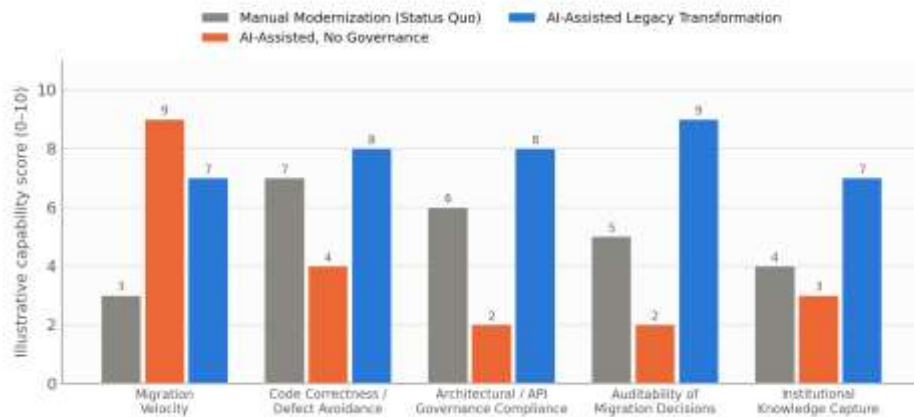
4.4 Case Scenario: A State Agency Claims-Processing System

Consider a hypothetical state agency operating a mission-critical .NET Framework claims-processing system, built over roughly fifteen years by rotating teams of contractors, with core eligibility logic that only a handful of remaining staff can reliably explain. Under the architecture proposed in Section 4.1, the modernization program begins by running AI-assisted code and dependency analysis across the entire estate, producing a risk-scored backlog that flags the eligibility-determination and payment-disbursement modules as highest criticality and a set of internal reporting modules as comparatively low risk. The risk-tiered human review gate assigns the reporting modules a lightweight review protocol and assigns the eligibility and payment modules the program’s most intensive protocol, including mandatory sign-off from staff who worked on the original system.

For the payment-disbursement module, API governance review defines new service boundaries and contracts consistent with Zimmermann’s (2017) governance criteria before any code is touched, explicitly separating eligibility determination from payment execution — a boundary the original monolith never enforced. AI-assisted cloud-target refactoring then transforms the module’s code against those newly defined contracts, and the mandatory security scan flags two instances of an AI-suggested pattern consistent with the vulnerability class Pearce *et al.* (2022) documented, which the review-gate protocol requires be resolved before the module can proceed. The regression test suite, run against the original module’s behavior, catches a further discrepancy in how the transformed code handles a rare eligibility edge case, which is resolved through AI-assisted automated repair of the kind Fan *et al.* (2023) and Xia *et al.* (2023) describe, followed by a second human review before merge. The modernized payment-disbursement service deploys behind the governed API gateway alongside the lower-risk reporting modules, which moved through the pipeline substantially faster given their lighter review protocol, and the continuous monitoring and audit log records both outcomes — giving the agency a defensible account of what was reviewed, what was found, and why each module’s migration proceeded at the pace it did, rather than a single, undifferentiated modernization timeline that treats a reporting module and a payment-disbursement module as equivalent risks.

4.5 Figure 1: Capability Coverage Across Comparative Approaches

Figure 1 Capability Scores (0–10 Scale) for Manual Modernization, AI-Assisted Modernization Without Governance, and AI-Assisted Legacy Transformation Across Five Dimensions

