Architecture for a Multi-Agent AI System for Mainframe to Java Application Conversion

Executive Summary

The modernization of legacy mainframe applications to modern Java-based systems represents a strategic imperative for enterprises seeking enhanced agility, cost efficiency, scalability, and robust security. Mainframes, while historically reliable, present significant challenges in today's rapidly evolving digital landscape due to their inherent complexity, proprietary technologies, and a growing shortage of specialized skills. This report proposes a sophisticated multi-agent Artificial Intelligence (AI) system architecture designed to automate and streamline the intricate process of converting mainframe applications, including their associated data structures and operational scripts, into functionally equivalent and idiomatic Java applications.

The proposed architecture leverages the principles of AI agent specialization, scalability, and fault tolerance, orchestrated primarily through a Blackboard pattern for dynamic collaboration and shared knowledge. Core AI agents, each equipped with advanced AI/ML techniques such as Large Language Models (LLMs), static and dynamic code analysis, and rule-based systems, will collectively manage the entire transformation lifecycle—from deep code understanding and business logic extraction to data schema conversion, automated code generation, and rigorous testing. This comprehensive approach is designed to mitigate critical migration hurdles, including the complexity of legacy code, data integrity risks, skill gaps, and the imperative for continuous business operations. By providing a structured, automated, and human-supervised framework, this multi-agent AI system offers a transformative pathway for organizations to unlock the full potential of their critical business functions in a modern, cloud-native environment.

1. Introduction: The Strategic Imperative for Mainframe Modernization

Mainframe systems have long served as the bedrock of enterprise computing, renowned for their unparalleled performance, reliability, and scalability in managing vast volumes of data processing and high-volume transaction handling tasks.[1] These robust platforms are indispensable for critical business operations, ranging from payroll processing to inventory management, often operating under stringent Service-Level Agreements (SLAs) that dictate precise processing timeframes.[3] The underlying z/Architecture, which defines the system's hardware, software, and input/output (I/O) interactions, along with concepts like Address Spaces that ensure secure isolation and efficient memory management, have historically provided the stability and integrity demanded by core business functions.[3]

Despite their enduring role, legacy mainframe systems face increasing pressure to evolve within the dynamic digital era. The drivers for their modernization are multifaceted, stemming from both economic and operational considerations.

Drivers for Modernization: Agility, Cost Efficiency, Scalability, Security

The sustained operation and maintenance of legacy mainframe infrastructure incur substantial costs, encompassing licensing fees, infrastructure expenditures, and the escalating labor costs associated with specialized mainframe expertise.[4] Modernization offers a compelling pathway to significantly reduce these expenses through optimized resource distribution and the strategic application of AI-powered automation.[4]

Furthermore, contemporary applications demand real-time processing capabilities and dynamic scalability that legacy systems, often built on older programming languages like COBOL, PL/I, and Assembler, struggle to provide.[4] Modernization facilitates seamless integration with cloud architectures, enabling scalable storage solutions and virtualization capabilities that combine the inherent strengths of legacy systems with the flexibility of cloud platforms.[4]

The imperative for business agility and innovation is another significant driver. Organizations require adaptable systems to support rapid digital transformation, a challenge when foundational applications are rooted in less flexible, older languages.[4]

Modernization, particularly through API-driven integration with cloud-native services, enhances application functionality and fosters greater agility, paving the way for long-term innovation.[4]

Finally, security and compliance concerns are paramount. Legacy systems may lack the advanced security measures and compliance capabilities mandated by today's regulatory landscape.[2] Modern architectural frameworks inherently provide more robust security features and comprehensive support for regulatory compliance, including standards such as HIPAA, PCI-DSS, GDPR, and SOX.[2]

A deeper examination of these modernization drivers reveals their profound interconnectedness. The high maintenance costs associated with legacy systems often originate from the scarcity of specialized skills required to manage them and the inherent rigidity of older, less adaptable architectures. This lack of architectural flexibility impedes innovation, which in turn can impact an organization's competitiveness and revenue potential. Similarly, security vulnerabilities in legacy systems are frequently exacerbated by the difficulty of applying timely patches and updates, making compliance more challenging and increasing overall risk exposure. This intricate web of dependencies suggests that addressing one modernization driver, such as cost reduction, frequently yields positive cascading effects across others, including an improved security posture through modern frameworks and increased agility from refactoring efforts. Consequently, a comprehensive modernization strategy, especially one empowered by a multi-agent AI system, can deliver synergistic benefits rather than isolated improvements, with the system's capacity to automate complex tasks across these dimensions being central to realizing this synergy.

2. Understanding the Mainframe Landscape: Core Components and Application Structures

A thorough understanding of the mainframe environment is critical for any successful modernization initiative. Mainframes are sophisticated computing platforms comprising a diverse array of hardware and software components, each playing a vital role in their operation.

Mainframe Hardware and Software Architecture

At the foundational level, mainframes integrate powerful hardware components, including Central Processing Units (CPUs), substantial Memory (RAM), Input/Output (I/O) Channels, and various Storage mechanisms like disk and tape drives.[1] These elements are orchestrated by the z/Architecture, which dictates how the entire system, encompassing hardware, software, and their interactions, is designed and functions.[3] System administrators interact with and monitor the mainframe primarily through Consoles.[1]

The core software component is the Operating System (OS), with z/OS being the predominant choice. The OS manages hardware resources, schedules tasks, and provides the essential platform for applications to execute. Other notable mainframe operating systems include z/VM, Linux on Z, and z/VSE.[1] The Base Control Program (BCP) within the OS is responsible for managing I/O operations and providing fundamental system control.[3]

Virtualization is a key feature of mainframes, implemented through Logical Partitions (LPARs). LPARs enable the partitioning of mainframe resources, allowing different departments or users to securely share the system's capabilities while maintaining isolation.[1] This resource partitioning is complemented by the z/OS Workload Manager (WLM), which prioritizes specific tasks to ensure efficient processing within predefined timeframes, thereby adhering to critical Service-Level Agreements (SLAs).[3]

Security is a paramount concern on mainframes, addressed by robust mechanisms such as the Resource Access Control Facility (RACF). RACF provides identification, authentication, access control, and logging services, working in conjunction with the System Authorization Facility (SAF) to deliver strong security capabilities. A comprehensive understanding of RACF's authorization and access control mechanisms is essential for maintaining a secure mainframe environment.[3]

For high availability and continuous operations, IBM developed Parallel Sysplex technology. This allows multiple mainframes to function cohesively as a single system, sharing workloads and resources, ensuring that if one mainframe fails, others can seamlessly take over without interrupting critical business processes.[1]

Key Mainframe Application Technologies

Mainframe applications are typically built using specific programming languages and interact with specialized data management and transaction processing systems.

