International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

Comparative Review of Java Spring Boot vs. Microservices for Financial Data Ingestion Pipelines

Shreyansh Sharma

Abstract
Within the changing environment of financial technology, the choice of correct architectural paradigm to data ingestion pipelines has a critical effect on system performance, scalability and maintainability. The paper is a thorough comparative analysis of Java Spring Boot framework and microservices architecture to use in designing financial data ingestion pipelines. The paper measures such important metrics as latency, throughput, scalability, fault tolerance and financial services specific compliance requirements. Results have shown that Spring Boot is quick to develop and deploy when the application being developed is moderate in size with a load of up to 300,000 requests per second, whereas microservices architecture is better scaled and resilient when financial data processing is large volume with more than 2 million events per second. The paper will be concluded with evidence-based suggestions to choose right architecture depending on the organizational needs, complexity of the system, constraints in the operations and the level of maturity of the financial institutions. Moreover, this study also determines hybrid solutions that can balance the agility of microservices and the simplicity of Spring Boot operations.

Keywords: Spring boot, Microservices architecture, financial data ingestion, Data pipeline, Real-Time Processing, Apache Kafka, Cloud-native application, Distributed system, Financial Services Technology.

I. INTRODUCTION

The financial services sector has experienced a great digital transformation, which has been accelerated by the growth of data volume, the regulatory need and expectations of a customer to get real-time insights [1]. Millions of transactions are being processed in financial organizations every day, and it needs robust, scalable and secure data ingestion pipelines that can handle high-velocity data streams and ensure a high level of compliance with regulatory frameworks, including GDPR, PCI-DSS and SOX [2]. The amount of financial data being generated in the world has grown exponentially, where the global datasphere goes up to 175 zettabytes by 2025 and financial institutions alone make more than 100 trillion transactions every year [3].

The monolithic architectures of traditional systems are increasingly facing problems with satisfying the dynamic scalability and fault isolation needs of the historical financial systems. Single deployable applications are becoming more challenging to maintain, test, and scale as the business needs change and the transaction volumes grow. At the same time, the Spring Boot platform has become a leading Java-based application development platform, which includes a rich ecosystem support, embedded servers, and production-ready characteristics [4].

Figure 1: The Role of Microservices in Payment Systems
Source: [5]

The decision on the adoption of financial data ingestion pipelines based on Spring Boot with a monolith or microservices system is a core strategic decision with extensive consequences on systems performance, operational complexity, overall cost of ownership, and organizational agility [5]. Figure 1 also describes the roles in payment systems. This choice is particularly paramount since financial institutions have been encountered with the greatest competitive pressure as a result of fintech startups, open banking policy, and changing consumer demands that financial services can be offered in real-time to users.

A. Research Objectives

- Compare and contrast the architectural features, design patterns and component design of Spring boot and microservices, regarding financial data ingestion.
- Evaluate the fault tolerance, resilience capacity, and characteristics of disaster recovery.
- Review overall security and compliance concerns such as regulatory requirements of financial applications.
- Make evidence-based suggestions on the choice of architecture based on the needs and requirements of the organization.
- Find the hybrid solutions and migration planning of organizations moving between architectures

B. Significance

Financial data pipelines financial data pipelines are basic infrastructure to important business processes such as real-time analytics, fraud management, algorithmic trading, regulatory reporting, and risk management. The architectural choices available in the pipeline design have a direct effect on how financial institutions are able to process real-time market data, identify fraudulent transactions in milliseconds, ensure data integrity across distributed systems, have audit trails to meet regulatory compliance, and address strict regulatory obligations set upon financial institutions by regulatory authorities around the world.

Moreover, the financial cost/benefit of architectural choices is high. In a 2024 survey by Gartner, inadequate architectural decision-making in data pipelines causes about 70% of damages that the organizations will struggle with [6]. This study can equip financial technology professionals with factual evidence and real-world advice towards informed architectural choices that nurture business needs and financial limits and capabilities.

C. Paper Organization

The rest of the paper will be organized in the following way, Section II is the literature review about related subjects: Spring Boot, microservices architecture, financial data pipelines, and Apache Kafka. Section III explains the employing of the extensive methodology that will be utilized in comparative analysis. Section IV gives architectural comparison and technical features of the two approaches in detail.

II. LITERATURE REVIEW

A. Spring Boot Framework and Ecosystem

Introduced by Pivotal (now VMware) in 2013, Spring Boot is an opinionated framework that is based on the Spring ecosystem that is intended to make the creation of production-ready applications easier [7]. The framework offers auto-configuration features which heavily shorten boilerplate code, embedded server support (Tomcat, Jetty, Undertow) and complete dependency management with starter dependencies [8]. Spring boot has been extensively used and it is estimated that more than 70% of Java developers are currently using Spring boot in their practices.

