AMERICAN ACADEMIC PUBLISHER

INTERNATIONAL JOURNAL OF NETWORKS AND SECURITY (ISSN: 2693-387X)

Volume 05, Issue 01, 2025, pages 147-173
Published Date: - 07-06-2025
Doi: -https://doi.org/10.55640/ijns-05-01-09

AI-Assisted Legacy Modernization: Automating Monolith-to-Microservice Decomposition

Sandeep Reddy Gundla
Lead Software Engineer, MACYS Inc, GA, USA

ABSTRACT

Legacy systems are still critical business operations in many industries – but they are becoming roadblocks to innovation, agility, and scalability. As enterprises increasingly pressure themselves to modernize their aging infrastructures, strategic implementation of a transition from monolithic to microservices is gaining ground. Transforming this type of complex monolith into microservices is not a trivial task. It presents technical and organizational challenges, including bureaucratic service boundaries embedded in legacy codebases that tightly couple the service's functionality. The topic of this article is how artificial intelligence (AI) can help automate the decomposition of monolithic systems into decomposed, scalable microservices. By using machine learning, natural language processing, and clustering algorithms, AI tools can analyze source code, runtime data, and interactions between system components to determine intelligent service boundaries. A detailed methodology for AI-assisted decomposition is presented, along with real-world tools such as IBM Mono2Micro and AWS Microservice Extractor. A practical case study involving a global e-commerce company is included to illustrate applied outcomes. Additionally, the article addresses key challenges such as data inconsistency, domain misalignment, and organizational resistance. How it works outlines best practices to support successful implementation, including incremental migration patterns, domain-driven design, and DevOps integration. The article concludes with strategic recommendations and a forward-looking perspective on how AI will further change the modernization process. When done right, AI improves organizations' ability to create agile, future-prepared software ecosystems.

KEYWORDS

Legacy Modernization, Microservices Architecture, Artificial Intelligence, Monolith Decomposition, Software Engineering Automation

1. INTRODUCTION

Legacy systems have been the backbone of many critical business operations in the enterprise software world for a long time now. Frequently constructed with monolithic architectures, these systems took decades to formulate to fulfill a particular organizational requirement. The existence of a single, unified codebase with all functionalities: the user interface, the business logic, and the data access are tightly coupled in a monolithic architecture. While the design approach described above once accelerated initial development while simultaneously simplifying deployment, it has become increasingly a problem as digital demands have matured. With cloud computing, continuous delivery, and rapid innovation on tap, monolithic applications are starting to look their age. The architectures are difficult to scale, hard to maintain, and near impossible to evolve rapidly without putting the entire system at risk.

Today's business has two fronts to fight. However, the valuable logic and data embedded in legacy systems must be preserved. Existing industry practices insist on the organization's need to respond speedily to market changes, shifting customer expectations, and new technologies. So many organizations are approaching.

This tension is legacy modernization today as a strategic imperative. Moving from monoliths to microservices is just one of many modernization strategies and is especially compelling. Microservice splits an application into smaller, independently created, deployed, and scaled independent units. This architecture's agility is greater, the time–to–the market is faster, and fault isolation improves. There is no easy transition for this, however. This means digging into tightly knit code, figuring out the best way to carve out logical groups so data remains consistent, and reshaping teams and corresponding workflow.

There is no such thing as manual decomposition of a monolithic system into microservices; it is a tiresome and laborious work full of errors. Additionally, it necessitates a deep understanding of an existing codebase, along with more information on business domains and operational mandates. Also, the human process is not scaleable because other, more complex applications with millions of lines of code do not lend themselves well to the human process. It's artificial intelligence (AI), and there is promise. Using machine learning and natural language processing techniques combined with automated reasoning, AI technology can analyze source code, automatically detect patterns, and suggest boundaries that are very close to the principles of microservice.

Modernization comes in the form of an AI-assisted automation of all manual tasks. For example, AI static analysis tools can find code dependencies and bundles, while dynamic analysis plots runtime interactions and usage patterns. Advanced models utilize historical data, log files, and user behavior to create a mental picture of system components. It then allows us to propose or execute the ultimate service boundaries, which can shorten the expected time and effort needed for transition by about an order of magnitude. However, iterative learning, feedback loops, and accuracy and relevance over time allow AI systems to become more accurate and relevant in business operations as time goes on, therefore becoming an incredible tool to augment intelligence without replacing humans. However, this is not theoretical in its potential for AI. Some organizations and tech providers have started exploring and implementing AI-assisted heritage modernization. Recently, various tools have been launched by IBM, Amazon, and Microsoft to bust monoliths and adopt the migration to microservices using AI. These tools consider application architecture, propose candidate microservices, and generate scaffolding code for new services. While not mature, these technologies show how modernization can be done more confidently and efficiently.

This article discovers how AI can help automatically decompose monolithic systems into microservices. First, it will present an overview of monolithic and microservice architectures, followed by the growing need for modernization. The article will then examine how AI could help with this process in terms of methods and tools. The transport method of application of AI techniques in practice will be detailed in a separate methodology section. A case study will illustrate real-world applications of the above methodology, the challenges faced, tools used, best practices, and strategic recommendations are also discussed. Finally, the article concludes with an outlook on the future of AI-driven software modernization. In so doing, it aims to create a bridge between traditional modernization approaches and new opportunities afforded by AI, making it a complete, practical guide for organizations charting this critical transition.

2. Background and Motivation

2.1 Historical Context of Enterprise Applications

In the late 20th century, enterprises across industries developed centralized software systems to automate core business operations (Delsing, 2017). These systems, which were usually monolithic, bundled all functionality into a single executable or codebase. The languages used tended to be COBOL, Java, C++, and more, and they ran on mainframes or large application servers. Their structure was simple to deploy and initially developed and stable, with infrequent changes. As organizations grew and newer demands arose—online access, real-time data processing, mobile integration—they became a problem. While these systems are reliable, they have not been easily adapted to modern computing paradigms. The process took too long, eliminating project one function without regression testing the entire application. Consequently, monolithic applications ended up as a barrier to digital progress.

This diagram below shows how critical enterprise functions such as logistics, financial operations, production planning, and human capital management are interconnected through master data, emphasizing the tightly coupled nature of traditional monolithic systems.

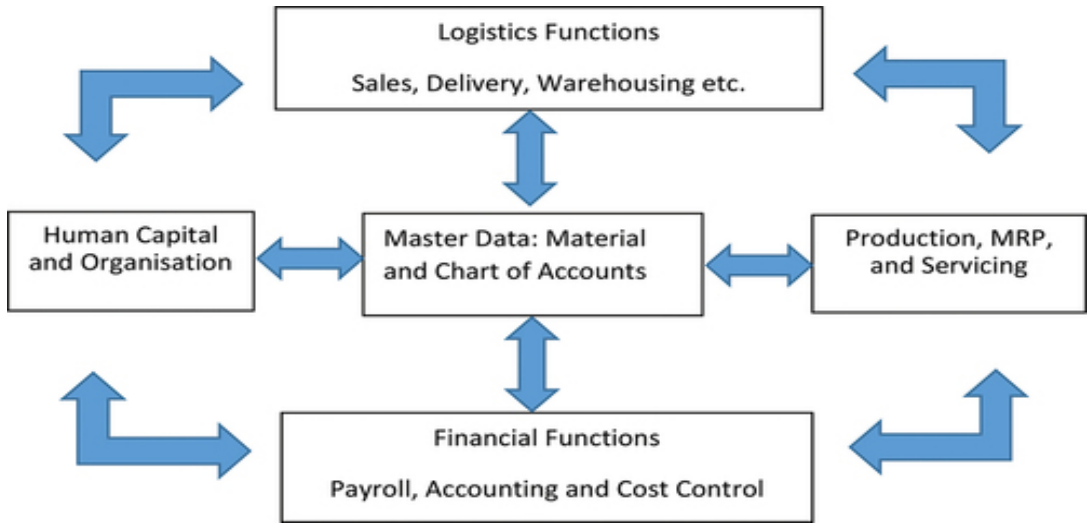

Figure 1: Enterprise systems core – ERP.

2.2 Limitations of Monolithic Architecture.

There are several operational and developmental bottlenecks inherent in monolithic systems (Rüßmann et al., 2015). Technically, one of the most significant issues is the absence of clear module boundaries, which leads to tight coupling among components. In such architectures, different parts of the system are highly interdependent, making it difficult to update or modify individual modules without impacting others. This tight interconnection creates a fragile development environment where even minor changes can necessitate full application redeployment. As a result, there is a heightened risk of system downtime and production errors, which hampers agility and responsiveness. This condition not only reduces modular clarity but also complicates decision-making and increases system-wide vulnerability during updates. (Goel & Bhramhabhatt, 2024).

Another problem is scale. These systems generally scale vertically by increasing the GPU(s), CPU(s), RAM, or disk resources to meet larger loads rather than horizontally distributing the loads over multiple instances. This is both an expensive and inflexible vertical approach. In addition, monolithic systems are also compromised on fault tolerance. Because everything runs in a shared memory and runtime environment, a bug in one part of the application (the billing module) could crash the whole system. The issue from a development standpoint is that

large teams working from the same codebase can produce strains by themselves (Šmite et al., 2017). Version control can quickly become overwhelming, code review needs to happen, and testing coverage needs to be verified. On top of that, it is difficult to onboard new developers because understanding a big, interdependent codebase requires a steep learning curve. These limitations are compounded when these organizations try to align their development processes with modern practices, like agile development and continuous integration/continuous deployment (CI/CD). Iterative releases and fast feedback loops don't come naturally to monoliths. Companies that want to stay relevant need an architectural solution that readily supports rapid change and scale.