COBOL and PL/I: These are the primary programming languages for most legacy mainframe applications. Programs written in COBOL or PL/I often contain intricate business logic and extensive interdependencies, making their analysis and migration a complex undertaking.[4]

VSAM (Virtual Storage Access Method): A high-performance file management system unique to mainframes. VSAM files come in several structures:

- **Entry Sequenced Data Set (ESDS):** Records are stored and accessed strictly in the order they were entered, supporting only sequential access.[8]
- **Key Sequenced Data Set (KSDS):** Records are organized by a primary key, allowing sequential, random, or dynamic (mixed) access based on key values.[8]
- **Relative Record Data Set (RRDS):** Records are accessed by their relative record number, also supporting sequential, random, or dynamic access.[8]

  Understanding these access methods and file organizations is crucial for correctly transforming data access logic to a Java environment.

DB2: IBM's relational database management system, widely used on mainframes for managing large volumes of structured data.[7] COBOL programs often embed SQL statements to interact with DB2.[11]

IMS DB (Information Management System Database): A hierarchical database system where data is structured and accessed in a parent-child hierarchy of segments and fields.[12] A database record in IMS DB comprises a single root segment and all its dependent segments. IMS DB supports up to 255 different segment types and 15 levels of hierarchy.[13] IMS also includes Main Storage Databases (MSDBs), which are memory-resident for high performance, with specific characteristics regarding updates and ownership.[12]

IMS TM (Information Management System Transaction Manager): Often used in conjunction with IMS DB, IMS TM manages transactions, ensuring data integrity and efficient processing in high-volume environments.[7]

CICS (Customer Information Control System): A transaction management system on z/OS that facilitates online transaction processing. CICS applications frequently employ a technique called pseudo-conversational processing. Unlike traditional conversational transactions that hold system resources while waiting for user input, pseudo-conversational transactions release resources during user think time, giving the appearance of a continuous conversation while conserving valuable mainframe resources.[14] This involves the client initiating each step of the conversation as a new, non-conversational transaction.[15]

JCL (Job Control Language): A scripting language used to define and control batch jobs on IBM mainframe systems. JCL specifies the program to execute, the input and output files required (via DD statements), and any runtime parameters.[11] JCL explicitly defines all required datasets and devices, which aids the scheduler in pre-allocating resources and preventing deadlocks.[16] JCL adheres to strict syntax rules, including mandatory column positioning, specific comment formats, and an 80-character line limit.[11]

The deep intertwining of mainframe application logic and infrastructure presents a significant challenge. The unique characteristics of components like IMS's hierarchical data model, CICS's pseudo-conversational design, and JCL's explicit resource allocation are not merely implementation details. They are fundamental architectural patterns driven by the mainframe's resource constraints and operational philosophy, representing how business logic was originally expressed and optimized for that specific environment. A direct, literal translation of COBOL code to Java without a profound understanding and re-architecting of these underlying mainframe-specific patterns—including data access, transaction management, and batch processing—would result in a Java application that is inefficient, difficult to maintain, and incapable of leveraging modern cloud-native capabilities. Therefore, the multi-agent system must perform a semantic transformation and architectural pattern mapping, extending beyond mere syntactic translation. This necessitates agents capable of analyzing not only the code itself but also its intricate interactions with the mainframe's unique middleware, operating system, and resource management systems.

Table 1: Key Mainframe Components and Their Modernization Equivalents/Strategies

Mainframe Component	Description	Modernization Equivalent/Strategy	Relevant Source
COBOL, PL/I, Assembler	Primary programming languages for business applications.	Java (Spring Boot, Microservices)	4
VSAM ESDS	Entry Sequenced Data Set; sequential file access.	Relational Database Table (e.g., PostgreSQL, MySQL, Oracle) with sequential access patterns, or flat files for simple data transfer.	8
VSAM KSDS/RRDS	Key Sequenced Data Set (indexed) / Relative Record Data Set (direct access).	Relational Database Table with appropriate indexing, or NoSQL Database (e.g., Cassandra, MongoDB) for key-value access.	8
IMS DB	Hierarchical database system.	Relational Database (transform hierarchy to relational schema), or Document Database (e.g., MongoDB) for hierarchical data.	7
CICS Pseudo-conversational	Transaction monitor for online processing, uses pseudo-conversational technique to save	RESTful APIs, Microservices, Asynchronous message queues (e.g., Kafka,	7

	resources.	RabbitMQ) for managing state across interactions.	
JCL Batch Job	Scripting language for batch processing and resource allocation.	Modern Schedulers (e.g., Spring Batch, Apache Airflow, Kubernetes Jobs), Container Orchestration (e.g., Kubernetes), Cloud Batch Services.	7
DB2 on z/OS	Relational database management system.	Cloud-native Relational Databases (e.g., AWS RDS, Azure SQL Database, Google Cloud SQL), or other enterprise RDBMS.	7
RACF / SAF	Security mechanisms for identification, authentication, access control.	Cloud Identity and Access Management (IAM) services, OAuth2, OpenID Connect.	3
z/OS WLM	Workload Manager for prioritizing and managing tasks.	Cloud workload management services, Kubernetes resource management, autoscaling groups.	3

3. Navigating the Migration Journey: Key Challenges from Mainframe to Java

The transition from mainframe to Java applications is a complex undertaking, fraught with technical, operational, and organizational hurdles. A clear understanding of these challenges is essential for designing an effective multi-agent AI solution.