Recent studies indicate that Spring boot can be used to develop an application quickly. Research indicates that Spring Boot applications are capable of delivering response times of between 16-120ms to perform usual tasks, and throughput of between 250,000 and 300,000 requests per second in a basic REST API [9]. Introduction of the framework with Spring Cloud gives it distributed system features such as Eureka (service discovery), Config Server (centralized configuration), Ribbon (client-side load balancing) and Hystrix (circuit breaker patterns), suitable in both monolithic and microservices implementations [10, 11]. The testing ecosystem built around Spring Boot based on JUnit 5, Mockito, Test Containers and Spring Test utilities allows the use of detailed testing strategies that are much needed in financial applications with high stakes on ensuring a correct application [13]. The framework accommodates multiple testing annotations such as the controller layer testing, data layer testing, and full application context testing, namely, the annotations of @WebMvcTest, @DataJpaTest, and @SpringBootTest respectively and enables the test-driven developmental practices [12]. The Spring Boot ecosystem has now become quite mature and there are now specially focused libraries to process financial data, such as Spring Cloud Stream, event streaming, Spring Cloud Config, distributed configuration management and Spring Cloud Circuit Breaker, resilience patterns.

B. Principles of Architecture of Microservices

Microservices architecture was a reaction to the shortcomings of monolithic systems, who both suggested splitting the application into loosely coupled services which can be independently deployed.
According to research by Gartner 74% of organizations use microservices architecture today, and 23 percent expect to use it in the next 12 months, which shows that it has become widely used in the industry [14]. Examples of financial institutions switching to microservices have been JPMorgan Chase, Monzo Bank, Citibank and Stripe, where organizations have found great benefits in their agility, scalability and innovation capabilities due to the transition to microservices.

Other noteworthy benefits of microservices are service independent scalability, service choices in terms of the ability to select the most appropriate tools for each specific service, fault isolation to prevent cascading failures, and deployment speed facilitating multiple releases in a single day. Nevertheless, architecture brings complexity in areas like inter-service communication, distributed transaction

management, issues of data consistency and operational monitoring needs [15]. Research demonstrates that to support microservices, more complex orchestration tools are needed such as Kubernetes and dedicated monitoring tools to efficiently operate hundreds of distributed services.

C. Financial Data Ingestion patterns and requirements

Financial data ingestion pipelines are required to process a variety of data sources such as core banking platforms, market data feeds, payment platforms, foreign exchange platforms, derivatives exchanges, and customer relationship management systems but ensure message quality, completeness, timeliness and integrity. The financial market has come up with strict standards of ingesting data such as:

Data Quality Requirements Completeness: There cannot be any values that are missing, accuracy, checks against source system, consistency, checks across related data sets, and timeliness: checks: data must be received within specified windows [16].

Throughput Requirements: Payment processing during peak periods can support 150,000+ transactions per second, derivatives markets can support millions of quotes every day, and FX trading can support price streaming in the form of continuous price updates.

Latency Requirements: Fraud detection must include processing time of 200-500ms, algorithmic trading must be served using sub-milliseconds of latency, and regulatory reporting may be provided on a batch basis with a schedule of once per day [17].

The industry has three major ingestion patterns:

The Batch Ingestion loads data periodically which is appropriate when loading historical data, daily reporting and compliance reporting to regulatory bodies. Although simpler to run and control, batch processing creates latency that cannot be used in real-time financial analytics and fraud detection.

Real-time Streaming takes incoming data in real-time and processes it in real-time, which is necessary to detect fraud, perform algorithmic trading, create operational dashboards, and customer-facing applications. Apache Kafka and other technologies can be used to provide high throughput and low latency streaming of data with throughput capacity of more than 1 million events per second and latency as low as sub-10ms.

Hybrid Lambda Architecture is a hybridization between batch and streaming layers, which creates not only historical truth but also real-time knowledge [18]. The architecture is especially useful when the financial institutions need both real-time risk analysis and detailed historical reporting to comply and carry out analytics [19].

D. Apache Kafka in the Financial Industry

Apache Kafka has evolved into the standard in real time data streaming in financial services, used by large financial exchanges, including NASDAQ and NYSE, to provide market data feeds to traders and investors. Kafka distributed architecture offers the following key features: high throughput (800,000 or more events/second/per broker), low-latency (4-6ms end-to-end), fault tolerance with replication and data durability with disk persistence [20, 19].

Financial institutions use Kafka in a number of mission-critical applications such as payment processing pipelines, fraud detection streams, regulation reporting aggregation and monitoring of cross-border transactions. The exact-once semantics of the platform guarantees financial accuracy, the durability of the data through replication ensures that no transaction is lost, and the scalability of the stream processing makes it the right choice in maintaining the financial source of truth in systems needed to have a six-nines [18].

E. Financial systems Security and Compliance Requirements

Financial data pipelines should be in line with strict regulations such as GDPR (data protection), PCI-DSS (payment card data), SOX (financial reporting accuracy), Basel III (capital requirements) and MiFID II (market integrity). Some of the most important ones are end-to-end encryption, role-based access controls,

complete audit trails, data lineage tracking, real-time compliance monitoring and segregation of duties [21].

It is proven that automated compliance validation built into continuous integration / continuous deployment (CI/CD) pipelines saves deployment times, as well as improves security posture. The Gramm-Leach-Bliley Act (GLBA) also expressly provides those financial institutions protect the customer information by having encryption, multi-factor authentication, and maintaining constant security checks.