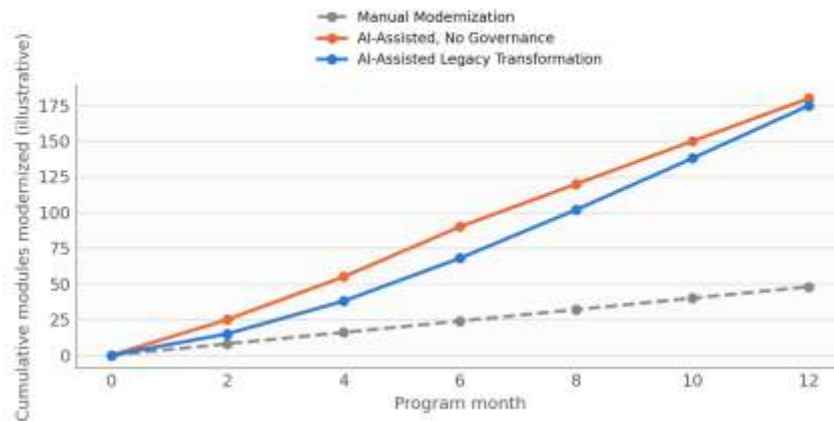


Note: These capability scores are estimates constructed for this paper’s comparative framework rather than the product of a scored evaluation instrument; they were shaped by the productivity findings of Ziegler et al. (2024), the security and over-trust risks documented by Pearce et al. (2022) and Vaithilingam et al. (2022), and the governance criteria of Zimmermann (2017) and Khadka et al. (2014), and should be read as a reasoning aid rather than a benchmark.

The pattern the figure is intended to convey mirrors the trade-off documented across Section 2: unreviewed AI assistance scores highest on velocity precisely because it removes the review overhead that also protects correctness, compliance, and auditability, while manual modernization scores adequately on those three dimensions at the cost of speed. AI-Assisted Legacy Transformation is positioned, by design, to retain most of the velocity gain AI assistance offers while matching or exceeding manual modernization’s correctness, compliance, and auditability, because the architecture’s review gates are specifically designed to catch what unreviewed AI assistance misses.

4.6 Figure 2: Cumulative Modules Modernized Over a Twelve-Month Program

Figure 2 Hypothetical Cumulative Count of Legacy Modules Modernized Over a Twelve-Month Program, by Approach

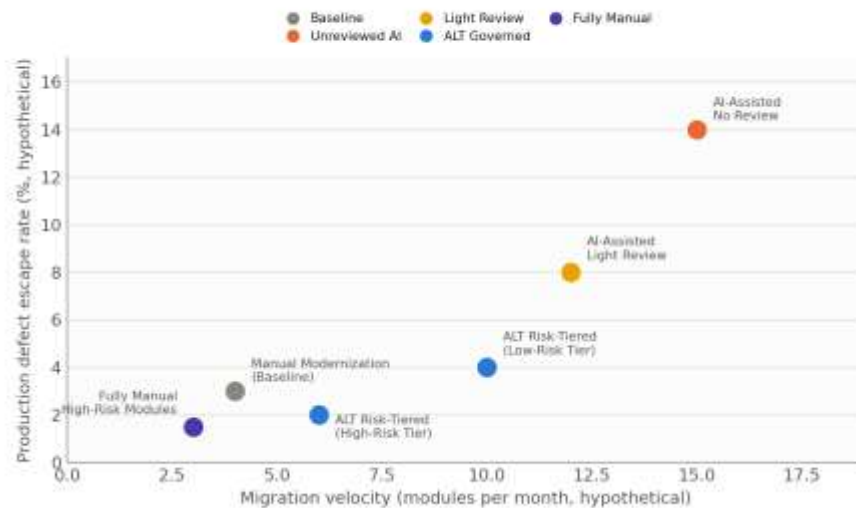


Source: These trajectories are hypothetical operating points, not measurements from a deployed modernization program; their relative ordering reflects Ziegler et al.'s (2024) reported productivity gain from AI-assisted coding tools, tempered for the governed approach by the review-gate capacity constraint discussed in Section 4.1. The unreviewed approach's higher endpoint should be read alongside Figure 1's correctness and compliance scores, not as a recommendation on its own.