2.3 Drivers for migration to microservices

More and more decisions are not motivated solely by technical grounds; business needs are pushing as well (McClelland & Burnham, 2017). Today, organizations are pressured to innovate, reduce costs, and respond quickly to market changes. Customers demand feature updates, a digital experience that is as if it were on a silver platter, and no downtime. Teams need to be able to build, test, and deploy services independently, which microservices address by allowing this. For example, an e-commerce platform might have separate microservices for inventory, payments, user authentication, and order tracking. None of these should be taken as being either inviolable or unchangeable. Each of them can be updated or scaled independently of the others. This adds modularity to increase development speed and lessen the chance of systemic failure. In addition, microservices are a very good fit for cloud-native architectures and container technologies like Docker and Kubernetes, which are built to manage loosely coupled distributed systems.

AWS, Azure, and Google Cloud, among other cloud service providers, provide abundant support for microservice deployments (Gohil & Patel, 2024). Offering orchestration, monitoring, logging, and auto-scaling tools, they remove the operational burden of working on an internal team. Therefore, companies can spend more time creating business value and less time managing infrastructure. Also, microservices support organizational agility. They result in better accountability and faster decisions, which deliver better services for customers. This autonomous structure allows companies to scale their technology and their workforce with very little friction compared to monolithic development models.

The performance and flexibility limitations of monolithic systems are becoming increasingly evident, particularly as digital demands grow more dynamic. Traditional architectures often fail to support real-time data processing, scaling, and system responsiveness. At the same time, market pressures demand rapid innovation while furthering operational efficiency and cost control. These challenges offer a strong argument for redesigning the system as a more adaptive model. Going from monoliths to microservices is a strategic move to enhance modularity, scalability, and reaction speeds. The transition is complex but is often performed incrementally to reduce the risk and the disruption. Organizations can decompose monolithic architectures into independent services to better address gaps in performance and consistency and achieve business agility goals. This process is further augmented by AI-driven tools that automate dependency analysis, suggest optimal service boundaries, and make deployment constant easy. The performance of these advancements helps close the gap between performance and reliability for modern applications (Dhanagari, 2024).

3. Understanding Monoliths vs. Microservices

Table 1: ***Key Differences Between Monolithic and Microservices Architectures***

Feature	**Monolithic Architecture**	**Microservices Architecture**
Codebase	Single, unified	Multiple, independently managed
Deployment	One unit	Independent service deployments
Scalability	Vertical	Horizontal, service-specific
Fault Isolation	Low	High
Development Teams	Centralized, large teams	Decentralized, smaller cross-functional
Release Cycle	Infrequent, risky	Frequent, incremental
Technology Stack Flexibility	Limited	High (language/framework per service)
Testing Complexity	High	Service-specific, easier in isolation

3.1 Defining Monolithic Applications

A monolithic application is an application whose functionalities live in the same logic unit in the same deployable unit (Tapia et al., 2020). Under this structure, the business logic, database, and user interface layer are interfaced together as a single unit. This model can help with the early stages of development—everything is in one code base—but complicates matters significantly over time. Most enterprises have grown their monolithic applications to millions of lines of code and thousands of interdependent modules. Such modules tend to share memory, global state, and closely coupled logic, and therefore, the application is deceptively sensitive to even slight modifications.

A problem with monolithic systems is that anything as small as a minor change can require one to replete and redeploy the entire system (Newman, 2019). This increases deployment time and the chances of bugs being introduced. Testing becomes more difficult because the application must be validated to guarantee that innovative additions or changes to one area will not conflict with the others. As a result, the release cycle is long, and updating frequently is discouraged. Monolithic architectures can work well for small-scale applications or even startups but become less sustainable as systems grow and evolve.

This diagram below shows the unified composition of a monolithic application, where the user interface, business layer, and data interface are tightly integrated and deployed as one unit alongside the central database.

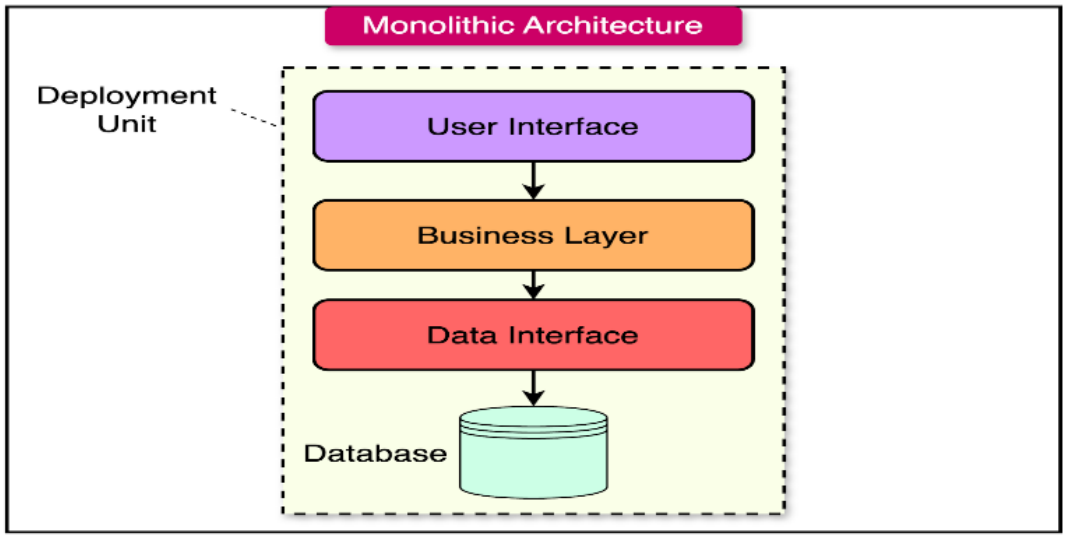

Figure 2: ***Monolith vs Microservices vs Modular Monoliths***

3.2 Microservices are a set of key characteristics.

The architectural approach of microservices is fundamentally different. With the microservices model, instead of placing all functionalities into one application, they are decoupled into loosely coupled services that function independently. They employ lightweight protocols such as HTTP or messaging queues for communication between services that handle different business functions, e.g., user management, payment processing, or notification delivery, where each service is responsible for a specific business function. It allows each microservice to develop, test, and scale independently. It is also one of the most important characteristics, following domain-driven design capabilities (Zhong et al., 2024). A service can be bound (in the DDD sense) to a bounded context to own the logic and data needed to perform a particular domain function. These reasons make for architectural clarity, leading to higher maintainability and greater agility. It allows development teams to work on different services in parallel, without intersecting, which means faster innovation and more reliable releases. Scalability is also a microservices feature by design. The system's stability is achieved by allowing high-traffic services high-traffic services to be independently scaled. For instance, an order management service within an e-commerce platform will have high resource requirements during peak sales, while other services like content management may be relatively stable. This flexibility gives us flexibility to reduce infrastructure costs or to improve performance under load.

3.3 Technical and Organizational Comparison.

Monoliths and microservices need to be compared on both technical and organizational factors. From a technical point of view, monoliths mean a single über codebase that can make certain aspects of development easier, like debugging on initial deployment. However, the systems suffer from bad modularity, scalability, and change management. In opposition, microservices increase the complexity of managing distributed systems but provide high modularity, fault isolation, and scalability. Monoliths historically cause the code to be distributed among large centralized teams whose responsibilities are vast. As the team grows, coordination becomes harder, and a bottleneck can be experienced when multiple developers work on the same codebase. Because of its smaller, cross-functional teams can own a given service. Autonomy of this nature reduces communication overhead and increases development velocity. Moreover, microservices work perfectly with Agile and DevOps, making building continuous integration and deployment (CI/CD) pipelines easy. Microservices are great, but they come with their own set of

problems. These include robust service discovery mechanisms, strong, secure API gateways, consistent logging and monitoring, and reliable means of communication amongst the distributed components.

3.4 Real-World Decomposition Challenges

Breaking a monolithic system into microservices is no minor feat. The first thing to do is to discover service boundaries in an intertwined code base. One often have to duplicate or even spread business logic across multiple modules. There is no good way to isolate services from one another due to shared databases cleanly. In some cases, such separation is impossible without rewriting substantial parts of the code and increases the risk of interrupting key workflows. Data consistency is another major challenge faced in data management applications. Normally, monolithic systems run over a single relational database using transactions and constraints to maintain data integrity. In a microservice environment, data consistency becomes complicated and often relies on an eventual consistency model since each service might have its own database. Reflecting on this requires a different way of thinking about data ownership and synchronization. Lastly, backward compatibility and interoperability with current systems must be considered. Most enterprises do not have the luxury of starting over and rewriting everything. In reality, they'll need to support a hybrid model for a time, during which part of the monolith and new microservices will reside and collaborate. Managing the transitional phase requires careful planning, robust API contracts, and thoughtful orchestration strategies.