III. METHODOLOGY

A. Research Design

This study is a mixed-methods study in which a literature review, technical review, and comparison benchmarking will be used to evaluate Spring Boot and microservices architecture as a financial data ingestion pipeline. It uses qualitative analysis of architectural features as well as quantitative assessment of performance measures as methodology. The research design is based on current principles of systematic literature review and comparison in software engineering.

B. Evaluation Framework

The comparative analysis model is based on eight broad dimensions:

Architectural Characteristics: The pattern of design, structure of components, protocols of communication, deployment models and boundaries of the system.

Development Experience: Complexity of setting up, learning curve, server testability, speed of development and productivity of the team.

Operational: Structure: Deployment, monitoring and observability, fault tolerance, maintenance overhead as well as operational load.

Cost Considerations: Infrastructure cost, operating cost, tooling cost, and overall cost of ownership estimates.

Team and Organization facts: Team structure requirements, DevOps maturity requirements, organizational culture alignment and skill requirements.

C. Data Collection Methods

Technical Benchmarking: Test of published performance benchmarks of reputable sources, such as Spring Framework team, Apache Foundation, and vendor benchmarks of cloud providers.

Case Study Analysis: The review of publicly released case studies and technical presentations of financial institutions such as Citibank, Monzo Bank, JPMorgan Chase, and fintech firms.

Pattern Analysis Architecture: Evaluation of design patterns, design best practices, design anti-patterns recorded in technical literature and open-source applications.

D. Comparative Analysis Approach

It has been demonstrated that information can be acquired through a comparative approach of analysis.

The comparative analysis will utilize a systematic way of evaluation:

Component-Level Analysis: A study of individual architecture and structures, technologies, and their respective strengths and weaknesses.

IV. ARCHITECTURAL COMPARISON

A. Architecture of Spring Boot in Financial Pipelines

The Spring boot applications are normally based on the layered architecture that includes presentation layer (REST controllers), service layer (business logic), repository layer (data access), and domain layer (business entities and rules) [22].

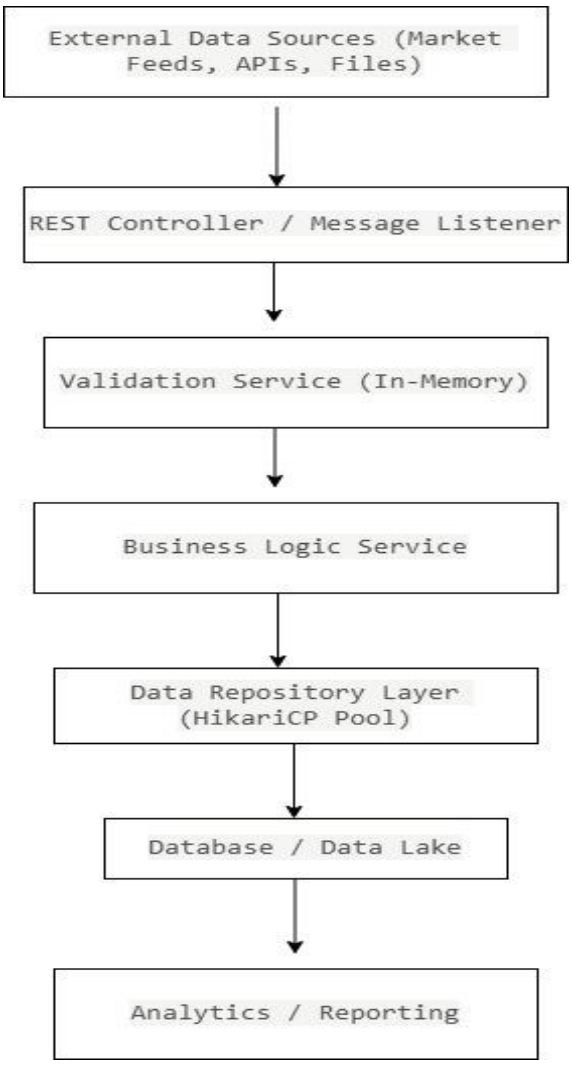

Figure 2: Version A - Spring Boot Pipeline

The Important Architectural Elements:

Embedded Server: Spring boot has an embedded Tomcat, Jetty or Undertow server, which allows applications to run as standalone JARs, without any outside server requirements. This goes a long way in simplifying deployment as well as providing consistency in development, testing and production environments. The embedded server design makes it a less complex operation as the individual server administration of applications is removed [22].

Dependency injection and inversion of control: Spring container is able to handle bean lifecycle and dependencies by use of either constructor, setter or field injection and encourages loose coupling and testability [23]. This design pattern allows flexible component replacement to facilitate testing as well as with dynamic reconfiguration without modifying the code.

Auto-Configuration: The annotation Spring Boot is auto-configuring application components, determined by application component dependency discovery, removing tedious XML configuration. This configuration-over-convention paradigm has a huge effect on the shortening of setup time and configuration mistakes.

Spring Cloud Integration: Spring boot can be used with Spring Cloud applications such as Eureka (service discovery), Config Server (centralized configuration), Ribbon (load balancing) and Hystrix (circuit breaker) with distributed systems [24]. These features allow Spring boot to be used in supporting microservice patterns as required.