Technical Complexities

Legacy Code Complexity: Mainframe COBOL programs, and those in other older languages, often contain highly intricate logic, extensive interdependencies between modules, and a notable absence of comprehensive documentation.[2] This inherent complexity, coupled with platform-specific customizations and proprietary or obsolete middleware, makes the analysis, refactoring, and subsequent migration process exceptionally intricate and time-consuming.[2]

Data Conversion and Validation: The structures of data on mainframes, particularly within VSAM files, IMS DB, and DB2 schemas, differ substantially from those found in modern relational or NoSQL databases. A critical challenge lies in ensuring accurate data conversion, precise schema mapping, and robust validation throughout the migration.[2] This often involves addressing schema mismatches, incompatibilities in data formats, and potential errors in transformation logic.[2]

Integration with Existing Systems: Modernizing an application extends beyond merely converting its internal code; it necessitates seamless integration with other systems and technologies already in use across the organization. This can frequently lead to compatibility issues between the new Java application and legacy or third-party systems.[6]

Operational and Organizational Hurdles

Lack of Expertise and Skill Gaps: A significant challenge is the scarcity of developers proficient in both legacy mainframe technologies (such as COBOL, JCL, CICS, and DB2) and modern Java/cloud development paradigms.[2] Many experienced mainframe engineers are nearing retirement, exacerbating this skill gap within the workforce.[2]

Downtime and Business Continuity Concerns: Mainframe systems underpin mission-critical business activities, meaning that any disruption or downtime during the migration process can have severe repercussions on an organization's profitability and brand reputation. The paramount objective during migration is to achieve near-zero downtime.[2] Real-world risks include broken integrations with downstream systems, delays in data synchronization, application compatibility issues, and performance degradation in the new environment.[2]

Security and Compliance Issues: Ensuring the integrity and confidentiality of sensitive data during migration and maintaining compliance with stringent regulatory mandates (such. as HIPAA, PCI-DSS, GDPR, and SOX) in the new Java environment is a major obstacle.[2] Legacy COBOL systems may lack the robust security measures required in today's technological landscape.[6]

Testing and Quality Assurance: Guaranteeing the reliability and stability of the migrated application is crucial. This necessitates comprehensive testing and quality assurance measures, encompassing unit, integration, and regression testing, to identify and resolve any issues before deployment.[6]

Cost Considerations: Mainframe migration is a substantial undertaking that can incur significant costs. Budgetary constraints often pose a hurdle to the project's progress and success, underscoring the need for a thorough cost-benefit analysis.[6]

The strong interdependency between human resources and technological complexities in migration challenges is a critical aspect. The persistent shortage of skilled professionals is not merely a staffing problem; it directly amplifies the technical difficulties. Without sufficient human expertise in COBOL and the intricacies of mainframe environments, understanding complex legacy code, systems lacking documentation, and proprietary middleware becomes exponentially more difficult. This human limitation renders the technical aspects of data conversion, mapping interdependencies, and re-architecting significantly more susceptible to errors, delays, and increased costs. This profound interrelationship means that an AI solution for mainframe modernization must not only automate technical tasks but also, through user-friendly interfaces and clear explanations, effectively function as an "expert system" that compensates for human skill shortages. The multi-agent system's capacity to analyze, interpret, and transform complex legacy artifacts efficiently thus acts as a force multiplier for organizations contending with a diminishing pool of mainframe specialists.

4. Foundations of Multi-Agent AI Systems for Complex Problem Solving

Multi-agent systems (MAS) represent a powerful design paradigm where multiple autonomous agents collaborate to achieve a common goal. This approach offers distinct advantages over monolithic single-agent systems, particularly when addressing complex, distributed, and real-time problems like mainframe modernization.

Advantages of Multi-Agent Architectures

Specialization: A primary benefit of MAS is the ability to specialize each agent for a distinct task.[17] This prevents a single, all-encompassing agent from becoming overwhelmed or inefficient when confronted with a complex, multifaceted problem. By mirroring human teamwork, complex problems are broken down into specialized units of work, allowing each agent to focus its capabilities where it is most effective.[18]

Scalability: Multi-agent systems inherently offer superior scalability. It is far simpler to expand system capabilities by adding more specialized agents rather than attempting to overload or re-engineer a single, monolithic agent.[17] This modularity facilitates growth and adaptation.

Fault Tolerance: The distributed nature of MAS enhances system reliability. If one agent encounters a failure, other agents can continue functioning, thereby ensuring the overall system's continuity and robustness.[17] This distributed control contributes to emergent cooperative behavior and resilience.[19]

Decentralization and Autonomy: Agents within a MAS maintain a degree of autonomy, performing local computations and making independent decisions. They collaborate by exchanging partial knowledge about their environment, enabling decentralized control and overcoming the inherent limitations of constrained single-agent systems, especially in complex and distributed environments.[19]

Core Building Blocks: Agents, Environments, Communication, Coordination, Knowledge

The effective functioning of a multi-agent system relies on several fundamental building blocks:

- **Agents:** These are the autonomous entities within the system, each possessing local knowledge and decision-making capabilities.[19]
- **Environment:** This refers to the shared context or problem space in which the agents operate and interact.
- **Communication:** Communication is paramount, enabling agents to share information, coordinate their actions, and collectively make decisions.[19] This can occur explicitly through message passing or implicitly through modifications to a shared environment.[19]
- **Coordination Mechanisms:** Agents must coordinate their actions to ensure the collective achievement of a common goal. This involves strategies for decomposing work, distributing resources, resolving conflicts, and engaging in cooperative planning.[17]
- **Knowledge Management:** Agents share partial knowledge about their environment and collaborate on decision-making.[19] A Shared Context and Memory Store provides a common space for storing data, intermediate outputs, and decisions, ensuring agents are aware of each other's contributions and maintain continuity throughout their workflow.[19]

Relevant Multi-Agent Orchestration Patterns

Various patterns exist for orchestrating multiple agents, each suited to different coordination requirements:

- **Centralized, Decentralized, and Hybrid Architectures:** These represent different levels of control and distribution within the system, chosen based on the specific use case.[17]
- **Blackboard Architecture:** A highly relevant general framework characterized by

a shared information space, known as the "blackboard," which is accessible and visible to all agents.[21] A control unit dynamically selects suitable agents based on the current content of the blackboard and the problem query. Selected agents then contribute their outputs to the blackboard. This iterative selection and execution process continues until a solution consensus is achieved or a maximum number of rounds is met.[22] This pattern allows for dynamic agent selection and flexible communication, making it particularly effective for complex problems where predefined workflows are unavailable.[22]