Any organization aiming at modernization must understand the fundamental differences between monolithic and microservices architectures (Fritzsch, 2024). This knowledge is critical to navigating the structural and operational shifts that modernization entails. Transitioning to microservices introduces significant technical and organizational challenges—ranging from distributed data management to inter-service communication and deployment orchestration. To overcome these hurdles, organizations need a thorough grasp of how such changes impact infrastructure, workflows, and governance models. As demonstrated in large-scale systems, scalable solutions like MongoDB are often leveraged to manage real-time data streams efficiently, supporting the modularity and speed that microservices demand. These systems also highlight that architectural adaptability is a critical feature for enabling high-volume data and responsiveness, further reinforcing the strategic value of microservices. The following sections outline how artificial intelligence can assist in the complex transition toward modernization, from automation to insights into the modernization process (Dhanagari, 2024)

4. Role of AI in Software Modernization

Table 2: ***AI Techniques and Their Roles in Modernization***

AI Technique	Function in Modernization	Example Tools/Applications
Machine Learning (ML)	Detect usage patterns and change frequencies	IBM Mono2Micro, AWS Microservice Extractor
Natural Language Processing (NLP)	Interpret code comments and documentation	Code2Vec, OpenAI Codex
Deep Learning (DL)	Analyze code structure and behavior at scale	Custom DL models, Microsoft research projects
Clustering Algorithms	Group related code components into service candidates	K-Means, DBSCAN, Hierarchical Clustering

AI Technique	Function in Modernization	Example Tools/Applications
Static Code Analysis	Detect code dependencies without execution	Sourcery, OpenRewrite
Dynamic Analysis	Map real-time interactions and system behavior	Runtime log analyzers, tracing tools

4.1 Introduction to AI Techniques (ML, NLP, DL)

Artificial intelligence (AI) has seen strides recently, providing practical solutions to complex and highly data-intensive problems in various domains (Xu et al., 2021). Legacy system modernization is a problem amenable to AI techniques, particularly machine learning (ML), natural language processing (NLP), and deep learning (DL), all of which have the potential to increase productivity in the software engineering domain at large. The techniques presented here address an urgent need of organizations trying to understand, restructure, and improve large software systems. Machine learning uses algorithms on historical or structured data so that systems learn patterns or make predictions. ML models trained on code repositories, commit histories, and runtime data are used to infer application segments that are frequently used together and commonly change within the same execution context. These insights help identify cohesive functionality groups that can be structured as independent services.

This process relies on natural language processing of textual data—such as code comments, documentation, naming conventions, and user stories. Several tools (including PyMC, jmmCNN, and distressed ML) have been developed to analyze collapsed attributes, detect continuous event patterns, and examine complex scenarios that require expert interpretation of code and version control logs. NLP models can be used to classify functional descriptions of modules (or find human-readable explanations of legacy components using NLP models); unlike many conventional machine learning techniques that process input, such as speech commands or waveform images, deep learning (DL) works through layered neural networks to process more complex data structures. Similarly, data load models can consume complex inputs, like source code hierarchies, dependency trees, and system event logs, which is relevant to the issue of software modernization. However, computationally intensive, such models can uncover hidden relationships in large codebases. Consequently, such tools are critical to automating sophisticated tasks involving code similarity detection, functional classification, and domain partitioning, which are highly labor-intensive tasks that require manual review. Automation strongly resonates with contemporary DevSecOps principles, whereby intelligent tools add to CI / CD pipelines to improve accuracy and speed delivery while protecting system security and integrity (Konneru, 2021).

As seen in the diagram below, areas within AI relevant to modernization touch on natural language processing, generative AI, predictive analytics, robotics, and computer vision, which can inform or complement ML, NLP, and DL techniques in practical implementations of these approaches.

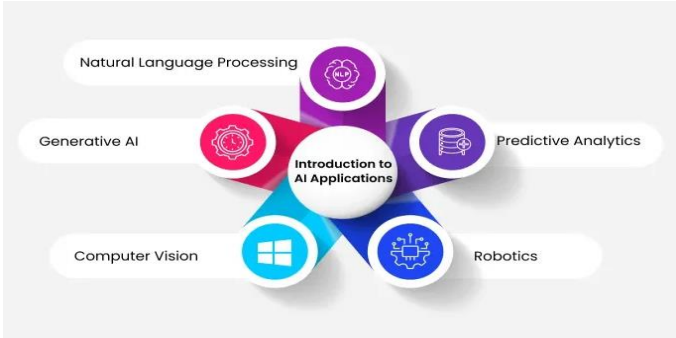

Figure 3: ***Artificial Intelligence (AI)***

4.2 AI-based Code Analysis and Service Discovery

Understanding the application marks the starting point of the monolith-to-microservices transition. This phase is accelerated with AI through automated static and dynamic code analysis (Gu, 2020). In static analysis, the source code is examined without the need for execution. By parsing class hierarchy, method call, and dependency graph, AI-driven tools can detect whether a module has tightly coupled modules or common components being accessed. Using these tools, it becomes possible to identify groups of functions that frequently interact with each other—making them strong candidates for microservices. In the context of dynamic analysis, the application is observed at runtime.

By analyzing logs, usage patterns, and monitoring data as required by AI models, real-world interactions between system components can be mapped. For example, if a set of functions is consistently triggered together during a customer checkout process, those functions can be logically grouped into a single microservice. A practical solution to this question is to apply clustering algorithms such as k-means and hierarchical clustering to chunk together classes or functions that happen to belong together based on several dimensions—frequency of use, semantic similarity, dependency strength, and runtime co-occurrence. Such clusters are used as a potential basis for services. In addition to synthesizing candidate service boundaries through visualization of relationships and proposing modular decompositions, AI can also help form a framework model. These suggestions lower manual effort while providing a structured starting point for software architects. The automatic insights greatly decrease analysis time and point out areas that might have slipped through the cracks without them.

4.3 AI-Driven Tooling Landscape: State-of-the-Art

AI is already starting to be used in software modernization, with several real-world tools and platforms already employing it. A case to mention is IBM's Mono2Micro, which uses AI to scan Java-based monoliths and suggest possible microservice candidates. The tool maps the application's structure by mixing static and dynamic analysis and clustering. Next, developers can go through these groupings and turn them into sensible microservice boundaries. Amazon Web Services (AWS) Microservice Extractor for .NET is a tool for identifying components in .NET applications that can be separated into microservices. It analyzes source code and dependencies, provides recommendations, and can even 'scaffold' a new service. This approach makes it easier—though not always easy—to extract reusable components from an application for deployment. In software engineering, Microsoft has also looked at AI and how language models can be used to analyze legacy code and suggest targeted modernizations. These models allow us to identify old patterns, make improvements, and smoothen the refactoring process. Simultaneously, open-source communities have developed several other AI-based static analysis tools, such as

Sourcery and Code2Vec. Its purpose is to automate code quality improvements, detect logical inefficiencies, and suggest refactoring aligned with modern software design principles. By integrating such tools into event-driven environments and CI/CD workflows, organizations can increase system resilience, reduce manual effort, and maintain fault tolerance in complex, distributed systems—capabilities that are critical in large-scale modernization efforts (Chavan, 2024). While not perfect replacements for human decision-makers, these tools provide considerable support around speed, scale, and consistency (Jarrahi, 2018). They ease engineers ' cognitive burden and ensure that modernization efforts are motivated by data-driven insights rather than intuition. Essentially, what was once a theoretical concept has transformed into a suite of practical tools that can enable improvement in virtually every stage of modernization. By taking us through the process involving legacy systems to architectural changes, AI tools supplement many human experts and make large-scale transforms workable.

5. Methodology

Transiting to a microservice architecture from a monolithic system is yet another complex process. Though methodical and usually complete, traditional manual methods are also time-consuming and seldom scale with the size and complexity of enterprise-level applications. These challenges are mitigated with an AI-assisted approach of precise, automated, adaptive modernization. Herein, a structured methodology is presented for applying artificial intelligence to support the decomposition of monolithic systems into microservices. The methodology is developed through six interlocking phases, achieving meaningful efficiencies and insights along the overall journey of transformation.

As a foundational step to modernization, comprehensive data must be collected from the monolithic system. This consists of its full source code, architectural diagrams, version control logs, application performance metrics, and runtime data (user activity traces and system logs). The goal is to develop a unified dataset characterizing application structural and behavioral characteristics. Static analysis of the source code is per-formed for the logical structure and depen-dencies and from the runtime metrics and logs to understand the interaction of different components in the production environments. However, suppose the documentation is incomplete and outdated. In that case, there is still little that can be done apart from using natural language processing (NLP) tools to extract semantic meaning from inline comments, variable names, and function identifiers. The reasoning behind this enriched analysis is especially important in determining appropriate service boundaries for microservices in that this provides the means to infer the roles of different modules. Accurate boundary definition ensures that services are cohesive, loosely coupled, and aligned with real-world business domains—an essential requirement for successful system decomposition and long-term maintainability (Chavan, 2022).

Once data is collected, AI models analyze static and dynamic dependency (Kibria et al., 2018). The static analysis looks at the application code and doesn’t execute any of it to reveal class hierarchies, method calls, import dependencies, and library linkages. Constructing such a map from this analysis of code components gives insight into how different components fit together systemically. In parallel, dynamic analysis monitors the application at runtime to infer how the application is used. AI systems infer which components tend to be used jointly, when, under what circumstances, and how data flows through the system. The combination of static to grow structural insight and dynamic to grasp behavioral understanding is critical for realizing a fuller picture of interdependencies and operational relationships to resolve the question of deciding which components are to remain coupled and which can be uncoupled.

Machine learning algorithms are applied to these to group related functions and components whose dependencies are well-defined. Source files, classes, or methods are grouped by clustering techniques (K-means, DBSCAN, or

hierarchical clustering) according to (for example) interaction frequency, semantic similarity, and shared data access. The first candidates for microservices are these clusters. Where arbitrary separation fails, the clustering is rooted in empirical usage and business logic. For instance, order processing-related code functions may be located together because they always work together and are in the same transactional sandboxes. They may also be spread throughout the original codebase.