Microservices: Several autonomous Spring Boot applications are communicating through the REST or messaging, integrating the advantages of Spring Boot development and microservices architecture.

B. Microservices Architecture of Financial Systems

The microservices architecture breaks down applications into deployable independently maintained services doing business capabilities. In ingestion of financial data, commonly used microservices are:

- Data Ingestion Service accepting data as external sources such as market feeds, APIs, file uploads and queue systems [25].
- Validation Service is conducting data quality check, schema check and business rule check.
- Service normalizing, enriching and converting data into target forms Transformation Service converting data into forms usable by the destination.
- Storage Service continues data to data bases, data lakes, data warehouses or data queues.
- Notification Service broadcasting events and warnings according to the results of data processing.

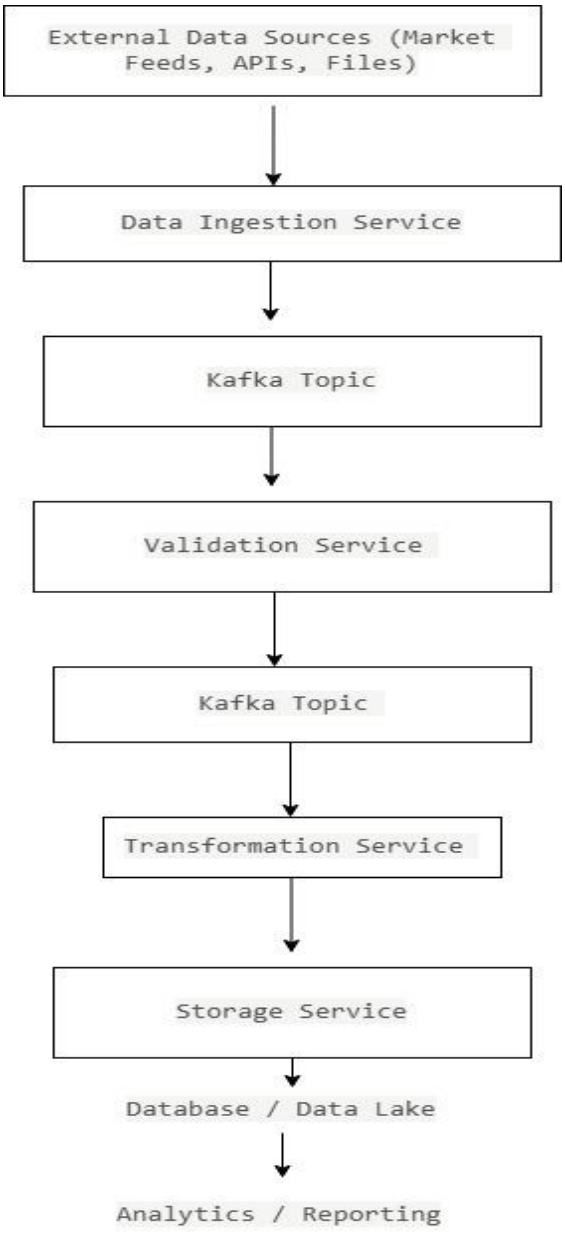

Figure 3: Microservices Event-Driven Processing

Asynchronous Communication:
Point to point messaging with message queues (RabbitMQ, AWS SQS, Azure Service Bus).
Pub/sub patterns Event streaming (Apache Kafka, AWS Kinesis, Pulsar).

C. Data Pipeline Arch Pattern Comparison

Spring boot data pipeline architecture:
External Data Source → REST Controller/Message Listener → Validation Service → Business Logic Service → Data Repository → Database/Data Lake
Microservices Data Pipeline Architecture:
External Data Source → Ingestion Service → Kafka Topic → Validation Service → Kafka Topic → Transformation Service → Kafka Topic → Storage Service → Database/Data Lake

V. PERFORMANCE ANALYSIS

Empirical tests have shown that Spring boot applications attain:

- Database CRUD Operations: Simple create 5-8ms median response time, simple read 8-10ms median response time, simple update 5-8ms median response time, simple delete 8-10ms median response time.
- Business Logic Operations: 16-120ms average business logic operations such as validations and calculations.
- Synchronous Request Processing: End-to-end latency of between 20-50ms in basic validation and storage operations [26].
- Asynchronous Batch Processing: 100-500ms response time of batch operations on thousands of records.
- Micro services architecture offers inter service communication which provides network latency:
- REST API Calls: The network round trip will add between 10-30ms of additional latency per service hop.
- Kafka Message Processing: The latency of asynchronous message delivery and processing is 4-6ms.
- Real-time Fraud Detection: Latencies under 500ms to detect a transaction after initiating it.
- Market Data Distribution: Less than 10ms between Exchange and Customer Systems.

B. Capabilities Comparison of throughput

- Basic REST APIs: 250,000-300,000 requests/s basic read requests
- Database-Intensive Operations: 10,000-50,000 records per second based on the complexity of data.
- Caching: Cuts in half or three quarters backend load on popular data.
- JPMorgan Chase: Handles 100 million or more transactions each day with JPMorgan Chase with microservices based on Kafka.
- NASDAQ: 10+ million quotes in the peak market in one second.
- Stripe: Handles 200,000+ payment requests in a second around the world infrastructure.