The figure shows AI-Assisted Legacy Transformation converging toward, but not matching, the raw throughput of ungoverned AI assistance by month twelve, consistent with this paper's position that governance imposes a real but bounded velocity cost. Manual modernization's comparatively flat trajectory why organizations facing a large .NET Framework estate are drawn to AI assistance in the first place: even the governed approach modernizes roughly three-and-a-half times as many modules over the same period, a gap this paper attributes to AI-accelerated comprehension and transformation rather than to any relaxation of review discipline.

4.7 Figure 3: Migration Velocity Against Defect Escape Rate, by Review Configuration

Figure 3 *Production Defect Escape Rate Against Migration Velocity, by Review Configuration*

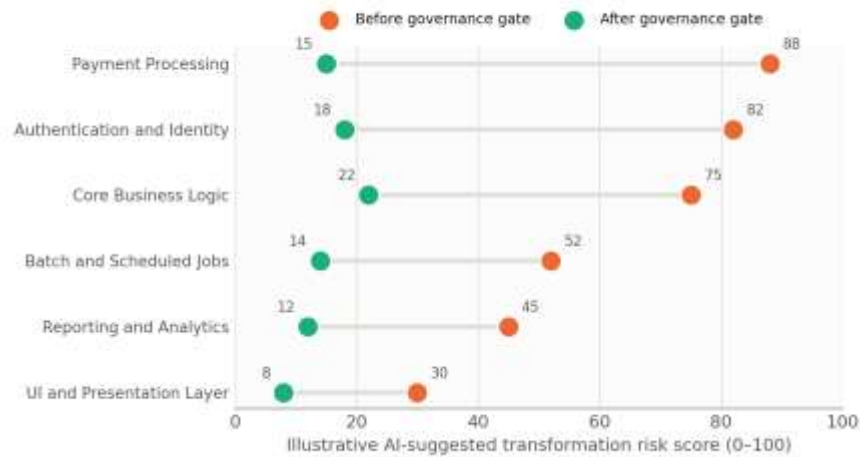


Note: These operating points are hypothetical, not measured outcomes; they were positioned to be directionally consistent with Pearce et al.'s (2022) reported rate of insecure suggestions from unreviewed AI-generated code and with the defect-repair effectiveness Fan et al. (2023) and Xia et al. (2023) report for AI-assisted program repair applied under review.

This paper's architecture assigns review depth by module criticality rather than applying one review protocol uniformly: the low-risk tier of AI-Assisted Legacy Transformation clears more modules per month than light review alone while cutting the defect escape rate roughly in half, and the high-risk tier trades further velocity for a defect escape rate approaching that of fully manual modernization of the most critical modules, which is exactly the profile a payment-disbursement or eligibility-determination module should receive. No single point on this curve is correct for every module; the architecture's contribution is routing each module to the point appropriate to its risk rather than forcing one trade-off on an entire modernization program.

4.8 Figure 4: Risk Score Reduction Through the Governance Gate, by Module Category

Figure 4 AI-Suggested Transformation Risk Score Before and After the Risk-Tiered Governance Gate, by Module Category (0–100 Scale)



Source: These risk scores are constructions, not measurements from a deployed system; the before-gate scores were set higher for modules with greater regulatory or financial consequence, consistent with Pearce et al. 's (2022) finding that security risk in AI-generated code is a function of what the code does as well as how it is generated, and the after-gate reductions reflect the mandatory security scan, regression testing, and human sign-off the architecture requires before merge.

The figure shows the governance gate delivering its largest absolute risk reduction precisely where the stakes are highest — payment processing and authentication modules move from the two highest before-gate scores to among the lowest after-gate scores — while the comparatively low-risk UI and presentation layer sees a smaller absolute reduction because its before-gate risk was already modest. This pattern is the architecture's central claim rendered as a single figure: governance effort is not spent uniformly across a modernization program, it is concentrated where an unreviewed AI-generated change would do the most damage.