The clusters are presented in AI-generated form, dependency, graphs, heat maps. These become vehicles for development teams to review and interpret the proposed service boundaries. At this stage, AI tools suggest clusters that could be used as microservices by assessing cohesion within modules and coupling between them. They flag those boundaries with high internal cohesion and low external dependencies as strong candidates. The tools help the architects validate visually if these suggestions match their system knowledge or what they know regarding the business domains of the organization. Human review at places where the outputs of the AI are ambiguous or at odds with operational knowledge helps keep critical logic from being unintentionally broken up. Although the work starts from an AI decomposition, the human in the loop must subsequently validate it. In partnership, domain experts, enterprise architects, and development leads assess AI-generated service candidates to determine if they are technically feasible and business-relevant. Depending on user roles and compliance requirements, and may need to adjust interdepartmental workflows or operational constraints. This phase ensures the microservices serve both the logical structure, organizational context, and domain expertise, translating into higher functionality and maintainability.

Automated code transformation and scaffolding constitute the final phase of the AI-assisted modernization process. Once service boundaries are validated, AI tools can generate foundational code for the proposed microservices. The output of this is often service templates, RESTful APIs, data access layers, and integration points. In more advanced situations, classes in the monolith are automatically chopped up into microservices, individually deployable, each with its own data store and interface. Also, these tools provide capabilities to generate deployment-ready artifacts like Dockerfiles, Kubernetes manifests, CI\CD configurations, to ease the journey to production. Human developers are still needed for validation and fine-tuning. However, automation reduces the amount of manual work. The development of some of these capabilities, particularly around modeling legacy code structure and the generation of corresponding service interfaces, depends largely on principles that exist in natural language inference—where the objective is to infer logical relationships from structured inputs—and show that AI is an important fundamental tool in contemporary software engineering (Raju, 2017). These six phases are executed as a holistic, scalable AI-assisted legacy modernization approach. Organizations that bring intelligent analysis, collaborative validation, and automated transformation together can accelerate microservices move with minimal risk, keeping technical and business objectives in balance.

6. Case Study / Real-World Example

It is no longer a theoretical model or experimental research but is becoming a practical enabler of enterprise transformation. This has been done by many organizations that have successfully leveraged AI to transform legacy systems to gradually move from rigid monolithic architectures to agile, scalable microservice architectures. These are no longer speculative transitions but generate operational improvements and business value in various sectors. A case study is presented based on a global retail company that modernized its core application infrastructure across the stack using end-to-end AI-enabled modernization. The initiative used an approach that combines AI-assisted code analysis, service decomposition, and deployment automation to turn a legacy system into a microservices architecture. Predictive analytics and intelligent tooling played in cases to achieve faster deployment

cycles, lower system downtime, and enhanced DevOps efficiency; this was a clear expression of the synergy between AI and Business intelligence and scalability (Kumar, 2019).

Below is a diagram that describes the four key dimensions organizations must consider if they want to embark on AI-assisted transformation: their operational workflows, legacy I.T. systems, when data is available, and how ready they are for AI.

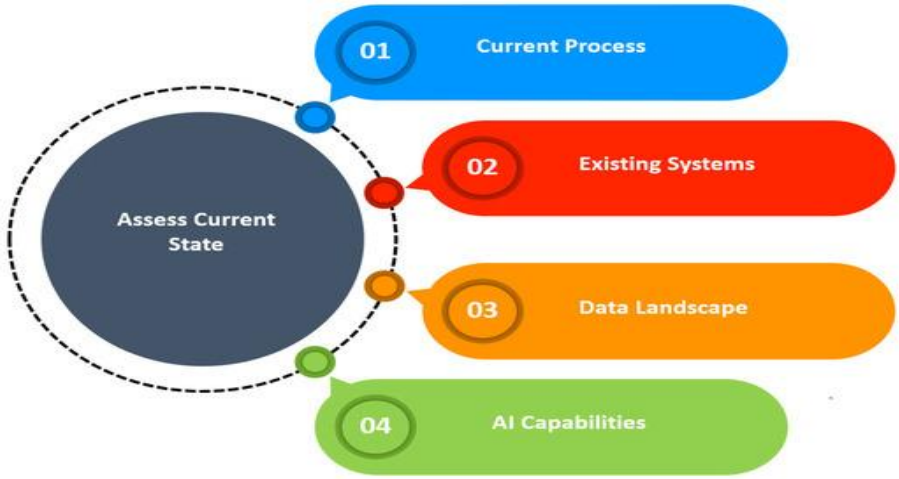

Figure 4: Element of current state assessment.

6.1 Modernization Scenario Overview

The company selected for this case study operated in the e-commerce space and managed one of the largest digital retail platforms in its region, serving millions of active users and processing numerous transactions daily. The core system had been developed over a decade earlier using Java EE and was structured as a monolithic application with approximately 8 million lines of code. It handled key operations such as inventory management, order processing, customer service, and promotional campaigns—all within a single deployment package.

Such an architecture created significant operational bottlenecks (Baldwin, 2015). Deployment cycles extended over several weeks due to extensive testing and quality assurance requirements. For instance, even when only a single component—such as payment processing—required additional capacity, the entire application had to be scaled by provisioning large server instances. Updates were often delayed due to the high risk of unintentionally impacting unrelated parts of the system. These constraints directly limited business agility, particularly during high-demand periods like Black Friday and seasonal promotional events. Realizing the need for change, the company continued with its modernization program. To avoid the high-risk, multiyear effort of rewriting the application from scratch, they chose to decompose the application with AI-assisted tools in phases.

6.2 AI Tools and Approach Taken

This Modernization journey started with a period of discovery, during which engineers used IBM Mono2Micro to examine a Java-based application. To identify dependencies and code interaction patterns, the tool combined static code analysis with runtime log inspection. Over the course of over twenty days, it covered every service call, method call, and class dependency to produce a dependency graph. By adding built-in clustering algorithms, the AI proposed groupings aligned with functional domains such as order fulfillment, customer account management, and discount application (Akerkar, 2019). These groupings were visualized on an interactive dashboard, enabling software

architects to review, validate, and refine the AI's logic based on real-world usage. In parallel, the engineering team employed NLP-based tools to scan the source code for business-related terms and logic patterns embedded in method names and comments. This process enhanced the mapping of technical components to business domains, ensuring better alignment and reducing the risk of logic fragmentation. The clustering approach shares conceptual similarities with techniques used in modern telematics systems, where intelligent grouping and real-time analytics optimize asset management and operational communication—demonstrating how data-driven models can enhance decision-making across domains (Nyati, 2018). Since then, the team has reviewed and refined the groupings and selected three clusters to start with microservice extraction. The services chosen were clear separation, low interdependency, high business value user authentication, product catalog, and shopping cart management.

6.3. Outcomes and Measured Benefits

The outcomes of the AI-assisted decomposition were tangible in a matter of months. Mono2Micro scaffolding code was used to develop the three first microservices, which were independently deployed using Docker containers orchestrated through Kubernetes. Load testing showed that under stress, these microservices were handling five times the traffic independently compared to the old monolith. With these services, development time on new features within them fell by more than 40% since teams could deploy changes without waiting for full application builds or for other unrelated teams to coordinate. With the adoption of CI/CD pipelines, these services were updated every week versus monthly release cycles on the monolith. The analysis of the operational metrics revealed a decrease in downtime and deployment-related incidents. On the other hand, the modernization effort did not disrupt business operations (Harris & Krueger, 2015). These new services used internal APIs to kick the monolith to the curb and use remaining functions while the legacy monolith continued to operate. This hybrid architecture provided the company an opportunity to modernize incrementally, shrinking both the technical and financial risks involved.

6.4 Challenges Confronted

Although the process had successful outcomes, it was not without challenges. The main challenge was ensuring the data stayed consistent throughout every system. But at first, parts of the database were still shared between the monolith and the new microservices, which led to synchronization issues. To solve this, the team gradually switched over to a data store per microservice and enabled asynchronous event-driven communication for updates between them. Another problem was team alignment. Even though AI tools offered better suggestions for service boundaries, human validation ultimately needed deep business knowledge so that product owners, QA analysts, or even customer service representatives answered the question of what makes sense. To feed off this new model of independent service ownership, teams had to increase their communication with each other dramatically. Security, too, needed careful planning. When these services were externalized, new access controls and API gateways had to be erected (Suzic & Latinovic, 2020). This entailed configuring token-based authentication, putting forward rate limits, and making certain that communication between services was encrypted.

7. Tools and Frameworks

For AI-assisted modernization to succeed, the tools and frameworks employed for code analysis, microservice identification, and architectural transformation are critical (Naeem Syed et al., 2018). Over the past few years, major cloud vendors and specialized software platforms have introduced robust solutions for decomposing monolithic systems. These tools blend artificial intelligence with domain-specific insights to help organizations automate some

of the most intricate aspects of modernization. Their capabilities range from source code clustering to runtime behavior analysis and scaffold generation. As with any transformation strategy, a deep understanding of each tool's functionality, required inputs, and output limitations is essential for success. Just as advanced generative models are used to synthesize highly complex and layered 3D environments in digital design, similarly sophisticated AI systems are now being applied to manage the architectural complexity of enterprise applications—highlighting the transformative potential of AI across technical domains (Singh, 2022).

