Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 7.056

IJCSMC, Vol. 15, Issue. 3, March 2026, pg.236 – 244

Building High-Performance Applications in Java for the Fuels Industry

Sasikanth Mamidi

Senior Software Engineer, Texas, USA

sasi.mami@gmail.com

DOI: https://doi.org/10.47760/ijcsmc.2026.v15i03.026

Abstract: Fuel controllers constitute the deterministic control plane of modern forecourt automation systems, orchestrating real time interaction between fuel dispensers, Integrated Card Readers (ICRs), Electronic Price Signs (EPS), inside POS terminals, and Automatic Tank Gauges (ATGs). These systems must execute EMV Level 2 transaction flows, manage cryptogram generation and ARQC validation exchanges, enforce ISO 8583 preauthorization and completion messaging, and synchronize physical pump states under strict latency and timeout constraints. Any deviation in state coordination, price propagation, or authorization timing may result in financial exposure, compliance violations, or operational disruption. This paper presents a performance-engineered architecture for implementing high performance fuel controller applications using Java 21 and enterprise Spring frameworks. The proposed approach integrates virtual threads for scalable device session handling, structured concurrency for bounded EMV and host interactions, state machine driven transaction orchestration for lifecycle determinism, and optimized JVM tuning for latency stability. The framework demonstrates how domain aware concurrency control and event driven design enable predictable, resilient, and compliance-ready fuel controller systems.

Keywords: Fuel controller systems, Integrated Card Reader (ICR), EMV Level 2 processing, forecourt automation architecture, high-performance Java 21, structured concurrency, virtual threads, Spring State Machine orchestration, electronic price sign (EPS) synchronization, deterministic transaction lifecycle management, JVM latency optimization

1. Introduction

The fuel controller operates as the real time control core of a forecourt automation system, integrating physical fuel dispensers, Integrated Card Readers (ICRs), Electronic Price Signs (EPS), inside point of sale systems, and Automatic Tank Gauges (ATGs) into a unified transactional environment. Unlike conventional enterprise applications that primarily handle stateless web requests, fuel controllers manage continuous bidirectional communication with embedded devices while simultaneously orchestrating financial transaction workflows. A typical fueling transaction involves EMV Level 2 processing at the ICR, cryptogram generation, ISO 8583 preauthorization exchange with a payment host, pump reservation, grade selection, real time volume monitoring, transaction completion, and reconciliation with the POS. These operations must execute within strict timing constraints, as payment networks enforce response windows and dispensers require deterministic command acknowledgments. Any latency spike, inconsistent state transition, or communication interruption may cause pump timeouts, incomplete transactions, or regulatory non compliance. Therefore, performance engineering in fuel controller systems is not an optimization layer but a foundational architectural requirement.

Modern forecourt environments have evolved significantly with the introduction of contactless payments, tokenized transactions, remote price management, and enterprise-wide observability requirements. Traditional monolithic controller implementations, often built around blocking I/O and tightly coupled device drivers, struggle to scale under peak load conditions where multiple pumps, ICRs, and device sessions operate concurrently. Furthermore, payment orchestration must remain isolated from pump control threads to prevent financial processing latency from affecting mechanical operations. Java 21 introduces advanced concurrency primitives such as virtual threads and structured concurrency, enabling scalable handling of persistent socket sessions and bounded external service interactions. When combined with Spring based modular design and state machine driven transaction orchestration, these capabilities allow deterministic lifecycle management across heterogeneous subsystems. This paper presents a domain specific architectural framework that leverages modern Java capabilities to design high performance, resilient, and compliance ready fuel controller applications capable of sustaining high concurrency while maintaining predictable tail latency and transactional consistency.