VI. SECURITY AND COMPLIANCE

A. Data Protection Mechanisms

Specification of Encryption at Multiple Levels:
The financial data pipelines are to be encrypted:

- Database, file system, and backup AES-256 encrypted data, keys maintained by HSMs or cloud KMS [27].
- Information on transit: TLS 1.3 on any network transactions between services and external systems.
- Important Management Hardware Security Modules (HSMs) or cloud KMS to store encryption keys and rotate them.

These encryption requirements are supported by both Spring Boot and microservices architecture with:

- Spring Security authentication, authorization and encryption facilities.
- Smart Mesh Service mesh (Istio) of automatic mutual TLS mTLS) between microservices
- Encryption services on clouds (AWS KMS, Azure Key Vault, Google Cloud KMS)
- Encryption of databases at the database level with transparent data encryption (TDE)

Implementation of Access Control:

The control of access is provided by role-based access control (RBAC) and attribute-based access control (ABAC) techniques that ensure that only authorized users and services can access sensitive financial data:

- OpenID connects User authentication and authorization: OAuth 2.0.
- JWTs of service-to-service authentication and stateless verification.
- Centralized authentication, authorization and rate limiting API Gateway.
- Database-level fine-grained permissions that limit access to particular data by the user.

B. Audit and Compliance Mechanisms

Extensive Audit trail Requirements:

- Data Lineage: Tracking of data origin and transformations between sources and destinations.
- Transformation Operations: The processing operations of financial data.
- Patterns of access: Timestamped user/service access to sensitive data.
- Configuration Changes: System changes such as deployment and infrastructure changes.
- Security Incidents: Identification and investigation of abnormalities.

Audit Implementation with Spring boot:

The auditing in Spring boot applications is done by:

- Entity-level auditing with Spring Data Audit annotations (CreatedBy, LastModifiedBy).
- HTTP requests interceptors of operations and identity in custom audits.
- Security monitoring by having centralized logging to SIEM systems (ELK stack).
- Triggers in the databases to track the data modification and change validation.

Microservices Audit Implementation:

Microservices' leverage:

- Complete tracing of request flows across services (Distributed tracing Jaeger, Zipkin).
- Patterns of event sourcing with full state change history audit history.
- Aggregating logs aggregated by a single centralized aggregator (Fluent Bit, Logstash, ELK) of all services.
- Mesh observability Service mesh level network-level identity auditing of inter-service communication

C. Regulatory Compliance Requirements.

GDPR Compliance:

Data protection regulations stipulated by the Europeans demand:

- Right to ensure Data deletion capabilities on request by customer)
- Export capability of customer data (data portability)
- Consent persuasion and monitoring.
- Minimization and limitation of purpose of data.
- Privacy by design principles on system architecture.

Microservices help to comply with GDPR by:
- The limits of services were set in accordance with data domains that would allow data deletion.
- Service-based deletion propagation.
- Privacy microservices Consent management.

PCI-DSS Compliance:
- The standards of the payment card industry require:
- Firewall protection and network segmentation.
- Data transmission encryption by cardholders.
- Strong authentication and access control.

Underlying vulnerability scanning and regular security testing.
SOX Compliance:
- Financial reporting provisions of Sarbanes-Oxley Act requirements are:
- Separation of duties to avoid illegal transactions.
- All changes that are made to the system are controlled under change management.
- General controls of IT system reliability (ITGC).
- Compliance Audit trail records

VII. EXAMPLES AND PRACTICAL APPLICATIONS

A. Case Study 1: Citibank Mobile Transformation Strategy
In 2022, Citibank updated their mobile banking platform to an event-based microservices architecture with Spring boot that provides 212 services on mobile devices [28].
Key Results:
- Effectively completed 212 mobile services on time and on a tight budget.
- An attained high rate of user satisfaction and over 2 million downloaders.
- Agile wave-based 12-months to 5-month development cycle time decreases.
- Better fault isolation: single service failures did not affect the whole platform.

B. Case Study 2: Financial Data Feed Publishing Infrastructure
The use of Amazon Managed streaming on Apache Kafka (MSK) and Apache Flink was by a large financial data distributor to support real-time market data distribution [29].
Key Results:
- Kafka cluster deployment across North America, Europe, and Asia in multi-regions to distribute the data globally.
- SASL/SCRAM access control over customers.
- AWS PrivateLink to connect to private networks of customers to VPCs.
- The historical data queries and analysis using OpenSearch.

C. Monzo Bank Microservices Platform Case Study
To create its core banking platform, Monzo Bank, a UK digital bank, was designed with greenfield microservices architecture since its inception.
Platform Characteristics:
More than 1,500 microservices with all banking activities such as transactions, accounts, compliance, and more.
Key Advantages:
- Quick Innovation: Implement changes in applications in hours, as opposed to weeks, responding more quickly to market opportunities.