Table 3: ***Tools for AI-Assisted Monolith Decomposition***

Tool Name	**Supported Language/Platform**	**Key Features**	**AI Capabilities**
IBM Mono2Micro	Java	Static/dynamic analysis, clustering	AI-driven grouping, scaffolding
AWS Microservice Extractor	.NET	Code analysis, extraction, AWS-ready	Rule-based + AI boundary detection
Microsoft IntelliCode	.NET, C#	Refactoring suggestions	AI-assisted recommendations
OpenRewrite	Java (open-source)	Rule-based code transformations	Limited AI integration
Service Cutter	General	Use-case & entity-based decomposition	Minimal AI, strong structural output

7.1 Leading Tools Overview

Several tools are currently available in the market to assist with AI-driven monolith decomposition. Each tool has a unique feature targeting a certain development stack, programming language, or cloud ecosystem. One of the most well-known platforms in this space is IBM Mono2Micro, which is specifically designed for modernizing Java-based monolithic applications (De Santis et al., 2016). The tool uses artificial intelligence to analyze class-level dependencies, interpret runtime behavior, and extract insights from application logs. Based on this data, Mono2Micro recommends domain-aligned partitions of the codebase, presenting them through interactive visualizations that assist developers in evaluating and refining architectural boundaries. While the tool provides automated suggestions, these recommendations are intentionally nonfinal, enabling engineers to manually adjust service groupings before generating scaffolding code for each microservice. Similar to how large language models interpret context in visual question answering systems to generate accurate responses, Mono2Micro applies structured reasoning across software components to produce interpretable, actionable outputs for modernization (Singh, 2023).

AWS Microservice Extractor for .NET is focused on splitting .NET technology applications. It provides static code analysis on top of heuristics to find the business logic and technical dependencies. The output can be deployed directly to AWS's ecosystem (such as Lambda functions, ECS, and EKS) and can extract functions into services without code change. The AI provided within Microsoft Visual Studio IntelliCode and other related tools offers suggestions for refactoring and finding reusable components across developers' .NET and C# applications. Isolating these patterns is useful, though. While not specifically aimed at monolith decomposition, it does provide great value

in terms of decomposing code.

Open Rewrite is an open-source framework that aims to improve automated code refactoring. It's especially useful when enforcing structural code transformations on a large codebase (Antal et al., 2016). While it lacks advanced clustering or AI-driven microservice extraction, it supports rule-based logic that can work well with AI tools in isolation of microservices. Service Cutter, another open-source tool, decomposes service micros by calculating architectural coupling from use cases, domain models, and entity relations. Although it doesn't depend on heavy AI, it gives structured outputs that can be used as inputs to AI techniques or combined with other tools.

7.2 Input Requirements and Output Capabilities.

The AI assistant tools used for decomposing monolithic systems typically require multiple types of input to function effectively. At a minimum, they need access to the source code to perform static analysis. However, tools such as IBM Mono2Micro or AWS Microservice Extractor are significantly more effective when combined with runtime data, including log files, execution traces, and application performance metrics. This enriched data allows the tools to identify real-world usage patterns and dependencies that may not be obvious through static code analysis alone. Much like how image captioning models benefit from contextual visual and semantic cues to generate meaningful descriptions, these AI-driven decomposition tools depend on comprehensive contextual inputs to suggest reliable and business-aligned service boundaries (Sukhadiya et al., 2018).

Some of these tools can be used without sophistication and generate different outputs. These visualizations offer code dependency, interaction graphs, and suggested service grouping representations. More powerful tools produce artifacts ready to use: scaffolding for microservices, Docker configurations, Kubernetes manifests, and RESTful API templates. These significantly decreased the time needed for the hands-on development and deployment stages following the analysis phase. For example, each microservice that Mono2Micro identifies can generate Java Spring Boot projects, preconfigured to route to and from the API, and make calls to external services. Also, the AWS Microservice Extractor creates corresponding directly integrated Code Pipeline instantiating refactored services for pushing to production environments.

The diagram below visualizes how AI systems process a variety of inputs—including questions, voice commands, text, and images—to generate actionable outputs such as decisions, predictions, suggestions, and even creative work. In software modernization, a similar process is followed, transforming source data into deployable assets.

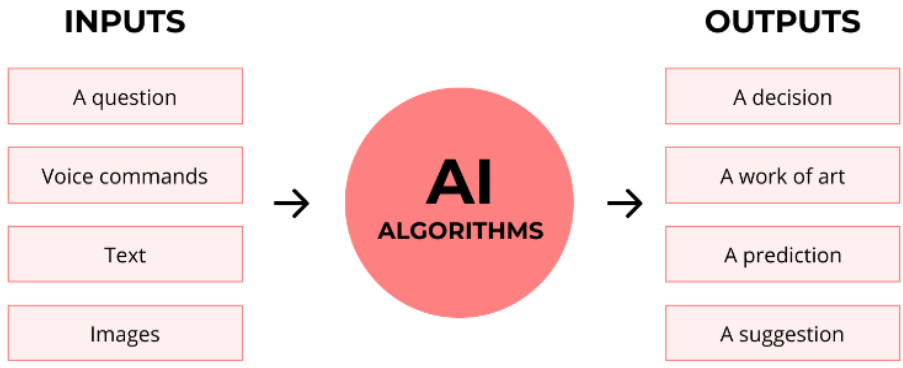

Figure 5: artificial-intelligence

7.3 Evaluation Criteria and Tool Limitations

Choosing the right AI-powered tool to perform monolith-to-microservices decomposition is driven by an

organization's technological stack, infrastructure maturity, and internal technical prowess. Although each organization's needs may differ, several factors must be considered in the evaluation process. The first issue is that there is no language and framework support; most tools are developed with particular programming languages in mind. For example, IBM's Mono2Micro works very well for Java applications, while AWS Microservice Extractor targets the .NET ecosystem. Then, the transparency of AI models is necessary. Unlike black boxes that are hard to understand or validate, explainable outcomes such as visual clustering diagrams, interactive dashboards, and editable recommendations help developers trust and allocate budgets better.

Integration capabilities are important. A robust modernization tool must seamlessly integrate with CI/CD pipelines, version control systems (such as Git), and container orchestration platforms like Kubernetes to support a streamlined development lifecycle. Scalability is equally critical—users should be able to work with enterprise-scale applications, often encompassing millions of lines of code, without encountering performance lags or crashes (Crookshanks, 2015). Just as important is the degree of user control these tools provide. The ability to fully edit AI-generated recommendations ensures that architects and developers can apply domain-specific expertise and make necessary adjustments to align with project goals. This mirrors how AI-powered platforms in other domains—such as personalized career coaching—enable users to engage dynamically with recommendations, tailoring outputs based on individual context and feedback. These parallels highlight that, regardless of domain, AI systems achieve the greatest impact when they support human agency rather than replace it (Karwa, 2023; Karwa, 2024).

Although these are strong points, most current AI tools are not fully autonomous. It is their job to supplement, but not replace, the decision-making of software architects. There are issues in highly heterogeneous and legacy environments (such as using an outdated programming language such as COBOL or Delphi) with some of the tools. Scenarios of this type may lie outside the realm of modern AI solutions and necessitate more specialized customization or even manual intervention. AI tools are very effective at modernizing, but only when juxtaposed with human oversight, aligned strategy, and a realistic understanding of the limits of what it can do.

8. Challenges and Risks

AI-assisted modernization has many advantages in reducing complex and time-intensive tasks that can be automated (Kim & Lim, 2021). However, the journey from the monolithic system to the microservices architecture is not smooth sailing. For a modernization effort to be successful, an understanding of and a way to address both technical and organizational risks is required. Though powerful, AI tools cannot solve every problem alone, and unwieldy AI tools might cause more problems than they solve. This section looks at the most common challenges and risks that heritage modernization with AI-assisted solutions encounters.

Table 4: ***Common Risks in AI-Assisted Modernization***

Risk Type	Description	Recommended Mitigation
Incomplete Dependency Mapping	Static/dynamic data may miss key relationships	Combine methods, extend observation windows
Domain Misalignment	AI suggests groupings lacking business context	Human validation by domain experts
Over-Decomposition	Excessive microservices increase operational overhead	Set service granularity thresholds

Risk Type	Description	Recommended Mitigation
Team Resistance	Cultural pushback against AI and new workflows	Training, communication, change management
Tool Limitations	Poor support for legacy languages or edge cases	Manual intervention, hybrid modernization

8.1 Missing or unclear dependency information

One of the greatest risks in decomposing monolithic applications is the use of incomplete or inaccurate system information. For many years, many legacy systems have been in use without comprehensive documentation or up-to-date architectural diagrams. These things make it hard for AI tools to grasp application structure and relationships completely. However, applications that use dynamic code loading, reflection, or a configuration file external to the application to modify program logic can often defeat static analysis, missing important runtime behaviors. Moreover, dynamic analysis might only see a portion of the behavior during the observation window, depending on the inputs or usage patterns used during analysis. When those captured logs do not show all the scenarios of usage in a system, the resulting service boundaries are likely built on a partial view of the system (Pasquier et al., 2017).

8.2 Automatically decomposing into domain contexts

AI systems are good at taking large datasets and identifying patterns and correlations, but they don't have business domain knowledge. Consequently, automated decomposition can produce technically sound recommendations that are at odds with the functional reality of the business. For example, an AI tool might bring product and pricing modules together simply because both have data models, even though these are managed by separate departments and governed by different business rules. This becomes more acute in highly regulated industries, including healthcare, banking, and insurance, where service separation is not only about code modularity but also a requirement for compliance and data privacy. AI-assisted decomposition without domain-specific validation may result in service designs that violate an organization's policies or create legal risk.

Human-in-the-loop processes should always be integrated from scratch. AI-generated outputs need to be reviewed by subject matter experts to ensure that business logic is constrained, boundaries are accurate regarding real-world workflow, and sensitive data is protected. As illustrated below in Figure 6. Just as a biologist interprets machine-generated insights in the context of biological systems, software architects must ground AI outputs in organizational realities to achieve meaningful and sustainable transformation.