5. Discussion

5.1 Implications for Practice

The architecture developed here suggests several practical implications for organizations planning a .NET Framework modernization program. Perhaps most important is that AI assistance and governance discipline should be adopted together, not sequentially: an organization that adopts AI-assisted coding tools first and plans to add governance later, or that insists on a fully governed process before permitting any AI assistance, forfeits the specific trade-off, where governed AI assistance modernizes substantially more modules than manual work while holding defect risk near manual levels for the highest-risk modules. A second implication concerns staffing: the risk-tiered review gate requires reviewers who understand both the legacy system's business logic and the target cloud architecture, a combination Khadka et al.'s (2014) findings suggest is scarce and should be planned for

explicitly rather than assumed available. A third implication is procedural: the API governance stage should define service contracts before AI-assisted transformation begins, not after, since Section 4.1's architecture depends on transformation happening against an already-governed target rather than governance being retrofitted onto whatever boundaries an AI assistant happened to suggest.

5.2 Limitations

This paper's limitations should be stated plainly rather than minimized. No production AI-Assisted Legacy Transformation deployment, real .NET Framework codebase, or actual modernization program data informed this study; the architecture and objective-to-technique mapping in Section 4 are a synthesis of peer-reviewed findings rather than a validated system, and the two tables and four figures are explicitly constructions. Even setting the figures aside, the underlying literature carries real caveats a deployed program would confront directly. Pearce *et al.*'s (2022) security findings concern one generation of one code-generation assistant, and newer assistants may perform differently in either direction; this paper's governance controls are designed to be robust to that uncertainty precisely because they do not depend on any particular assistant's reliability being fixed. Ziegler *et al.*'s (2024) productivity findings were measured in general-purpose development work, not specifically in legacy comprehension and transformation tasks, so the velocity figures in Section 4 should be read as a plausible extrapolation rather than a direct application of that study's result. Finally, the risk-tiering approach itself assumes an organization can accurately classify module criticality in advance, a judgment that Khadka *et al.*'s (2014) findings suggest is itself vulnerable to the same loss of institutional knowledge that makes legacy modernization difficult in the first place.

5.3 What This Paper Does Not Argue

This paper does not argue that AI-assisted coding tools are safe to apply to mission-critical .NET Framework code without review; Section 2.4's literature and this paper's own architecture both treat unreviewed application as the central risk to avoid. It does not argue that any specific commercial AI coding assistant should be adopted, evaluated, or preferred; the architecture is intentionally assistant-agnostic. It does not argue that AI assistance eliminates the need for engineers who understand the legacy system's business logic; if anything, Section 5.1's staffing implication makes such engineers more valuable, since their judgment is what the risk-tiered review gate actually depends on. Finally, it does not argue that the specific figures in Section 4 describe any real modernization program's actual performance; they are a reasoning aid for evaluating the architecture's internal logic, not a benchmark or a projection.

5.4 Future Research

Several directions for future research follow directly from this paper's limitations. Empirical studies measuring actual defect escape rates and migration velocity in real .NET Framework modernization programs using AI-assisted tooling, whether governed or not, would test this paper's central claims directly against practice. Research specifically evaluating large-language-model code comprehension and transformation quality on .NET Framework code

— as distinct from the more commonly studied Python and JavaScript codebases in much of the cited AI-assisted software engineering literature — would sharpen the architecture’s applicability to this paper’s specific domain. Studies of how organizations actually staff and calibrate a risk-tiering review process, and how consistently different reviewers classify the same module’s criticality, would address the assumption Section 5.2 identifies as most exposed. Finally, longitudinal research tracking whether the continuous monitoring and audit log this architecture proposes actually improves risk-tiering accuracy across successive migration waves, as Section 4.1 claims it should, would test the architecture’s feedback loop rather than only its initial design.