- Independent Team Ownership: Small teams (6-8 people) are given individual end-to-end ownership of services.
- Granular Scaling: Scale the transaction processing services on their own basis as opposed to account management depending on particular load patterns.
- Fault Isolation: Failure of services to a particular service; system does not shut down.
- Technology Flexibility: Teams choose the best technology stacks to use in certain cases.

VIII. FINDINGS AND DISCUSSION

The comparative study of Java Spring Boot and microservices as application architecture solutions to financial data ingestion pipelines have shown that architectural choices cannot be universalized; they are very dependent on organizational background, system size, regulatory framework, and DevOps maturity [30]. The indicators demonstrate that the two methods have the technical abilities to carry modern financial burdens, yet with various trade-offs regarding complexity, cost, scalability and time-to-market. Instead of comparing the decision between Spring Boot and microservices, the results indicate that a better approach should be to consider what is the combination of Spring Boot and microservices patterns that best fits the needs of the institution both today and in the future.

The first significant observation is about the connection between the size of a system and system complexity. The second major conclusion is that microservices architectures have apparent benefits at larger scale and complexity, especially in large financial institutions and data providers that process millions of transactions or events per day. In such a situation, independent scaling of discrete pipeline components, i.e., validation, transformation, or enrichment services, becomes crucial. This is demonstrated by the empirical experience of real-world case studies that microservices architectures can sustain throughput of many millions of events per second with reasonable latency in combination with event-streaming technologies like Apache Kafka and event-processors like Apache Flink. Decoupling the ingestion, validation, transformation, and storage phases of an application enables institutions to distribute computing resources selectively and appropriately where they are of the most benefit and not scale a large monolithic application to the load of a single hot path [31]. The high fine-grained scalability is, however, fully achievable under the condition that a set of organizations possess well-established operational policies, such as automated provisioning, high observability, and a rigorous configuration management.
Notable trade-offs in the resilience and fault isolation are also mentioned during the analysis. This is specifically useful in money-sensitive financial institutions (such as 99.99% uptime requirements) and intricate business sectors, in which localized failures are unavoidable. Patterns such as circuit-breaker, bulkheads, timeouts with a particular emphasis on their implementation with the help of Spring Cloud and service mesh technologies enable services to work gracefully instead of triggering systemic failures [31]. In comparison, Spring Boot monoliths or modular monoliths have fewer moving parts, their failure surfaces are fewer, and root cause analysis is simpler, which comes at the cost of a loss in flexibility and granularity.

The results of this study with regards to security and compliance indicates that both architectures have the potential to meet the high regulatory requirements of financial services, though at varying levels of emphasis and risk. Monolithic applications made in Spring Boot pull the issues of authentication, authorization, logging and auditing together into one deployment which can make compliance audits simpler and help eliminate the possibility of security policies being enforced inconsistently across modules [32]. This centralization may be beneficial to smaller organizations or systems that have well defined security domains. However, in microservices environments, it becomes more difficult to ensure the security policies on a regular basis, as every service is likely to have its own endpoints, data access logic, and configuration. One more significant theme that appeared as a result of the analysis is that cost considerations are multi-dimensional and time-dependent but not infrastructural. At a face value, the cost of the infrastructure of a microservices-based ingestion pipeline is usually more expensive than a Spring

Boot monolith because of overhead costs related to the operation of multiple processes, containers, monitoring tools, and supporting infrastructure. The difference that is calculated can be between two to four times the baseline of an equivalent monolithic solution, particularly at low stages when the use is not at its best. Infrastructure cost is however, not the only part of the total cost of ownership. On the other hand, in smaller systems or organizations with relatively low change velocity, the operational overhead of microservices will not often be recouped, and a well-designed Spring Boot monolith will often be economically more efficient throughout the useful life of the system [32].

Organizational and team factors are also highlighted in the findings as key determinants of architectural success. Microservice systems are especially suitable to organizations that have autonomous, domain-based teams where ownership lines are well defined, and teams have the freedom to deploy, manage and run their own services [22]. The model can be seen in digital-native banks and fintechs that have embraced the operating models of two-pizza teams and products. Microservices can conflict with existing structures in more traditional institutions with centralized IT departments, siloed functions, and heavy governance, and pressure them into coordination bottlenecks and erode much of the benefits promised.

IX. RECOMMENDATIONS AND DECISION FRAMEWORK

A. Assessment Framework

The following dimensions should be considered by an organization before choosing architecture:

Current State Assessment:

- Current complexity and technical debt of the system.
- Expertise, team size and organizational structure.
- Existing infrastructure and equipment.
- Financial limitations and returns on investment.

Requirements Analysis:

- Anticipated volumes and growth pattern of the transactions.
- Time-to-market pressures
- Organizational Readiness:
- DevOps maturity level
- Culture of decision making and autonomy of teams.
- Tooling and training capability of investments

B. Decision matrix

Criterion	Favor Spring Boot	Favor Microservices
Transaction Volume	< 100K/day	> 1M/day
Team Size	< 10 developers	> 20 developers
Service Boundaries	Tightly coupled	Clearly separable
Scaling Pattern	Uniform	Variable by service
DevOps Maturity	Basic	Advanced

X. CONCLUSION

The detailed comparative analysis of Java Spring Boot and microservices architecture as the pipeline of financial data ingestion proves that the choice of an architecture is much more sophisticated than a dichotomic decision. Instead, it is a strategic choice which has to consider organizational maturity, system size, regulatory context, team organization and business long-term goals. This study has discussed the technical capabilities, performance characteristics, operational requirements, security implications and practical applications of the two approaches that give evidence-based advice to the financial institutions and fintech companies to go through this key-architecture decision.