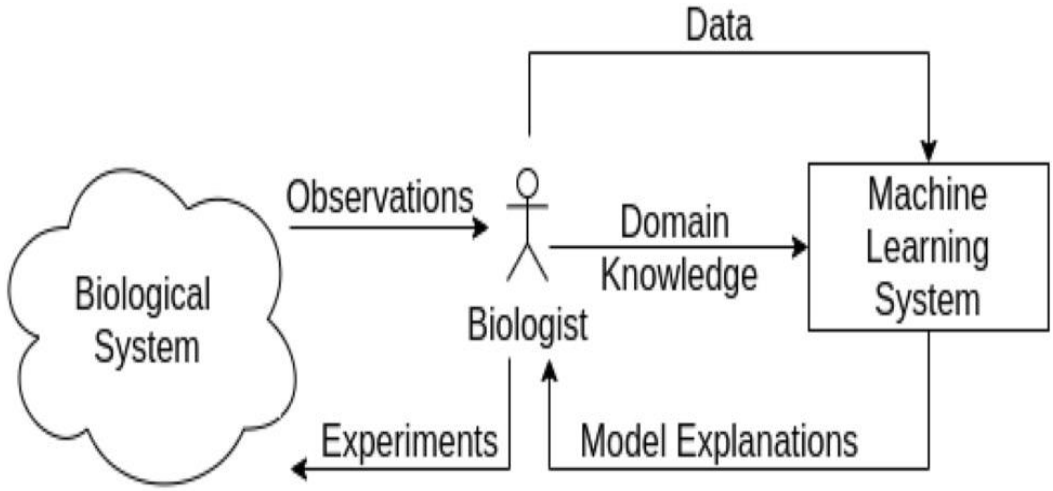

Figure 6: An example of AI for Science

8.3 Risks of Prior Decomposition

Another danger is over-decomposition—creating too many microservices. Although the concept of modularity is desirable, overly fine subdivision results in additional complexity, greater maintenance expense, and more degraded performance. This leads to each microservice being demanded to have its own deployment pipeline, monitoring tools, logging configuration, database or data access interface, and security controls (Ahmadvand et al., 2018). As the number of services grows beyond a manageable level, the overhead of maintaining and coordinating these services outweighs the benefit of scale. Another name for this is the microservice sprawl problem. When misused, the risk is that AI tools can help find even weakly interacting components as possible candidates for independent services. Organizations may find it difficult to manage, monitor, and troubleshoot services without thoughtful consolidation or prioritization of these suggestions.

8.4 Cultural and Process Resistance

But it's not a one-sided conversation. Furthermore, legacy modernization—helped by AI—brings a lot of organizational change. Years of development teams working with monolithic systems may resist the shift to microservices because they are unfamiliar with or scared of being made redundant. Especially in the architectural field, skepticism about AI-generated and AI-generated decisions, especially for those with experience as architects, because they value manual analysis and decision-making. Besides this, switching to microservices requires changes to team structure, workflows, and operational tooling. Today, teams must change to the latest deployment models, API contracts, or even decentralized service ownership. Modernization efforts can run aground on internal friction unless change management, training, and stakeholder engagement proceed properly.

9. Best Practices and Guidelines

The process behind the transition from monolithic to microservice architecture, especially when there's help from the AI, plays a much bigger role than the tools used. With a weak foundation in methodology, governance, and team coordination behind it, even the best technologies might not perform better than expected. This section summarizes a collection of best practices and practical guidelines based on experiences to help ensure that AI-assisted legacy modernization is efficient and sustainable. Based on industry experience and case studies, these recommendations provide a roadmap for dealing with complexity, maintaining business value, and reducing risk.

9.1. Domain-Driven and Business-Centric Planning

It is important to understand the business domains that the software supports before initiating technical changes. The domain of this principle is rooted in Domain Driven Design (DDD), where the key idea revolves around creating a software architecture that fits our real-world business model. While, once again, AI tools can help cluster and suggest service boundaries, they need to be validated against domain knowledge. First, define which business functions, such as order processing, inventory management, user accounts, or payment systems, must talk to each other (Rainer et al., 2020). Include business analysts, product owners, and domain experts early during decomposition. With their insights, AI-generated service grouping would respect functional boundaries and not create logic across teams or departments. It reduces the complexity of inter-service communication and later defines service ownership. Domain models, workflow diagrams, and user stories all help drive and fine-tune AI suggestions. Domain-driven reasoning should be enhanced by AI, not substituted.

As illustrated below in Figure 7, highlights how continuous communication with domain experts reinforces and improves software architecture. This cycle ensures that architecture evolves iteratively through expert feedback and real-world development experience, making it both technically robust and business-aligned.

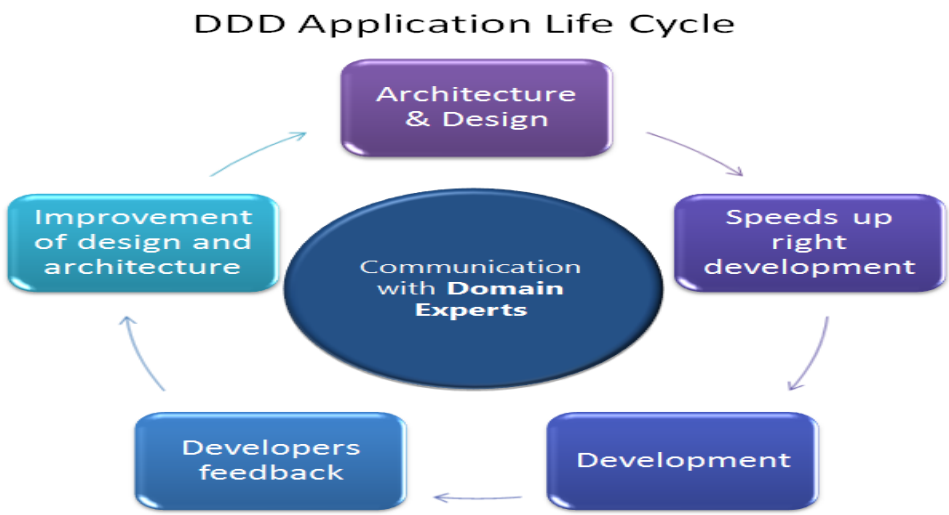

Figure 7: ***Domain Driven Design***

9.2 Incremental decomposition patterns

It's high risk because decomposing an entire monolithic application is far too complex to do in one phase. The use of incremental decomposition patterns, such as the Strangler Fig Pattern, represents a more practical and low-risk approach. This strategy consists of gradually replacing parts of the monolith with new microservices, doing so without disabling the rest of the system. A relatively isolated module with high business value and low dependencies—such as user authentication or product catalog management—should be selected as a starting point (Tahir et al., 2021). Interactions within the application can then be mapped using AI tools to support the generation of scaffolding code. Once the microservice is built and deployed, service calls should be rerouted from the monolith to the new microservice through an API gateway or proxy layer. This process should be repeated iteratively for additional modules to enable a gradual and controlled migration. This pattern minimizes the disruption caused by the release and allows time to validate system behavior, train staff, and tune the monitoring tools on the new generation of servers. In addition, it allows early benefits to be realized until transformation is complete.

9.3 Continuous Testing Integration and DevOps

Development and operational practices need to change to adapt to microservices' shift. Ensuring quality and performance during modernization requires integrating DevOps from the start. Microservices should be independently testable, deployable, and observable. CI/CD pipelines should be established for each service early in the process. This ensures that teams can develop against the AI-generated scaffolding code and immediately test and deploy within an automated workflow. Services should be containerized using Docker and deployed using Kubernetes for orchestration. This approach supports scalable deployment practices and maintains consistency across environments. Automated testing must be embedded at different levels, including unit, integration, and end-to-end testing. Observability is important in microservices, especially given their distributed nature, and Prometheus, Grafana with Dashboard Builder, and distributed tracing solutions (e.g., Jaeger) are essential tools to support that. These provide insights into system performance and the detection of problems across services.

9.4 Organizational collaboration and governance.

Modernization is not solely a technical endeavor; it also involves significant cultural and organizational changes. AI-assisted decomposition can only succeed when supported by cross-functional collaboration and well-defined governance structures. The initial step involves establishing small, autonomous teams that take full ownership of individual services. These teams should include developers, testers, product owners, and operations staff (Poth et al., 2020). Clear responsibilities must be defined—indicating who owns each service, who manages deployments, and who provides operational support. This decentralization fosters both agility and accountability throughout the development lifecycle. Create governance standards that cover service versioning, API documentation, security policy, and error-handling protocols. Although teams work independently in principle, they should strive to comply with shared guidelines to promote systemic reliability and consistency. Create open communication channels where people share AI-generated insights, decomposition plans, and test results with transparency across teams. This will help build trust in AI tools and organize collective decisions.

10. Recommendations

Moving to microservices is a large technical and organizational leap. Artificial intelligence, however, can be approached with well-informed strategies to make it faster, more accurate, and more scalable. This section synthesizes the insights of the time and offers practical and strategic advice for organizations considering or transitioning to AI-assisted modernization. These recommendations are classified under four key categories: strategic alignment, technical execution, organizational enablement, and long-term sustainability. Figure 8 illustrates this architectural constraint with the unified structure presented in monolithic applications.