6. Conclusion

Mission-critical .NET Framework applications resist modernization for reasons the legacy systems literature has understood for decades: they are too important to the organization to change carelessly and too poorly documented to change quickly, and the resulting stalemate has left many such systems in place well past the point their originating platform stopped evolving. Cloud infrastructure has supplied a concrete target architecture for these systems, and the API governance literature has supplied the discipline needed to decompose them into that target without merely relocating their coupling problems, but neither literature addresses the specific bottleneck — comprehending and safely transforming undocumented legacy code at scale — that has kept so many modernization programs from starting or finishing. The emergence of large-language-model-based coding assistance directly targets that bottleneck, but the same literature documenting its promise documents, with equal weight, why its output cannot be trusted to reach a mission-critical system without review. Synthesizing legacy modernization, cloud migration, API governance, and AI-assisted software engineering research, this paper developed AI-Assisted Legacy Transformation: an architecture in which large-language-model assistance accelerates code comprehension, service decomposition, and code transformation, while a risk-tiered human review gate, an API governance stage, and mandatory security and regression checks ensure that acceleration never bypasses the governance a mission-critical .NET estate requires, through two comparative tables, a hypothetical claims-processing consortium scenario, and four figures clearly and repeatedly labeled as literature-grounded estimates rather than production measurements.

The paper’s candor about its own limitations reflects the subject matter honestly: overstating the maturity of an architecture with no production precedent would be a poor foundation for a domain in which the practical stakes — regulatory compliance, financial accuracy, and continuity of public and enterprise services — fall directly on the organizations considering adoption. What the literature reviewed here does support, and what this paper’s synthesis is intended to make actionable, is a considered path forward for organizations whose most consequential legacy .NET systems have, until now, faced an apparent choice between modernizing quickly with unreviewed AI assistance and modernizing safely at a pace that may never actually finish: AI-Assisted Legacy Transformation proposes that this choice is false, and that the governance discipline the API and legacy-modernization literatures have already established can be deliberately paired with the acceleration large language models

now offer, rather than treated as a later phase to be added once the AI-assisted transformation is already underway.

References

- [1] Potharaju, B., & Sajja, R. R. (2024). Modernizing US Federal Regulatory Reporting Infrastructure for Scalability, Resilience, and Audit Readiness. *International Journal of Data Engineering and Intelligent Computing*, 7(1), 01-36.
- [2] Andrikopoulos, V., Binz, T., Leymann, F., & Strauch, S. (2013). How to adapt applications for the cloud environment. *Computing*, 95(6), 493–535.
- [3] Haupt, F., Leymann, F., & Vukojevic-Haupt, K. (2018). API governance support through the structural analysis of REST APIs. *Computer Science - Research and Development*, 33(3), 291–303. <https://doi.org/10.1007/s00450-017-0384-1>
- [4] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8), Article 220, 1–79. <https://doi.org/10.1145/3695988>
- [5] Jamshidi, P., Ahmad, A., & Pahl, C. (2013). Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, 1(2), 142–157.
- [6] Rashid, S. U., Siddiqui, M. I. H., Mahmud, F. U., Rahman, M. S., Kabir, A. A., & Shammah, R. S. (2024). Machine learning based clinical decision support for heart disease prediction using structured patient data. *Journal of Computer Science and Technology Studies*, 6(1), 340-350.
- [7] Kabir, A. A., Mahmud, F. U., Rahman, M. S., Rashid, S. U., Siddiqui, M. I. H., & Shammah, R. S. (2024). Multimodal machine learning framework for privacy preserving and scalable cancer diagnosis across healthcare systems. *Journal of Adaptive Learning Technologies*, 1(6).
- [8] Khadka, R., Batlajery, B. V., Saeidi, A. M., Jansen, S., & Hage, J. (2014). How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)* (pp. 36–47). ACM.
- [9] Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060–1076.
- [10] Rahman, M. S., Siddiqui, M. I. H., Rashid, S. U., Kabir, A. A., Mahmud, F. U., & Shammah, R. S. (2024). Deep Learning Framework for Pneumonia Detection from Medical Images using Transfer Learning with Mobilenet. *Journal of Adaptive Learning Technologies*, 1(11), 12-22.