- **Sequential Orchestration:** In this pattern, agents process tasks one after another, with the outputs or results from one agent serving as inputs for the next.[18]
- **Concurrent Orchestration (Fan-out/Fan-in):** This approach involves running multiple AI agents simultaneously on the same task. Each agent provides independent analysis or processing from its unique perspective or specialization, which can significantly reduce overall run time and provide comprehensive coverage of the problem space.[18]
- **Handoff Orchestration:** This pattern is useful when tasks need to be transferred between agents based on predefined rules, commonly applied in scenarios like workflow automation.[17]
- **Magentic Orchestration:** An extension of the group chat pattern, where a central agent constructs a plan, and other agents utilize tools to enact changes in external systems, rather than solely relying on their internal knowledge stores. This pattern is well-suited for complex or open-ended use cases where no predetermined solution path exists and requires input and feedback from multiple specialized agents.[18]
- **Contract Net Protocol (CNP):** A task-sharing protocol used to dynamically allocate tasks among autonomous agents, operating similarly to sealed auction protocols.[23]

The Blackboard architecture stands out as a particularly compelling choice for enabling iterative refinement in ambiguous problem domains. Mainframe modernization is rarely a straightforward, deterministic process. Legacy code often lacks comprehensive documentation, contains implicit business rules, and exhibits complex interdependencies. This renders the problem inherently "complex or open-ended," where "no predetermined solution path" exists.[18] Traditional sequential or simple hand-off patterns might struggle with the inherent ambiguities and the iterative refinement necessary to untangle such complexities. The Blackboard's capacity to allow multiple specialized agents—such as those focused on code analysis, logic extraction, or data transformation—to contribute, refine, and even resolve conflicts on a shared, evolving state (the blackboard) is uniquely suited for addressing these ill-defined problems. This architecture facilitates an "emergent cooperative behavior" [19] where the collective intelligence of specialized agents, through iterative contributions to a shared context, can uncover and resolve complexities in legacy code that would be exceedingly difficult for a single agent or a rigid sequential workflow. This makes the Blackboard a highly promising pattern for the core orchestration of a mainframe modernization multi-agent system.

5. Proposed Multi-Agent AI Architecture for Mainframe to Java Conversion

The proposed multi-agent AI system for mainframe to Java conversion is designed as a hybrid architecture, primarily leveraging a Blackboard pattern for central coordination and shared knowledge, complemented by sequential and concurrent orchestrations for specific sub-tasks. This design aims to provide a robust, flexible, and scalable solution for complex enterprise modernization.

5.1. System Overview and High-Level Flow

The overall system flow commences with the ingestion of various mainframe artifacts, including COBOL/PL/I source code, JCL scripts, VSAM file definitions, IMS DB/DB2 schemas, and CICS transaction definitions. This initial input is then subjected to a series of iterative phases: deep analysis, semantic transformation, Java code generation, and rigorous validation. The culmination of this process is the production of deployable Java applications, modernized data schemas, and associated operational artifacts. Throughout this complex workflow, strategic human-in-the-loop interaction points are integrated to provide essential oversight, facilitate validation, and enable human experts to intervene for complex decision-making or to refine agent outputs.

5.2. Core AI Agents and Their Specializations

The efficacy of the multi-agent system stems from the specialization of its constituent AI agents, each designed to address a particular facet of the mainframe modernization challenge.

Code Analysis Agent:
This agent is responsible for comprehending the intricate structure, logical flow, and dependencies embedded within mainframe source code, including COBOL, PL/I, and JCL. It employs a combination of techniques:

- **Static Code Analysis:** Examines source code without execution to identify syntax errors, violations of coding standards, potential security vulnerabilities, and to map control and data flow.[24] This is particularly valuable for large and complex codebases, allowing rapid identification of issues.[25]
- **Dynamic Code Analysis:** Involves executing the mainframe code (or a simulated environment) to observe its runtime behavior, identify performance bottlenecks, and detect issues that may not be apparent from static inspection, especially interactions with external systems.[24]
- **Rule-Based Systems:** Applies predefined rules and best practices to identify common mainframe patterns, such as CICS pseudo-conversational calls, specific VSAM access methods, and IMS DB interactions.[24]
- **Natural Language Processing (NLP) and Large Language Models (LLMs):** Utilizes these models for deeper semantic understanding of the code, recognizing complex patterns, and inferring undocumented business logic.[24]

Data Schema Transformation Agent:
This agent analyzes mainframe data structures, including VSAM file layouts, IMS DB hierarchical definitions, and DB2 schemas. Its role is to propose optimal, modernized relational or NoSQL schemas for the target Java applications. It also manages the intricate data mapping and transformation logic required for data migration.

- **Techniques:** Semantic analysis of data definitions (e.g., COBOL data divisions, IMS Database Descriptions (DBDs) [7]). Rule-based mapping for common mainframe data types to modern equivalents. LLMs are employed to infer complex relationships within hierarchical data and propose efficient schema designs. The agent also generates specifications for Extract, Transform, Load (ETL) processes necessary for data migration.[2]

Business Logic Extraction & Refactoring Agent:

This agent specializes in identifying discrete business rules and core logic often deeply embedded within complex, monolithic legacy code. It then suggests modularization and refactoring strategies to align the extracted logic with modern microservices principles.

- **Techniques:** LLMs are instrumental in understanding the nuanced logic of the code and generating natural language explanations, aiding human comprehension.[25] Static analysis helps identify areas of high code complexity and intricate dependencies.[2] AI-driven tools provide refactoring suggestions based on quantitative complexity metrics.[27]

Code Generation & Translation Agent (COBOL/PL/I to Java):
This is a core transformation agent responsible for converting the analyzed mainframe code into functionally equivalent, idiomatic Java code. It handles language-specific constructs and aims to integrate with modern Java frameworks.

- **Techniques:** LLMs are central for cross-language code translation and generation.[26] The agent can be customized with specific prompts to adhere to desired workflows and coding styles.[26] It leverages automated code generation tools.[26] A critical consideration is to augment LLMs with specialized knowledge to overcome their limitations regarding domain-specific IBM mainframe middleware, such as CICS or IMS, which they may not inherently understand from general training data.[29]

Testing & Validation Agent:
This agent ensures the correctness, reliability, and performance of the migrated Java application. It generates comprehensive test cases (unit, integration, regression) based on the extracted business logic and observed runtime behavior of the original mainframe application. It then executes these tests and validates the migrated Java application against the mainframe baseline.