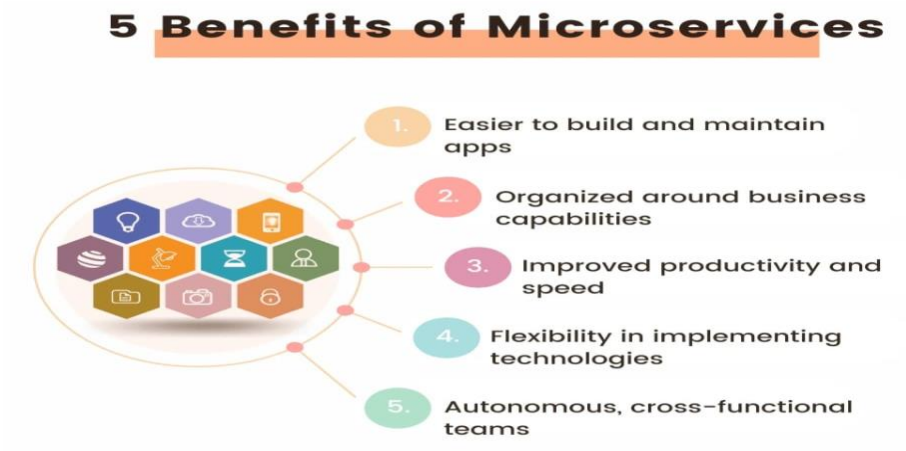

Figure 8: monolithic-application

10.1 Strategic Recommendations are given.

Modernization should not be viewed merely as a technical upgrade; rather, it must be recognized as a strategic initiative aligned with clearly defined business outcomes. The first recommendation is to explicitly define the business goals driving the modernization effort. Whether the objective is to improve scalability, reduce time to market, or support the delivery of new digital services, these goals should inform the selection of tools, the identification of decomposition targets, and the choice of appropriate deployment strategies. Leadership should treat AI-assisted modernization as a multi-phase transformation, not a one-time project (Geetha et al., 2019). These phases should include assessment, planning, initial pilots, iterative rollouts, and performance reviews. A one-week sprint should produce some measurable value. Setting up key performance indicators (KPIs) early in the process makes tracking progress easier and justifies further investment. Moreover, strategic buy-in from executive leadership is required. From an operational standpoint, top-down support helps provide the teams with the resources, time, and training to participate in transformation activities without impacting ongoing business operations.

10.2. Technical Implementation Guidelines

From a technical perspective, AI tools should not be seen as autonomous agents but as decision-support systems. Developers, architects, and domain experts must stay engaged in reviewing AI-generated recommendations against their business needs and architectural standards. Start with small, low-risk things. Pick up relatively independent services that the development team understands well. The behavior of the component is analyzed using AI, which attempts to propose boundaries and scaffold. This initial rollout provides much-needed feedback to fine-tune AI configs and decomposition strategies for the more complex modules. Make sure there is infrastructure in place. This encompasses CI/CD pipelines, service mesh configurations, containerization standards, and observability tooling. Therefore, AI-generated code should fit naturally into current development workflows. It is preferable to have tools that make visualizations and are capable of editing groupings and logs so that they are transparent and trustable, making easy validation.

10.3 Organizational readiness and training

Technical change must be accompanied by organizational change. Cross-functional team training is one of the most effective ways to improve product velocity and quality. In addition to understanding the fundamentals of

microservice design and DevOps practices, teams must also grasp how AI tools function, what inputs are required, and how to accurately interpret the outputs. Secondly, set up multidisciplinary squads of developers, QA, ops, and business analysts. This will improve decision-making regarding service boundaries and service deployment strategies. Within a well-defined governance framework, each team should be autonomous in managing its own services. It is all about communication. Regular briefings, documentation updates, and opportunities for open feedback reduce resistance and guarantee alignment across departments. Consider designating some internal modernization champions—people who serve as bridges between IT and business leadership, getting things going and clearing expectations.

10.4 Long-term Maintenance and Continuous Evolution

The ultimate recommendation is to view modernization as a continuing process rather than as a one-time aim. After the first set of services are deployed and stabilized, continuous improvement can commence. Assess the service health, measure the deployment frequency, and analyze use patterns to discover areas of savings. This should happen in parallel to continue refining AI models (Lu et al., 2022). As the organization collects more inputs (logs, user behavior, development activity), that information can be used to help improve future decomposition accuracy. This allows smarter automation to take place over time.

It is important to take a forward-thinking approach to managing technical debt. When designing services that are expected to grow and scale, care must be taken to avoid unintentionally creating distributed monoliths—systems that appear modular on the surface but remain heavily interdependent beneath. True modularity and service independence are achieved through regular architectural reviews, thorough dependency audits, and continuous refactoring cycles. Living Artifacts are artifacts whose maintenance is ongoing after creation. They are extremely important for new developers to onboard, for debugging production issues, and for facilitating cross-team collaboration.

11. Future Outlook

AI in Software Modernization is expected to become more sophisticated and deeply integrated. Most current tools are focused on supporting analysis and scaffolding; however, upcoming innovations are likely to represent context-aware AI systems capable of reasoning not only about code structures but also about business objectives, compliance requirements, and user behavior. In time, even entire microservices may be written autonomously by Generative AI models with their own test suites, database configuration, and deployment scripts. These tools will generate selectively tailored architectural patterns for specific industries, substantially reducing design overhead. These models could be integrated into development environments to suggest real-time decomposition solutions for teams while building and maintaining legacy systems.

One promising additional development is the prevalence of autonomous modernization pipelines (Dickerson & Worthen, 2024). Without waiting for the start of a formal modernization project, these systems could continuously scan live applications, recognize monolithic patterns or performance bottlenecks, and propose/recommend changes or carry out changes. Such pipelines, when paired with AI-driven observability platforms, could create self-optimizing systems that adapt in real time to usage trends and resource constraints. Modernization in this future will never be done; it will simply occur invisibly, as an operating principle, not an isolated effort. Today's legacy systems will incrementally evolve with AI and be controlled by teams that curate and validate change, not manually running every detail. This will rewrite the enterprise software lifecycle and how organizations think about technical debt, software aging, and architectural design (Martini et al., 2015). Organizations need to prepare for this shift by building internal capabilities, including AI tools, change management, cross-disciplinary collaboration, and long-

term platform strategy. Early adopters of this evolution will have modern, ever-evolving systems ready for any challenge the future may hold.

As illustrated below in Figure 10, AI is being applied across diverse domains—from customer analytics and supply chain management to personalized service delivery and risk mitigation. These expanding use cases underscore the potential of AI to become a central force in the next era of intelligent, adaptive enterprise systems.

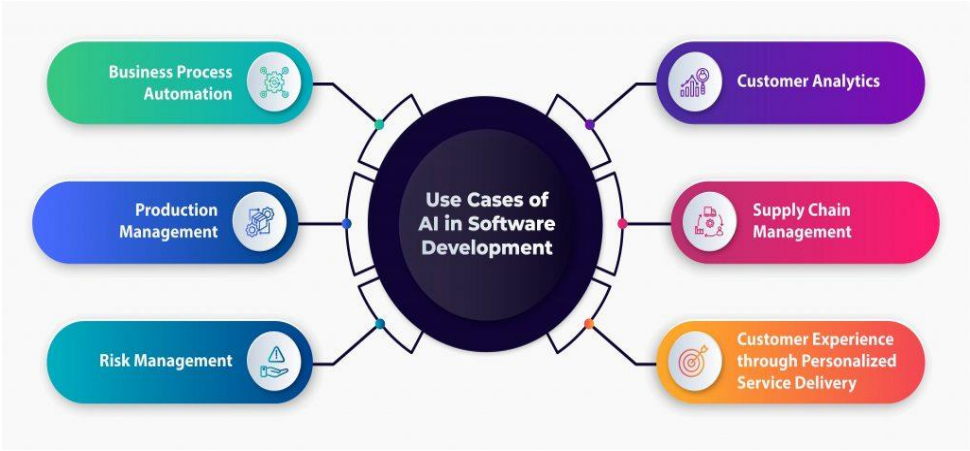

Figure 9: artificial-intelligence-ai-in-software-development

12. CONCLUSION

Legacy system modernization is no longer the domain of the 'technology' company only. Access to data has become a necessity in virtually every industry. Organizations across the spectrum—from finance and healthcare to retail and manufacturing—are under growing pressure to upgrade their aging digital infrastructure to meet evolving customer expectations, security standards, and market conditions. In terms of this, the drift from monolithic architectures to microservices is a technical enhancement and a strategic need to gain operational agility, scalability, and long-term resiliency.

The world has evolved past monolithic applications, once the standard in enterprise environments, to now imposing some major challenges. These systems tend to be large, complex, and deeply entangled, with scaling, updating, and integration with modern cloud-native services being hard. A microservices architecture (or microservices programming model) provides an architectural alternative, breaking rigid systems into smaller, inseparable, discrete components that enable faster development, independent scalability, and better fault tolerance. That transition is not easy, especially in systems that have evolved over a few decades and house the critical business logic. This transformation has become a critical enabler of artificial intelligence. The article outlines various aspects of AI-assisted modernization, beginning with data collection and dependency analysis and culminating in the identification of microservice candidates and the use of code scaffolding tools. Practical examples and tool evaluations demonstrate that AI can significantly reduce manual effort, accelerate project timelines, and enhance the quality of architectural decisions.

Relying solely on the automatic application of AI does not guarantee success. Challenges such as incomplete dependency data, overreliance on a single data source, and resistance to organizational change underscore the importance of intentional deployment planning and human oversight. To maximize effectiveness, AI tools must be

combined with best practices, like incremental refactoring strategies, robust DevOps pipelines, and clearly defined governance models. Cross-functional collaboration and sufficient training of teams to adopt such new technologies and adjust to changing development procedures are just as important. AI is not an alternative to human expertise but an amplifier for modernization. Modernization becomes a responsible, collaborative, business-aligned strategy and becomes structured and scalable. Organizations adopting digital transformation with a balanced blend of both aspects lay the foundation for long-lasting digital growth through modern, future-ready systems that combine to form adaptable, intelligent, better-enabling business outcomes in years to come.