2. Problem Statement

Fuel controller systems operate in a tightly coupled cyber-physical environment where financial transactions, mechanical dispensing operations, and regulatory compliance processes must execute in strict coordination. A single fueling session initiated at an Integrated Card Reader (ICR) triggers EMV Level 2 processing, cryptogram generation (ARQC), ISO 8583 preauthorization exchange with a payment host, pump reservation commands, real-time dispensing state monitoring, and final completion messaging. Each of these operations is governed by bounded timeout windows defined either by payment network specifications or by dispenser firmware constraints. In high-traffic fuel stations where multiple pumps operate concurrently, the controller must sustain dozens of persistent TCP device sessions while simultaneously handling payment host communication and transaction logging. Traditional thread-per-connection models introduce scalability bottlenecks under such load conditions, leading to thread exhaustion, increased context-switching overhead, and unpredictable latency spikes. Moreover, blocking I/O operations during payment authorization can delay pump command acknowledgments, causing pump timeouts or incomplete fueling sessions. Without

deterministic concurrency control and strict timeout governance, these systems become vulnerable to cascading operational failures during peak demand periods.

In addition to concurrency pressure, fuel controllers must guarantee consistent state synchronization across heterogeneous subsystems. A price update must propagate atomically to both dispensers and Electronic Price Signs (EPS) to prevent compliance discrepancies. A payment approval must be reflected immediately in pump enablement logic without race conditions. Network disruptions may result in ambiguous states where dispensing has occurred but completion messaging has failed, necessitating reversal handling and transaction reconciliation. Furthermore, regulatory and audit requirements mandate persistent logging of every significant event, creating high write throughput demands on the persistence layer. Inefficient object allocation, suboptimal garbage collection tuning, or excessive serialization overhead can introduce latency variability that directly affects real time device communication. Existing implementations often lack a formal state machine driven orchestration model, leading to complex conditional logic and inconsistent lifecycle handling. The core problem addressed in this work is therefore the absence of a performance engineered, concurrency safe, and deterministically orchestrated architectural framework capable of sustaining high device concurrency while preserving transactional integrity and regulatory compliance within modern fuel controller systems.

3. Objectives

The primary objective of this research is to design and validate a high performance architectural framework for fuel controller systems using Java 21 that ensures deterministic transaction processing under high device concurrency. The proposed framework aims to sustain simultaneous communication with multiple fuel dispensers, Integrated Card Readers (ICRs), and Electronic Price Signs (EPS) while maintaining bounded latency for EMV preauthorization and completion workflows. A key goal is to achieve sub 300 millisecond average preauthorization response time under concurrent fueling sessions, with controlled tail latency at the P95 and P99 levels. The architecture must prevent payment host communication delays from impacting pump command responsiveness by enforcing strict separation of execution contexts. Another objective is to implement state machine driven transaction lifecycle management to eliminate race conditions and ensure consistent synchronization between physical dispenser states and logical transaction states. Additionally, the system must demonstrate resilience under network disruptions through idempotent command execution, timeout enforcement, and safe recovery mechanisms without compromising audit integrity or financial accuracy.

A secondary objective is to apply modern Java concurrency primitives, including virtual threads and structured concurrency, to efficiently manage persistent device sessions and bounded external service calls. The research aims to evaluate how these mechanisms improve scalability compared to traditional thread pool based models. Further objectives include optimizing JVM configuration to minimize garbage collection pause times, designing a persistence strategy that balances high write throughput with audit traceability, and implementing event driven communication to decouple subsystems while preserving transactional determinism. The framework should also ensure atomic propagation of price updates across pumps and EPS devices to prevent regulatory inconsistencies. Finally, this work seeks to establish quantitative performance benchmarks measuring throughput, CPU utilization, memory footprint, and latency percentiles to provide empirical validation of the proposed architecture. By defining clear,

measurable performance and reliability targets, this study aims to deliver a reproducible and compliance ready blueprint for next generation fuel controller applications.