- **Techniques:** Utilizes automated testing tools.[27] AI solutions are employed to generate test cases based on real usage patterns, enhancing test coverage.[27] Dynamic code analysis helps identify runtime errors and performance issues in the translated code.[25] LLMs can act as Judges (LaaJs) to assess the overall translation quality, semantic fidelity, and preservation of logic. These LaaJs are complemented by analytic checkers that rigorously detect hallucinated elements and ensure syntactic and semantic accuracy.[29] Integrated performance testing and optimization techniques are applied to ensure the migrated application meets performance expectations.[6]

Integration & API Generation Agent:
This agent focuses on designing and generating the necessary APIs to facilitate the integration of the modernized Java applications with other internal or external systems. It

handles the transformation of mainframe communication protocols and middleware interactions to modern equivalents.

- **Techniques:** Employs robust API generation tools. LLMs can suggest optimal integration patterns. Rule-based systems are crucial for mapping mainframe-specific integration points (e.g., MQ, SNA) to modern protocols such as REST or Kafka.[6]

Orchestration & Control Agent:
This agent serves as the conductor of the entire multi-agent system, managing the overall workflow, task sequencing, and dynamic selection of agents. It is also responsible for conflict resolution within the system and monitoring progress against migration goals.

- **Techniques:** Functions as a flow orchestrator, defining execution logic, task sequencing, branching, and handling errors and retries.[19] It dynamically selects suitable agents based on the evolving content of the Blackboard and the current problem query.[22] It may incorporate a conflict-resolver sub-agent to detect contradictions between agent outputs and prompt further discussion or refinement among involved agents.[22] A cleaner sub-agent helps maintain the quality and relevance of information on the Blackboard, reducing token consumption.[22]

Table 2: Proposed AI Agents, Their Roles, and Key AI/ML Techniques

AI Agent	Primary Role in Migration	Key AI/ML Techniques Employed	Relevant Source
Code Analysis Agent	Understands mainframe code structure, logic, and dependencies (COBOL, PL/I, JCL).	Static Code Analysis, Dynamic Code Analysis, Rule-Based Systems, NLP, Large Language Models (LLMs)	[24]
Data Schema Transformation Agent	Analyzes mainframe data structures (VSAM, IMS DB, DB2) and proposes modern schemas,	Semantic Analysis, Rule-Based Mapping, LLMs for schema design, ETL Process Generation	[2]

	manages data mapping.		
Business Logic Extraction & Refactoring Agent	Identifies discrete business rules, suggests modularization and refactoring for microservices.	LLMs for code understanding/explanation, Static Analysis for complexity, AI-driven Refactoring Suggestions	2
Code Generation & Translation Agent	Converts mainframe code (COBOL/PL/I) into idiomatic Java code.	LLMs for Cross-Language Translation and Generation, Automated Code Generation Tools, Custom Prompts	26
Testing & Validation Agent	Generates and executes test cases, validates correctness and performance of migrated Java application.	Automated Testing Tools, AI-driven Test Case Generation, Dynamic Code Analysis, LLMs as Judges (LaaJs), Analytic Checkers, Performance Testing	6
Integration & API Generation Agent	Designs and generates APIs for modernized Java applications, handles protocol transformation.	API Generation Tools, LLMs for Integration Patterns, Rule-Based Systems for Protocol Mapping	6
Orchestration & Control Agent	Manages overall workflow, task sequencing, agent selection, and conflict resolution.	Flow Orchestration, Dynamic Agent Selection, Conflict Resolution Algorithms, Cleaner Agents	19

5.3. Orchestration and Communication Mechanisms

The system adopts a hybrid orchestration pattern, with the Blackboard architecture serving as the foundational coordination mechanism. The Blackboard acts as the central nervous system, providing a dynamic, shared memory space where all agents can post observations, partial solutions, and requests for further processing.[21] This shared space facilitates dynamic, context-dependent agent selection and fosters comprehensive information exchange, which is crucial for complex, iterative problems.[22]

Within this Blackboard-centric framework, other orchestration patterns are selectively applied:

- **Sequential orchestration** is utilized for well-defined sub-workflows where one task logically precedes another, such as the progression from code analysis to business logic extraction and then to code generation.
- **Handoff orchestration** facilitates the seamless transfer of specific tasks or artifacts between agents. For example, the Code Analysis Agent might hand off identified business logic components to the Business Logic Extraction Agent for further refinement.[17]
- **Concurrent orchestration** can be employed for tasks that benefit from multiple perspectives or parallel processing. This might involve multiple LLMs or specialized analysis tools simultaneously working on the same code segment to derive diverse insights or accelerate processing time.[18]

Inter-agent communication primarily occurs through the Blackboard. Agents post and retrieve structured data, such as Abstract Syntax Trees (ASTs), data mappings, refactoring suggestions, and test results, ensuring that all relevant information is centrally available and consistently updated.[22] When direct communication between specific agents is necessary, standardized message passing protocols are utilized, for instance, an explicit request from the Orchestration & Control Agent to a particular specialized agent. The shared context and memory store maintained on the Blackboard are vital for ensuring continuity across the workflow and enabling agents to remain aware of each other's contributions and the overall project state.[19]

5.4. Knowledge Management and Shared Context

Effective knowledge management is paramount for the multi-agent system's ability to navigate the complexities of mainframe modernization. The Blackboard itself functions as a dynamic, shared memory, effectively replacing individual agent memory modules and promoting a more comprehensive exchange of information among agents.[22] This shared knowledge base stores the evolving state of the migration project, including detailed representations of the source code, granular extracted business rules, proposed Java code, transformed data schemas, and comprehensive test results.

Crucially, this shared knowledge base is augmented with domain-specific knowledge about mainframe technologies. This includes detailed information on CICS API calls, VSAM file structures, JCL command syntax, and IMS DB intricacies. This augmentation is vital for overcoming inherent limitations of general-purpose LLMs, which may lack deep, specialized understanding of these niche mainframe middleware components.[29]

Large Language Models (LLMs) are integral to the system's ability to understand context and perform complex reasoning. They assist in navigating changing contexts and filling knowledge gaps during collaborative tasks.[19] LLMs contribute significantly to the Blackboard by providing outputs such as detailed code explanations, intelligent refactoring suggestions, and nuanced translation proposals. Their capacity to understand the structure and logic of code at a complex level is a cornerstone of the system's analytical capabilities.[25]

5.5. AI/ML Techniques Employed Across the System

The proposed architecture integrates a diverse set of AI/ML techniques, each contributing uniquely to the overall transformation process.