REFERENCES

1. Ahmadvand, M., Pretschner, A., Ball, K., & Eyring, D. (2018). Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. In *Software Technologies: Applications and Foundations: STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers* (pp. 573-588). Springer International Publishing.
2. Akerkar, R. (2019). *Artificial intelligence for business*. Springer.
3. Antal, G., Havas, D., Siket, I., Beszédes, Á., Ferenc, R., & Mihalicza, J. (2016, October). Transforming c++ 11 code to c++ 03 to support legacy compilation environments. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (pp. 177-186). IEEE.
4. Baldwin, C. Y. (2015). Bottlenecks, modules and dynamic architectural capabilities. *Harvard Business School Finance Working Paper*, (15-028).
5. Chavan, A. (2022). Importance of identifying and establishing context boundaries while migrating from monolith to microservices. Journal of Engineering and Applied Sciences Technology, 4, E168. http://doi.org/10.47363/JEAST/2022(4)E168
6. Chavan, A. (2024). Fault-tolerant event-driven systems: Techniques and best practices. Journal of Engineering and Applied Sciences Technology, 6, E167. http://doi.org/10.47363/JEAST/2024(6)E167
7. Crookshanks, E. (2015). *Practical enterprise software development techniques: Tools and techniques for large scale solutions*. Apress.
8. De Santis, S., Florez, L., Nguyen, D. V., & Rosa, E. (2016). *Evolve the Monolith to Microservices with Java and Node*. IBM Redbooks.
9. Delsing, J. (2017). Local cloud internet of things automation: Technology and business model features of distributed internet of things automation solutions. *IEEE Industrial Electronics Magazine*, *11*(4), 8-21.
10. Dhanagari, M. R. (2024). MongoDB and data consistency: Bridging the gap between performance and reliability. *Journal of Computer Science and Technology Studies, 6*(2), 183-198. https://doi.org/10.32996/jcsts.2024.6.2.21
11. Dhanagari, M. R. (2024). Scaling with MongoDB: Solutions for handling big data in real-time. *Journal of Computer Science and Technology Studies, 6*(5), 246-264. https://doi.org/10.32996/jcsts.2024.6.5.20
12. Dickerson, P., & Worthen, J. (2024, May). Optimizing Pipeline Systems for Greater Precision, Efficiency & Safety Using Emerging Technologies. In *PSIG Annual Meeting* (pp. PSIG-2426). PSIG.
13. Fritzsch, J. (2024). Architectural refactoring to microservices: a quality-driven methodology for modernizing monolithic applications.

14. Geetha, R. S., Gowdhamkumar, S., & Jambulingam, S. (2019). Energy challenge, power electronics & systems (PEAS) technology and grid modernization. *International Research Journal of Multidisciplinary Technovation*, *1*(2), 116-129.

15. Goel, G., & Bhramhabhatt, R. (2024). Dual sourcing strategies. *International Journal of Science and Research Archive*, 13(2), 2155. https://doi.org/10.30574/ijsra.2024.13.2.2155

16. Gohil, R., & Patel, H. (2024, June). Comparative Analysis of Cloud Platform: Amazon Web Service, Microsoft Azure, And Google Cloud Provider: A Review. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (pp. 1-5). IEEE.

17. Gu, Q. (2020). *A meta-approach to guide architectural refactoring from monolithic applications to microservices* (Bachelor's thesis).

18. Harris, S. D., & Krueger, A. B. (2015). *A proposal for modernizing labor laws for twenty-first-century work: the" independent worker"* (p. 2015). Washington, DC: Brookings.

19. Jarrahi, M. H. (2018). Artificial intelligence and the future of work: Human-AI symbiosis in organizational decision making. *Business horizons*, *61*(4), 577-586.

20. Karwa, K. (2023). AI-powered career coaching: Evaluating feedback tools for design students. Indian Journal of Economics & Business. https://www.ashwinanokha.com/ijeb-v22-4-2023.php

21. Karwa, K. (2024). Navigating the job market: Tailored career advice for design students. *International Journal of Emerging Business*, *23*(2). https://www.ashwinanokha.com/ijeb-v23-2-2024.php

22. Kibria, M. G., Nguyen, K., Villardi, G. P., Zhao, O., Ishizu, K., & Kojima, F. (2018). Big data analytics, machine learning, and artificial intelligence in next-generation wireless networks. *IEEE access*, *6*, 32328-32338.

23. Kim, S. H., & Lim, Y. J. (2021). Artificial intelligence in capsule endoscopy: A practical guide to its past and future challenges. *Diagnostics*, *11*(9), 1722.

24. Konneru, N. M. K. (2021). Integrating security into CI/CD pipelines: A DevSecOps approach with SAST, DAST, and SCA tools. *International Journal of Science and Research Archive*. Retrieved from https://ijsra.net/content/role-notification-scheduling-improving-patient

25. Kumar, A. (2019). The convergence of predictive analytics in driving business intelligence and enhancing DevOps efficiency. International Journal of Computational Engineering and Management, 6(6), 118-142. Retrieved from https://ijcem.in/wp-content/uploads/THE-CONVERGENCE-OF-PREDICTIVE-ANALYTICS-IN-DRIVING-BUSINESS-INTELLIGENCE-AND-ENHANCING-DEVOPS-EFFICIENCY.pdf

26. Lu, J., Wang, X., Cheng, X., Yang, J., Kwan, O., & Wang, X. (2022). Parallel factories for smart industrial operations: From big AI models to field foundational models and scenarios engineering. *IEEE/CAA Journal of Automatica Sinica*, *9*(12), 2079-2086.

27. Martini, A., Bosch, J., & Chaudron, M. (2015). Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*, *67*, 237-253.

28. McClelland, D. C., & Burnham, D. H. (2017). Power is the great motivator. In *Leadership Perspectives* (pp. 271-279). Routledge.

29. NAEEM SYED, A. A., BAIG, Z., & ZEADALLY, S. (2018). Artificial Intelligence as a Service (AIaaS) for Cloud, Fog and the Edge: State-of-the-Art Practices.

30. Newman, S. (2019). *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media.

31. Nyati, S. (2018). Transforming telematics in fleet management: Innovations in asset tracking, efficiency, and communication. International Journal of Science and Research (IJSR), 7(10), 1804-1810. Retrieved from https://www.ijsr.net/getabstract.php?paperid=SR24203184230

32. Pasquier, T., Han, X., Goldstein, M., Moyer, T., Eyers, D., Seltzer, M., & Bacon, J. (2017, September). Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing* (pp. 405-418).

33. Poth, A., Kottke, M., & Riel, A. (2020). The implementation of a digital service approach to fostering team autonomy, distant collaboration, and knowledge scaling in large enterprises. *Human Systems Management*, *39*(4), 573-588.

34. Rainer, R. K., Prince, B., Sanchez-Rodriguez, C., Splettstoesser-Hogeterp, I., & Ebrahimi, S. (2020). *Introduction to information systems*. John Wiley & Sons.

35. Raju, R. K. (2017). Dynamic memory inference network for natural language inference. International Journal of Science and Research (IJSR), 6(2). https://www.ijsr.net/archive/v6i2/SR24926091431.pdf

36. Rüßmann, M., Lorenz, M., Gerbert, P., Waldner, M., Justus, J., Engel, P., & Harnisch, M. (2015). Industry 4.0: The future of productivity and growth in manufacturing industries. *Boston consulting group*, *9*(1), 54-89.

37. Singh, V. (2022). Advanced generative models for 3D multi-object scene generation: Exploring the use of cutting-edge generative models like diffusion models to synthesize complex 3D environments. https://doi.org/10.47363/JAICC/2022(1)E224

38. Singh, V. (2023). Large language models in visual question answering: Leveraging LLMs to interpret complex questions and generate accurate answers based on visual input. International Journal of Advanced Engineering and Technology (IJAET), 5(S2). https://romanpub.com/resources/Vol%205%20%2C%20No%20S2%20-%2012.pdf

39. Šmite, D., Moe, N. B., Šāblis, A., & Wohlin, C. (2017). Software teams and their knowledge networks in large-scale software development. *Information and Software Technology*, *86*, 71-86.

40. Sukhadiya, J., Pandya, H., & Singh, V. (2018). Comparison of Image Captioning Methods. *INTERNATIONAL JOURNAL OF ENGINEERING DEVELOPMENT AND RESEARCH*, *6*(4), 43-48. https://rjwave.org/ijedr/papers/IJEDR1804011.pdf

41. Suzic, B., & Latinovic, M. (2020, March). Rethinking Authorization Management of Web-APIs. In *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (pp. 1-10). IEEE.

42. Tahir, Y., Khan, I., Rahman, S., Nadeem, M. F., Iqbal, A., Xu, Y., & Rafi, M. (2021). A state-of-the-art review on topologies and control techniques of solid-state transformers for electric vehicle extreme fast charging. *IET power electronics*, *14*(9), 1560-1576.

43. Tapia, F., Mora, M. Á., Fuertes, W., Aules, H., Flores, E., & Toulkeridis, T. (2020). From monolithic systems to microservices: A comparative study of performance. *Applied sciences*, *10*(17), 5797.

44. Xu, Y., Liu, X., Cao, X., Huang, C., Liu, E., Qian, S., ... & Zhang, J. (2021). Artificial intelligence: A powerful paradigm for scientific research. *The Innovation*, *2*(4).

45. Zhong, C., Li, S., Huang, H., Liu, X., Chen, Z., Zhang, Y., & Zhang, H. (2024). Domain-driven design for microservices: An evidence-based investigation. *IEEE Transactions on Software Engineering*.