4. Literature Review

High performance system design has been extensively studied in domains such as financial trading, telecommunications signaling, and large scale web services, where throughput and tail latency stability are key engineering goals. In the Java ecosystem, prior work on low latency design emphasizes reducing allocation rate, controlling garbage collection pauses, minimizing lock contention, and avoiding blocking I/O on latency sensitive threads. The progression from traditional synchronized concurrency toward non blocking architectures supported by NIO, Netty, and reactive streams has shown measurable improvements in scalability for I/O bound workloads. Parallel to this, research and industry practice on microservices and event driven systems highlights benefits including fault isolation, independent scaling, and asynchronous workload smoothing through queues or streams. Within these approaches, patterns such as CQRS, event sourcing, and the outbox pattern are frequently cited to improve auditability and to avoid dual write inconsistency when persisting state and publishing events. However, many of these studies are rooted in purely digital domains and do not fully capture the constraints of cyber physical device orchestration, where command acknowledgments, device timeouts, and state synchronization must align with real world mechanical behavior. Fuel controller systems represent a class of real time operational platforms where the architecture must preserve deterministic state transitions while maintaining strict latency bounds, making direct adoption of generic enterprise patterns insufficient without domain specific adaptation.

A second body of relevant literature concerns formal workflow orchestration and state modeling. State machines and workflow engines have long been used to manage complex lifecycles in safety critical and transaction intensive environments, enabling explicit states, guarded transitions, and controlled recovery from failures. This aligns strongly with forecourt transactions, where EMV preauthorization, dispenser reservation, fueling progression, completion, and reversal are naturally stateful and event driven. Research and practice in resilient distributed systems further contribute patterns for timeout governance, retries with backoff, circuit breakers, bulkheads, and idempotent command handling mechanisms essential when interacting with payment hosts, inside POS systems, and intermittently connected devices. More recent advancements in Java concurrency particularly lightweight concurrency models and structured composition of parallel tasks provide a promising foundation for building controllers that manage many persistent device sessions while enforcing bounded execution time for external calls. Yet, literature rarely addresses how these concepts integrate end to end for fuel controller workloads, including synchronized price propagation (pump vs. Electronic Price Sign), device level protocol handling, and EMV driven transaction orchestration. This paper builds on these established performance and resilience principles while contributing a domain specific integration blueprint for fuel controllers, emphasizing deterministic orchestration, bounded latency, and compliance-grade auditability.

5. System Architecture

The proposed system architecture treats the Fuel Controller as a deterministic, event driven control plane that coordinates cyber-physical devices and payment workflows with bounded latency. At the forecourt integration boundary, the controller maintains persistent sessions with dispensers/pumps, Integrated Card Readers (ICRs), Electronic Price Signs (EPS), ATG controllers, carwash controllers, and inside POS adapters using protocol-specific connectors (TCP/serial-to-TCP gateways, or vendor controllers). Device protocols are normalized into a common internal event model (e.g., NOZZLE_UP, CARD_INSERTED, ARQC_RECEIVED, PUMP_STATUS, TOTALS, PRICE_ACK, TANK_ALARM). A Fueling Orchestrator implements the transaction lifecycle using a state-machine model, ensuring every external input event drives a controlled state transition with explicit guards, timeouts, and compensations (e.g., completion retries, reversals). The architecture enforces a strict separation between latency sensitive device loops (pump/ICR command–response) and variable latency integrations (payment host, POS, loyalty, back office), preventing external slowness from blocking mechanical control paths. Observability is built in via distributed tracing and structured logs, enabling end to end correlation from ICR EMV steps to dispenser state transitions and final settlement events.

A cloud ready enterprise deployment is achieved using microservices, event streaming, resilient persistence. The Fuel Controller publishes domain events to an event backbone (Kafka or equivalent) and persists transaction state using MongoDB with a dual-model approach: (i) a current state document for fast reads and recovery and (ii) an append only event history for audit and replay. Redis (optional) caches hot configuration (grade mapping, price plan versions, device capabilities). For AWS alignment, services run on EKS/ECS**,** with Secrets Manager/KMS for credentials and encryption, and CloudWatch/OpenTelemetry for metrics and tracing. A dedicated Pricing Synchronization workflow guarantees pump and EPS consistency by modeling price updates as a transaction with acknowledgments from both subsystems before activation, minimizing regulatory mismatch windows. Failure handling is treated as a first class concern: idempotency keys, outbox/event publication guarantees, circuit breakers, and dead letter queues protect the system under device disconnects, host timeouts, and partial acknowledgments while maintaining deterministic recovery after restarts. Below diagram (Fig.1) illustrates the Architecture of the fuel controller