Deep Dive into LLMs for Code Understanding and Generation:
LLMs, trained on vast datasets of code, are fundamental to the system's ability to recognize patterns, identify potential issues, and generate new code.24 They are particularly crucial for cross-language translation, enabling the conversion of COBOL or PL/I code into semantically

equivalent Java.26 Beyond mere translation, LLMs can explain complex code segments in natural language and generate human-like comments, significantly aiding human reviewers in understanding the transformed code.25 However, it is important to acknowledge that LLMs can "hallucinate," producing plausible but incorrect outputs, and may lack deep domain-specific knowledge concerning proprietary IBM mainframe middleware like CICS.29 This necessitates their outputs being rigorously validated and augmented with specialized knowledge bases and other AI techniques.

Application of Static and Dynamic Analysis for Quality and Security:

- **Static Code Analysis:** This technique involves analyzing the source code without executing it. It is invaluable for early detection of syntax errors, violations of coding standards, and potential security vulnerabilities.[24] For large and complex codebases, static analysis can rapidly scan thousands of lines of code, pinpointing issues and providing detailed reports.[25]
- **Dynamic Code Analysis:** In contrast, dynamic analysis executes the code to observe its behavior at runtime. This allows the system to identify runtime errors, performance bottlenecks, and security vulnerabilities that might not be apparent from static inspection alone.[24] It provides a more complete picture of how the code interacts with external systems and resources.[25]

Role of Rule-Based Systems and Expert Knowledge:
Rule-based systems apply predefined rules and best practices for consistent code analysis, ensuring adherence to established industry standards and company guidelines.24 These systems are critical for encoding specific mainframe patterns, such as the nuances of JCL syntax, VSAM access patterns, or CICS transaction flows. They also enforce security principles like "Defense in Depth" and "Least Privilege".3 Their deterministic nature provides a reliable baseline for analysis and can automatically correct applications to ensure good code quality.24 Furthermore, rule-based systems are essential for augmenting LLMs by providing the precise, domain-specific mainframe knowledge that LLMs might lack, thereby mitigating the risk of hallucinations in critical areas.29

The necessity of a hybrid AI approach for robustness cannot be overstated. No single AI technique is sufficient to address the multifaceted complexities of mainframe modernization. While Large Language Models are powerful for understanding context, generating code, and providing high-level explanations, they are susceptible to inaccuracies (hallucinations) and may lack deep, specific domain knowledge, particularly regarding niche mainframe middleware. Conversely, rule-based systems and traditional static/dynamic analysis, while less flexible in their application, provide deterministic accuracy for known patterns, security vulnerabilities, and adherence to coding standards. This combination ensures that the "creative" and "understanding" capabilities of LLMs (e.g., initial translation, refactoring suggestions) are balanced by the "rigor" and "verification" provided by rule-based systems and analytical tools (e.g., ensuring security compliance, identifying specific mainframe-to-Java anti-patterns, validating data transformations). The concept of using LLMs as Judges (LaaJs) further reinforces this by employing LLMs for holistic quality assessment in conjunction with analytical checkers, maximizing both the breadth of understanding and the precision of the transformation. This layered approach is fundamental to building a reliable and effective modernization system.

6. Addressing Key Migration Challenges with the Multi-Agent AI System

The proposed multi-agent AI architecture is strategically designed to directly confront and mitigate the most significant challenges inherent in mainframe to Java migration.

Mitigating Legacy Code Complexity and Interdependencies
The Code Analysis Agent, utilizing both static and dynamic analysis, is specifically engineered to identify and map complex logical flows, intricate dependencies, and undocumented sections within legacy COBOL and PL/I code.2 Following this deep understanding, the Business Logic Extraction & Refactoring Agent takes over, systematically modularizing and refactoring the identified business logic. This process simplifies the underlying codebase, making it more amenable to modern Java paradigms and reducing the extensive time and effort typically required for manual analysis and restructuring.6 The iterative nature of the Blackboard architecture allows different agents to contribute to understanding and transforming these complexities, significantly reducing manual effort and potential errors.
Ensuring Data Integrity and Schema Compatibility
Data migration and ensuring integrity are critical aspects. The Data Schema Transformation Agent is tasked with analyzing the diverse mainframe data structures (VSAM, IMS DB, DB2) and developing comprehensive data mapping and validation strategies. This directly addresses common concerns such as schema mismatches, data format incompatibilities, and errors in transformation logic.2 The system can automate the generation of Extract, Transform, Load (ETL) processes, streamlining the migration of structured data to modern database systems while maintaining data fidelity.2
Enhancing Security and Compliance in the Modernized Application
Security and compliance are non-negotiable in enterprise migrations. The Code Analysis Agent and the Testing & Validation Agent work in concert to incorporate industry-standard

security practices, including encryption, secure authentication, and robust access controls, into the migrated Java application.6 They also conduct thorough security audits to identify and address vulnerabilities. Furthermore, rule-based systems within the architecture are crucial for ensuring strict adherence to specific compliance mandates, such as HIPAA, PCI-DSS, GDPR, and SOX, by embedding relevant checks and validations throughout the transformation process.2
Streamlining Testing and Quality Assurance
The reliability and stability of the migrated application are paramount. The Testing & Validation Agent plays a central role by generating automated test cases—including unit, integration, and regression tests—based on the extracted business logic and observed behavior of the original mainframe system.6 This automation significantly streamlines the testing process. The integration of LLMs as Judges (LaaJs) alongside traditional analytic checkers provides a multi-faceted approach to quality assurance, ensuring both the functional correctness and the semantic fidelity of the translated code.29 Additionally, the system incorporates performance testing and optimization techniques to guarantee that the migrated application meets or exceeds expected performance benchmarks.6

The automation provided by the multi-agent AI system directly addresses the pervasive skill gap in mainframe modernization by encapsulating mainframe expertise within the agents themselves. For instance, the Code Analysis Agent's ability to understand intricate JCL syntax or the nuances of CICS pseudo-conversational patterns effectively embeds this specialized knowledge. This significantly reduces the reliance on scarce human experts for tedious, error-prone manual analysis and translation. Furthermore, by automating complex transformations and enabling robust automated testing, the system substantially reduces the risk of errors and facilitates advanced deployment strategies such as phased, blue-green, or parallel run deployments.[2] This approach minimizes downtime, thereby ensuring business continuity, which is a top priority for organizations operating mission-critical mainframe systems. The multi-agent AI system thus transcends being merely a "code translator"; it becomes a strategic tool for organizational resilience, enabling enterprises to modernize critical systems even when faced with significant human resource constraints and extreme demands for uninterrupted operations. It effectively scales human expertise through AI.