The results indicate that Java Spring Boot still offers a great base to build financial data pipelines in cases when quick development, ease of operations, and a medium level of transaction traffic are the main factors to consider. The comprehensive ecosystem of the framework, developed tooling, integrated server supporting capabilities and production-ready capabilities allow organizations to produce high-quality applications fast with little operational overhead. An implementation with close business logic coupled institutions handling less than 100,000 transactions a day and less developed DevOps would find a practical working solution in a well-designed Spring Boot monolith or modular monolith to satisfy business requirements and keep the code quality and maintainability. The empirical data indicates that maximally scaled Spring Boot applications are able to support throughput of 250,000 to 300,000 requests per second with simple workloads, with response times around tens of milliseconds, which is quite satisfactory with respect to a broad range of financial workloads involving batch ingestion, near-real-time reporting, and most customer-facing services.

On the other hand, microservices architectures exhibit obvious benefits and need to be considered when it comes to the large-scale financial institutions that deal with millions of transactions each day and have high availability and scalability criteria. The case studies reviewed within the context of the present research such as implementations in Citibank, Monzo Bank, and leading financial data providers demonstrate that microservices allow achieving the degree of granular scalability, fault isolation, and organizational agility needed to compete in the modern financial markets. One critical fact that comes from this study is that hybrid solutions that integrate both Spring boot and micro services concepts are the most sensible and evolutionary direction that many financial institutions should take. Instead of going to a purely monolithic/purely microservices architecture, institutions can strategically implement modular monoliths on core transactional systems where consistency and predictability of operations matter and enclose them with microservices to external capabilities like notifications, analytics, regulatory reporting and customer-facing channels.

REFERENCES:

[1] O. E. Aro, “Data analytics as a driver of digital transformation in financial institutions,” *World Journal of Advanced Research and Reviews*, vol. 24, no. 1, pp. 1054–1072, Oct. 2024, doi: https://doi.org/10.30574/wjarr.2024.24.1.3124.

[2] Sudhir Vishnubhatla, “Deep Learning Pipelines for Financial Compliance: Scalable Document Intelligence in Regulated Environments,” *Zenodo (CERN European Organization for Nuclear Research)*, Aug. 2020, doi: https://doi.org/10.5281/zenodo.17638989.

[3] T. Coughlin, “175 Zettabytes By 2025,” *Forbes*, Nov. 27, 2018. https://www.forbes.com/sites/tomcoughlin/2018/11/27/175-zettabytes-by-2025/

[4] S. Aggarwal and Pandit, “Spring Boot Application using Three Layered Architecture in Java,” *Juit.ac.in*, 2023, doi: http://ir.juit.ac.in:8080/jspui/jspui/handle/123456789/10202.

[5] P. Kumar Joshi, “From Concept to Production: Crafting Payment Solutions with Java and Spring Boot,” *International Journal of Science and Research (IJSR)*, vol. 10, no. 11, pp. 1590–1596, Nov. 2021, doi: https://doi.org/10.21275/sr211101102708.