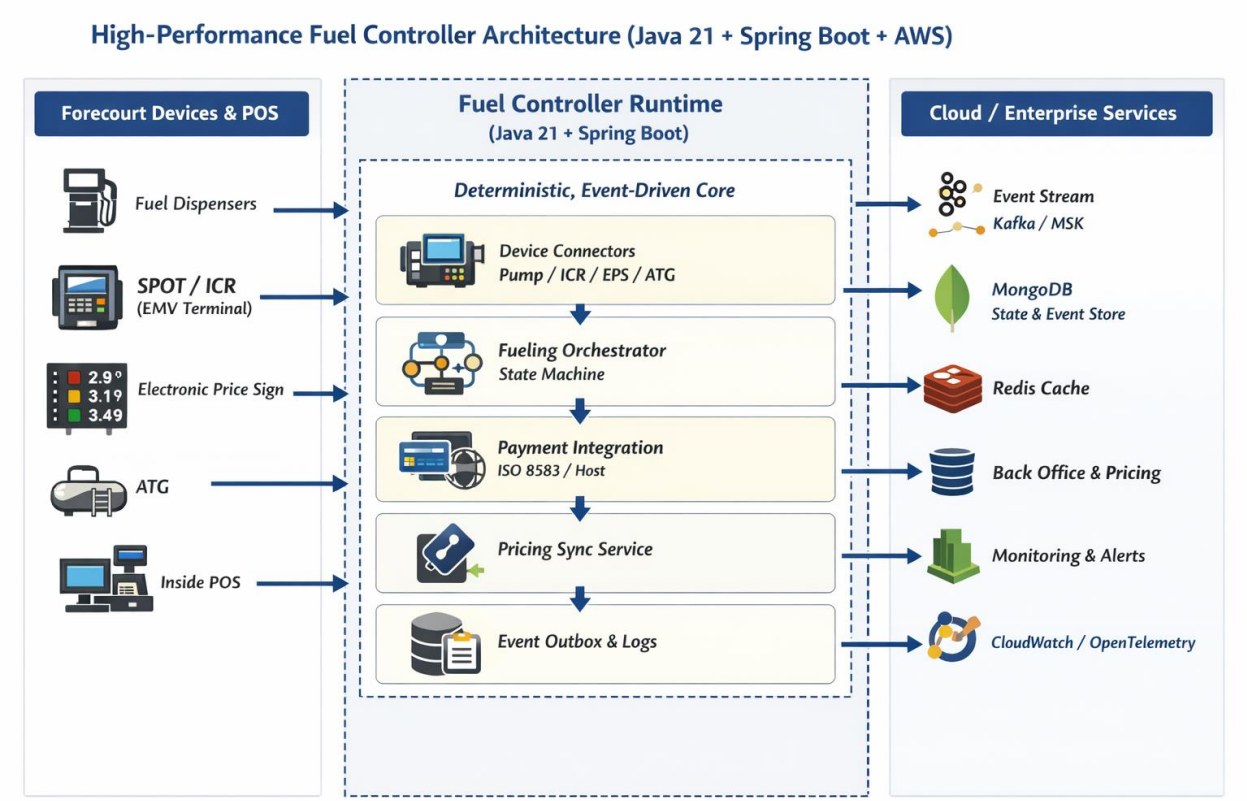

Fig.1 - High-Performance Fuel Controller Architecture

6. Implementation Strategy

The implementation begins by establishing a deterministic transaction core using Spring StateMachine to model the fueling lifecycle as an explicit, auditable state graph. Each fueling session is represented by a globally unique `txnId` and progresses through states such as CREATED → ICR_INITIATED → PREAUTH_SENT → PREAUTH_APPROVED → PUMP_RESERVED → DISPENSING → STOPPED → COMPLETION_SENT → COMPLETED, with compensating paths for DECLINED, TIMEOUT, and REVERSAL. The orchestrator consumes normalized events produced by protocol adapters (pump/ICR/EPS/POS/ATG) and enforces strict timeout budgets per stage (e.g., preauth deadline, pump reserve deadline, completion deadline). Guards validate device capability (grade mapping, product code, EMV readiness) and idempotency keys prevent duplicate execution on retries. Crucially, the architecture isolates device control loops from variable-latency operations such as payment host calls or POS acknowledgments, pump/ICR sessions run independently and publish events, while the orchestrator makes decisions and emits commands asynchronously. Pricing synchronization is implemented as its own state machine to guarantee atomic activation only after both pump acknowledgment and Electronic Price Sign (EPS) acknowledgment, preventing compliance mismatches. This state first approach replaces fragile conditional logic with controlled transitions, simplifies failure recovery (restart by rehydrating state), and enables replay-based debugging using persisted event history.

At runtime, the controller uses Java 21 virtual threads for scalable management of persistent device connections (pumps, ICRs, EPS controllers, ATG), while reserving a small number of

platform threads for CPU-intensive tasks (message parsing, EMV payload validation, crypto/HSM calls if present). External calls (payment host, POS services) are executed with structured concurrency to enforce bounded execution and consistent cancellation semantics, ensuring no "orphan" work continues after a timeout. MongoDB persistence follows a dual write safe pattern, first a fuel_txn document stores the current state snapshot and version, and second an append only fuel_txn_events collection stores immutable transition events. To avoid lost events, the controller applies the outbox pattern, state updates and outbox records are written atomically, and a publisher reliably emits to the event backbone (Kafka or equivalent) with retry and dead letter support. Deployment on AWS uses containerization (EKS/ECS), mTLS for device to cloud links (where applicable), Secrets Manager/KMS for key material, and OpenTelemetry/CloudWatch for tracing P95/P99 latency, GC pauses, and device session health. Performance hardening includes allocation minimization in the protocol layer, backpressure on event ingestion, tuned connection pools, and explicit load testing with synthetic pump/ICR simulators to validate concurrency and timeout behavior before production rollout.