Table 3: Mainframe Migration Challenges Addressed by the Multi-Agent AI System

Mainframe Migration	How Multi-Agent AI System	Relevant Source

Challenge	Addresses It	
Legacy Code Complexity & Interdependencies	Code Analysis Agent identifies complex logic. Business Logic Extraction & Refactoring Agent modularizes and simplifies. Blackboard facilitates iterative understanding.	2
Data Conversion & Validation	Data Schema Transformation Agent maps schemas, generates ETL. Automated validation ensures integrity.	2
Lack of Expertise / Skill Gaps	AI agents encapsulate mainframe knowledge, automating tasks that require specialized skills, reducing reliance on scarce human experts.	2
Downtime & Business Continuity Risks	Automated testing, phased/blue-green deployments, and parallel run capabilities minimize disruption and ensure continuous operations.	2
Security & Compliance Issues	Code Analysis and Testing Agents enforce security practices. Rule-based systems ensure adherence to compliance mandates.	2
Integration with Existing Systems	Integration & API Generation Agent designs and creates modern APIs, facilitating seamless interoperability.	6

Testing & Quality Assurance	Testing & Validation Agent generates comprehensive automated test cases (unit, integration, regression) and uses LaaJs for quality assessment.	[6]
Cost Considerations	Automation reduces manual effort, accelerates migration timelines, and lowers long-term maintenance costs, improving ROI.	[4]

7. Implementation Strategy and Phased Approach

The successful deployment of a multi-agent AI system for mainframe modernization requires a well-defined implementation strategy that emphasizes iterative development, careful deployment, and continuous human oversight.

Iterative Development and Deployment

A phased migration approach is highly recommended, beginning with non-priority applications. These initial projects serve as crucial learning opportunities, allowing for vigorous development and testing of the multi-agent system's capabilities before moving to more critical systems.[2] This iterative development cycle for the multi-agent system itself allows for continuous refinement of agent capabilities, optimization of orchestration patterns, and adaptation to unforeseen complexities in legacy environments.

For deployment, strategies such as blue-green deployment methods and parallel runs are invaluable. Blue-green deployments involve running both the original mainframe application and the new Java application simultaneously, routing a small portion of traffic to the new system initially. Parallel runs involve operating both systems in parallel and comparing outputs to ensure functional equivalence. These methods significantly minimize disruption during the transition period, ensuring near-zero downtime for critical business operations.[2]

Human-in-the-Loop Integration for Oversight and Refinement

The integration of human expertise throughout the migration process is not merely a fallback but a fundamental component of the architecture. A dedicated Human Interface component allows users to monitor the agentic workflow, observe intermediate outputs on the Blackboard, and intervene when necessary.[19] This enables human experts to provide invaluable feedback to the LLMs, correct "hallucinations" or inaccuracies, and validate agent outputs, particularly for highly complex or ambiguous legacy code segments. This hybrid approach, combining the efficiency and scalability of AI with the nuanced judgment and domain knowledge of human experts, is crucial for ensuring accuracy and reliability in high-stakes enterprise migrations.[29]

The criticality of trust and explainability in AI-driven migration cannot be overstated. For technical leaders to confidently entrust an AI system with the transformation of mission-critical mainframe applications, the system cannot operate as an opaque black box. The "human-in-the-loop" mechanism extends beyond simple error correction; it is fundamental for building confidence in the system's outputs and ensuring that the AI's transformations align precisely with business intent and regulatory requirements. The capability of LLMs to "generate human-like comments and explanations for the code" [25] and the demand for "detailed reasoning" from LLMs when acting as judges [29] underscore the profound need for explainability. The architecture must therefore prioritize not only automation but also transparency and interpretability. The Blackboard, by making all intermediate outputs and agent contributions visible, and the LLMs, by providing clear justifications for their transformation decisions, contribute significantly to this objective. This fosters trust, enables effective human oversight, and facilitates crucial knowledge transfer, which is particularly vital given the existing skill gaps in mainframe expertise. The system needs to be designed to be auditable, providing clear justifications for its transformation decisions at every step.

8. Conclusion and Future Outlook

The modernization of mainframe applications to Java represents a profound strategic shift for enterprises aiming to thrive in the digital economy. The inherent complexities of legacy systems, coupled with a growing scarcity of specialized mainframe skills, necessitate innovative approaches. This report has detailed a robust multi-agent AI system architecture that offers a transformative pathway for mainframe to Java conversion.

The proposed architecture harnesses the power of AI agent specialization, scalability, and fault tolerance, orchestrated predominantly through a dynamic Blackboard pattern. This design enables a collaborative and iterative approach to tackling deeply intertwined mainframe application logic and infrastructure. Core AI agents, each equipped with advanced AI/ML techniques—including sophisticated Large Language Models for semantic understanding and code generation, rigorous static and dynamic analysis for quality assurance, and precise rule-based systems for enforcing standards and handling mainframe-specific patterns—collectively manage the entire migration lifecycle.

This comprehensive system directly addresses the most formidable challenges of mainframe modernization: mitigating legacy code complexity and interdependencies, ensuring data integrity and schema compatibility, enhancing security and compliance, and streamlining the critical testing and quality assurance phases. By automating complex transformations and encapsulating specialized mainframe knowledge, the multi-agent AI system effectively bridges critical skill gaps and minimizes the risks of downtime, thereby safeguarding business continuity. This positions the system not merely as a technical solution but as a strategic tool for organizational resilience and competitive advantage.