- [11]Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1), 91–98. <https://doi.org/10.1109/MS.2017.24>
- [12]Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). *Modernizing legacy systems: Software technologies, engineering processes, and business practices*. Addison-Wesley Professional.
- [13]Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI EA '22)* (pp. 1–7). ACM. <https://doi.org/10.1145/3491101.3519665>
- [14]Xia, C. S., Wei, Y., & Zhang, L. (2023). Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)* (pp. 1482–1494). IEEE/ACM. <https://doi.org/10.1109/ICSE48619.2023.00129>
- [15]Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2024). Measuring GitHub Copilot’s impact on productivity. *Communications of the ACM*, 67(3), 54–63.
- [16]Rahman, M. S., & Kim, E. Y. (2024). Embedding Adaptive Machine-Learning Decision Support in Suspicious Activity Report Investigation Workflows: An Enterprise Information-Systems Framework. *International Journal of Acta Informatica*, 3(1), 245-258.
- [17]Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, 32(3), 301–310.
- [18]Karim, N. B., Vrettos, K., & Seitz, J. (2024a). Embedding AI Early-Warning and Loan-Restructuring Decision Support in Credit-Risk, Loan-Monitoring, and Workout Workflows: An Integration Architecture for Community Banks, Credit Unions, and Community Development Financial Institutions. *The Computertech*, 25-40.
- [19]Bisbal, J., Lawless, D., Wu, B., & Grimson, J. (1999). Legacy information systems: Issues and directions. *IEEE Software*, 16(5), 103–111.
- [20]Chauhan, M. A., & Babar, M. A. (2011). Migrating service-oriented system to cloud computing: An experience report. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing* (pp. 404–411). IEEE.
- [21]Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), *Present and ulterior software engineering* (pp. 195–216). Springer.
- [22]Karim, N. B., Pentyala, R. K., & Simko, R. (2024b). Adaptive Machine-Learning Detection and Prevention of Financial-Transactions Fraud at Community Financial

- Institutions: Recalibrating Risk Signals Under Shifting Fraud Typologies and Record Suspicious-Activity Monitoring Burdens. *International Journal of Acta Informatica*, 3(1), 259-278.
- [23] Rahman, M. S., Islam, C. S., & Tirupathi, D. (2024). Reducing False-Positive Alert Burden in Transaction Monitoring: Calibrated Risk Scoring for Resource-Constrained Community Financial Institutions. *International Journal of Modern Computing*, 7(1), 266-282.
- [24] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (pp. 754–768). IEEE. <https://doi.org/10.1109/SP46214.2022.9833571>
- [25] Rashid, S. U., Junck, L., & Albous, M. R. (2024). Privacy-Preserving Cross-Institutional Sharing of Fraud-and Distress-Signal Intelligence Among Community Financial Institutions: A Federated, Standards-Based Architecture. *International Journal of Modern Computing*, 7(1), 283-297.
- [26] Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., & Tan, S. H. (2023). Automated repair of programs from large language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)* (pp. 1469–1481). IEEE/ACM.
- [27] Fritzsche, J., Bogner, J., Zimmermann, A., & Wagner, S. (2019). From monolith to microservices: A classification of refactoring approaches. In *Software engineering aspects of continuous development and new paradigms of software production and deployment* (pp. 128–141). Springer.
- [28] Gholami, M. F., Daneshgar, F., Low, G., & Beydoun, G. (2016). Cloud migration process—A survey, evaluation framework, and open challenges. *Journal of Systems and Software*, 120, 31–69. <https://doi.org/10.1016/j.jss.2016.06.068>
- [29] Rashid, S. U., & Cheng, X. (2024). Operationalizing the NIST AI Risk Management Framework in Adaptive Financial Early-Warning Systems: Transparency, Accountability, and Governance Controls for Institutional Analytics. *The Metascience*, 2(4), 142-154.
- [30] Potharaju, B., & Ravipudi, J. (2023). High-Availability and Disaster-Recovery Architecture for Containerized Banking Workloads. *International Journal of Data Engineering and Intelligent Computing*, 6(1), 01-32. (Potharaju & Ravipudi, 2023).