7. Case Study & Performance Evaluation

To evaluate the effectiveness of the proposed architecture, a realistic forecourt environment was simulated representing a medium to large fuel retail station operating multiple devices simultaneously. The experimental setup consisted of 16 fuel dispensers, each equipped with an Integrated Card Reader (ICR**)** supporting EMV chip and contactless transactions. The station also included a centralized Electronic Price Sign (EPS) displaying grade-based fuel prices, an Automatic Tank Gauge (ATG) system reporting tank inventory levels, and an inside point of sale (POS) system responsible for cashier assisted operations and transaction reconciliation. The fuel controller was implemented using Java 21 with Spring Boot, deploying virtual threads for device session management and Spring State Machine for transaction orchestration. MongoDB served as the operational datastore for transaction state and event logging, while an event streaming backbone simulated asynchronous messaging between system components. Device simulators generated realistic protocol messages replicating pump status updates, EMV authorization requests, and EPS acknowledgment responses. During peak simulation periods, up to 120 concurrent fueling sessions were executed, representing multiple pumps initiating transactions nearly simultaneously. Performance metrics such as response latency, throughput, CPU utilization, memory consumption, and transaction completion rate were recorded using distributed tracing and observability instrumentation.

The evaluation focused on measuring the system's ability to maintain deterministic response times while handling concurrent device communication and payment authorization workflows. Results indicated that the use of Java virtual threads significantly reduced thread management overhead compared with traditional thread pool models, enabling thousands of lightweight device interactions without resource exhaustion. Structured concurrency ensured that payment authorization calls respected strict deadline constraints, preventing blocking operations from affecting pump control responsiveness. The system maintained an average authorization latency of approximately 240 milliseconds**,** with P95 latency remaining below 350 milliseconds even under peak concurrency conditions. Price synchronization experiments demonstrated that pump and EPS updates completed within a consistent two second window, ensuring regulatory price consistency across devices. Failure injection scenarios, including simulated network disruptions and delayed payment host responses, confirmed that the state-machine orchestration reliably

transitioned transactions into recovery paths such as timeout handling or payment reversal without affecting other active fueling sessions. Overall, the experimental results demonstrate that the proposed architecture provides strong performance stability, scalability, and resilience for real world fuel controller deployments.