Looking ahead, the evolution of this multi-agent AI system holds significant promise. Future directions include the continuous learning and adaptation of agents based on new migration projects and human feedback, leading to increasingly refined transformation capabilities. Deeper integration with advanced cloud-native services and platforms will further enhance scalability and operational efficiency. Furthermore, the foundational principles and architectural patterns developed for mainframe to

Java conversion can be extended to address other complex legacy modernization challenges across diverse technology stacks. Ultimately, this architecture provides a clear and actionable roadmap for enterprises to unlock agility, significantly reduce technical debt, and ensure the longevity and adaptability of their critical business functions in the ever-evolving digital era.

Works cited

1. Mainframe Components, accessed on August 12, 2025, https://www.mainframestechhelp.com/what-is/mainframe/components.htm
2. Mainframe Migration Challenges | How to Overcome Them - Royal Cyber, accessed on August 12, 2025, https://www.royalcyber.com/blogs/mainframe/top-5-challenges-in-mainframe-migration/
3. Accelerate Your z/OS Career by Learning the Basic Components of Mainframe, accessed on August 12, 2025, https://blog.share.org/Article/accelerate-your-zos-career-by-learning-the-basic-components-of-mainframe
4. What Is Mainframe Modernization and Why Does It Matter to Your Business? - OpenLegacy, accessed on August 12, 2025, https://www.openlegacy.com/blog/mainframe-application-modernization
5. Mainframe modernization: Challenges, strategies and benefits - CGI.com, accessed on August 12, 2025, https://www.cgi.com/en/article/alliances/mainframe-modernization-challenges-strategies-benefits
6. Mastering the COBOL to Java Migration: Top 10 Challenges and How to Overcome Them - Prepare for the Future With Legacy Application Modernization, accessed on August 12, 2025, https://cmfirstgroup.com/mastering-the-cobol-to-java-migration-top-10-challenges-and-how-to-overcome-them/
7. Developer: My Application Uses IMS, accessed on August 12, 2025, https://www.microfocus.com/documentation/enterprise-developer/ed60/ED-Eclipse/GUID-24A7A11E-F676-474B-A8A1-556625282D92.html
8. VSAM - COBOL - Angelfire, accessed on August 12, 2025, https://www.angelfire.com/folk/anoop/Vsamcob.pdf
9. Specifying access modes for VSAM files - IBM, accessed on August 12, 2025, https://www.ibm.com/docs/en/cobol-zos/6.3.0?topic=records-specifying-access-modes-vsam-files
10. Access Manager for CICS - Oracle, accessed on August 12, 2025, https://www.oracle.com/docs/tech/database/am4cics.pdf
11. Working with JCL in Mainframe: Practical Guide & Tutorial - Swimm, accessed on August 12, 2025, https://swimm.io/learn/mainframe-modernization/working-with-jcl-in-mainframe-practical-guide-and-tutorial

12. IMS 15.4 - Application programming - Database hierarchy examples - IBM, accessed on August 12, 2025, https://www.ibm.com/docs/en/ims/15.4.0?topic=database-hierarchy-examples
13. IMS DB Structure - Tutorialspoint, accessed on August 12, 2025, https://www.tutorialspoint.com/ims_db/ims_db_structure.htm
14. CICS conversational - IBM, accessed on August 12, 2025, https://www.ibm.com/docs/en/zos-basic-skills?topic=cics-conversational-pseudo-conversational-programming
15. Pseudoconversational transactions - IBM, accessed on August 12, 2025, https://www.ibm.com/docs/en/cics-ts/6.x?topic=transactions-pseudoconversational
16. Job Control Language - Wikipedia, accessed on August 12, 2025, https://en.wikipedia.org/wiki/Job_Control_Language
17. Multi-Agent Design Pattern - Microsoft Open Source, accessed on August 12, 2025, https://microsoft.github.io/ai-agents-for-beginners/08-multi-agent/
18. AI Agent Orchestration Patterns - Azure Architecture Center - Microsoft Learn, accessed on August 12, 2025, https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns
19. What is Multi-Agent Collaboration? | IBM, accessed on August 12, 2025, https://www.ibm.com/think/topics/multi-agent-collaboration
20. What is Communication in Multi-Agent Systems? | Activeloop Glossary, accessed on August 12, 2025, https://www.activeloop.ai/resources/glossary/communication-in-multi-agent-systems/
21. ryanstwrt/multi_agent_blackboard_system: This repo is for the implementation of a Multi Agent Blackboard System (MABS) - GitHub, accessed on August 12, 2025, https://github.com/ryanstwrt/multi_agent_blackboard_system
22. [Literature Review] Exploring Advanced LLM Multi-Agent Systems Based on Blackboard Architecture - Moonlight, accessed on August 12, 2025, https://www.themoonlight.io/en/review/exploring-advanced-llm-multi-agent-systems-based-on-blackboard-architecture
23. en.wikipedia.org, accessed on August 12, 2025, https://en.wikipedia.org/wiki/Contract_Net_Protocol#:~:text=The%20Contract%20Net%20Protocol%20(CNP,close%20to%20sealed%20auctions%20protocols.
24. AI Code Review - IBM, accessed on August 12, 2025, https://www.ibm.com/think/insights/ai-code-review
25. AI Code Review: How It Works and 5 Tools You Should Know - Swimm, accessed on August 12, 2025, https://swimm.io/learn/ai-tools-for-developers/ai-code-review-how-it-works-and-3-tools-you-should-know
26. 20 Best AI Coding Assistant Tools in 2025 - Qodo, accessed on August 12, 2025, https://www.qodo.ai/blog/best-ai-coding-assistant-tools/
27. The Impact of AI and Machine Learning on COBOL Development - Revolutionizing Legacy Systems - MoldStud, accessed on August 12, 2025,

https://moldstud.com/articles/p-the-impact-of-ai-and-machine-learning-on-cobol-development-revolutionizing-legacy-systems
28. Amazon Q Developer - Generative AI, accessed on August 12, 2025, https://aws.amazon.com/q/developer/
29. Quality Evaluation of COBOL to Java Code Transformation - arXiv, accessed on August 12, 2025, https://arxiv.org/html/2507.23356v1
30. Quality Evaluation of COBOL to Java Code Transformation - ResearchGate, accessed on August 12, 2025, https://www.researchgate.net/publication/394175354_Quality_Evaluation_of_COBOL_to_Java_Code_Transformation