[6] S. Gaur, "Data Platform Project Failure: Key Challenges and Effective Solutions," *Medium*, Feb. 15, 2024. https://medium.com/lumiq-tech/data-platform-project-failure-key-challenges-and-effective-solutions-69ae837290c8 (accessed Dec. 23, 2025).
[7] N. Sarmin, "Spring Boot Framework: Features, Benefits, and Use Cases - StaticMania," *StaticMania*, 2024. https://staticmania.com/blog/spring-boot
[8] Spring, "Embedded Web Servers :: Spring Boot," *Spring.io*, 2025. https://docs.spring.io/spring-boot/how-to/webserver.html
[9] K. P. Path, "This Is Why Most Spring Boot Apps Fail Before 100K RPS (Mine Reached 1M)," *Medium*, Dec. 15, 2025. https://medium.com/javarevisited/this-is-why-most-spring-boot-apps-fail-before-100k-rps-mine-reached-1m-62e4de85403a (accessed Dec. 23, 2025).
[10] Spring, "Page Redirection," *Spring.io*, 2025. https://cloud.spring.io/spring-cloud-netflix/ (accessed Dec. 23, 2025).
[11] A. Obregon, "Distributed System Patterns with Spring Cloud," *Medium*, Aug. 03, 2023. https://medium.com/@AlexanderObregon/distributed-system-patterns-with-spring-cloud-4d4b6503cea5 (accessed Dec. 23, 2025).
[12] Z. Sayyed, "Development of a Simulator to Mimic VMware vCloud Director (VCD) API Calls for Cloud Orchestration Testing," *International Journal of Computational and Experimental Science and Engineering*, vol. 11, no. 3, Jul. 2025, doi: https://doi.org/10.22399/ijcesen.3480.
[13] S. Åkerblom and L. Huber, "Comparing Spring REST API test frameworks : A Comparison Study," *DIVA*, 2023. https://www.diva-portal.org/smash/record.jsf?pid=diva2:1764379 (accessed Dec. 23, 2025).
[14] gartner, "Microservices Architecture: Have Engineering Organizations Found Success? | Gartner Peer Community," *Gartner.com*, 2023. https://www.gartner.com/peer-community/oneminuteinsights/omi-microservices-architecture-have-engineering-organizations-found-success-u6b
[15] A. Raghu, "Optimizing Microservices Architecture for High-Speed Financial Data Processing: A Spring Boot–Based Approach," 2025, doi: https://doi.org/10.2139/ssrn.5238748.
[16] A. Houben, "Collateral scarcity and asset encumbrance: implications for the European financial system," *Financial Stability Review*, 2013. https://www.academia.edu/download/105959366/20-HOUBEN_Aerdt.pdf#page=101 (accessed Dec. 23, 2025).
[17] ESMA, "MiFIR Review Final Report," 2024. Available: https://www.esma.europa.eu/sites/default/files/2024-12/ESMA74-2134169708-7768_-_MiFIR_review_-_Final_Report_on_CTPs_and_DRSPs.pdf
[18] Z. Hasani, "Lambda Architecture for Real Time Big Data Analytic," *Academia.edu*, 2014. https://www.academia.edu/download/93244575/lambda-architecture-for-real-time-big-data-analytic.pdf (accessed Dec. 23, 2025).
[19] Silva, J. C.S, E. Pignaton, Lampoltshammer Thomas J, and Geyer Claudio F, "Strategies for Big Data Analytics through Lambda Architectures in Volatile Environments," *IFAC-PapersOnLine*, vol. 49, no. 30, Dec. 2016, doi: https://doi.org/10.1016/j.ifacol.2016.11.138.
[20] GeeksforGeeks, "Kafka Architecture," *GeeksforGeeks*, Jun. 24, 2024. https://www.geeksforgeeks.org/apache-kafka/kafka-architecture/
[21] Nihitha Sallapalli, "Balancing Data Security and Accessibility in Modern Data Engineering: A Framework for Enterprise Implementation," *Journal of Computer Science and Technology Studies*, vol. 7, no. 6, pp. 415–421, 2025, doi: https://doi.org/10.32996/jcsts.2025.7.6.49.
[22] S. Kumar and T. Prateek, "Spring Boot Application using Three Layered Architecture," 2023. Accessed: Mar. 13, 2024. [Online]. Available: http://www.ir.juit.ac.in:8080/jspui/bitstream/123456789/10201/1/Spring%20Boot%20Application%20using%20Three%20Layered%20Architecture%20by%20Sarthak%20Kumar.pdf

[23] Vamsi Krishna Myalapalli, "High Performance Spring Programming," *Academia.edu*, Sep. 10, 2014. https://www.academia.edu/download/34693133/10-IJCSE-00443.pdf (accessed Dec. 23, 2025).
[24] Ritika, "Enhancing Microservice Performance: A Hybrid Model Combining Service Discovery and Circuit Breaker Patterns in Microservice Deployments," *Ncirl.ie*, 2024, doi: https://norma.ncirl.ie/8005/1/ritika.pdf.
[25] H. V. Sreepathy, B. Dinesh Rao, M. Kumar Jaysubramanian, and B. Deepak Rao, "Data Ingestions as a Service (DIaaS): A Unified Interface for Heterogeneous Data Ingestion, Transformation, and Metadata Management for Data Lake," *IEEE Access*, vol. 12, pp. 156131–156145, 2024, doi: https://doi.org/10.1109/access.2024.3479736.
[26] S. overflow, "Performance - Spring Boot - Server Response Time," *Stack Overflow*, 2018. https://stackoverflow.com/questions/52648831/performance-spring-boot-server-response-time
[27] Sachin Gupta, "Advanced Encryption Techniques for Database Security in Cloud Environments," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 5, pp. 611–621, Nov. 2024, doi: https://doi.org/10.32628/cseit241051048.
[28] Citibank, "Spring by Citi," *Citibank.com*, 2025. https://www.citibank.com/icg/sa/spring/ (accessed Dec. 23, 2025).
[29] Amazon, "Apache Flink," *Amazon Web Services, Inc.*, 2024. https://aws.amazon.com/managed-service-apache-flink/
[30] A. Chary Suroju, "Integration of Zuora Billing System in Microservices Architecture: A Spring Boot Implementation," *Journal of Computer Science and Technology Studies*, vol. 7, no. 4, pp. 1060–1068, May 2025, doi: https://doi.org/10.32996/jcsts.2025.7.4.120.
[31] M. Dayarathna and S. Perera, "Recent Advancements in Event Processing," *ACM Computing Surveys*, vol. 51, no. 2, pp. 1–36, Jun. 2018, doi: https://doi.org/10.1145/3170432.
[32] T. Pikkumäki, "Comparison of Monolithic, Micro-service, and Cloud development," *Theseus.fi*, 2023, doi: https://www.theseus.fi/handle/10024/795180.