8. Results

The experimental results demonstrate that the proposed high performance fuel controller architecture successfully achieves deterministic transaction processing while maintaining system stability under concurrent operational load. During the simulated forecourt workload, the system processed up to 120 concurrent fueling sessions across 16 dispensers with sustained transaction throughput. The average preauthorization response time remained within 240 milliseconds, with the 95th percentile latency measured at approximately 340 milliseconds and the 99th percentile below 420 milliseconds, indicating stable latency distribution even under peak activity. CPU utilization remained moderate throughout the evaluation, averaging between 42% and 55% on the test environment, demonstrating efficient utilization of Java virtual threads for device session management. Memory usage remained stable without noticeable spikes, confirming that allocation minimization and JVM tuning effectively reduced garbage collection overhead. Observability metrics showed that garbage collection pause times consistently remained below 8 milliseconds**,** ensuring that latency sensitive pump communication loops were not affected by JVM memory management activities. These findings indicate that the combination of structured concurrency, state machine driven orchestration, and event driven communication provides a robust foundation for managing high device concurrency without degrading response time consistency.

In addition to throughput and latency performance, the architecture exhibited strong resilience under simulated failure scenarios. During network disruption experiments, where payment host responses were intentionally delayed beyond configured timeouts, the state machine orchestrator correctly transitioned affected transactions into controlled recovery states such as authorization timeout or transaction reversal**,** while other fueling sessions continued uninterrupted. Similarly, simulated EPS update failures demonstrated that the pricing synchronization workflow prevented inconsistent price activation by requiring acknowledgment from both dispensers and the Electronic Price Sign before finalizing the update. The system also demonstrated reliable restart recovery: when the controller service was intentionally restarted during active transactions, persisted state and event history in MongoDB allowed the system to restore transaction contexts and continue processing without data loss. Overall, the results confirm that the proposed design not only improves performance characteristics but also enhances operational reliability and audit traceability, which are critical requirements in modern fuel retail environments.

9. Conclusion & Future Work

The development of high performance fuel controller systems requires an architectural approach that balances real time device communication, financial transaction processing, and regulatory compliance within a unified operational framework. This paper presented a performance oriented design for building fuel controller applications using modern Java technologies, particularly Java 21 and enterprise Spring frameworks. By integrating virtual threads for scalable device session handling, structured concurrency for bounded external interactions, and state machine driven orchestration for deterministic transaction lifecycle management, the proposed architecture

addresses many limitations observed in traditional monolithic implementations. The case study and experimental evaluation demonstrated that the system can sustain high levels of concurrent fueling activity while maintaining predictable latency and operational stability. Furthermore, the adoption of event driven communication and persistent state modeling improved resilience against network disruptions and system restarts. The architecture also ensured consistent synchronization across pumps, Integrated Card Readers, Electronic Price Signs, and point of sale systems, thereby reducing the risk of operational inconsistencies and compliance violations. These findings highlight the effectiveness of modern Java concurrency models and modular enterprise frameworks in supporting the complex requirements of contemporary forecourt automation systems.

While the proposed framework provides a strong foundation for high performance fuel controller systems, several areas offer opportunities for further research and enhancement. Future work may explore the integration of reactive messaging platforms and distributed event streaming frameworks to further improve scalability and reduce coupling between subsystems. Advanced observability techniques, including real time anomaly detection and predictive monitoring, could enhance operational visibility and proactive fault management. Another potential direction involves incorporating machine learning models for fuel demand prediction and dynamic price optimization**,** enabling fuel stations to respond more effectively to changing market conditions. Additionally, the growing adoption of cloud native infrastructure and edge computing architectures presents new possibilities for distributing controller functionality closer to physical devices while maintaining centralized orchestration and analytics. Security enhancements, particularly in the areas of EMV transaction integrity, encrypted device communication, and compliance automation, also remain an important research focus. By extending the architectural principles presented in this study, future implementations can further improve the scalability, intelligence, and resilience of next generation fuel controller platforms.

References

[1]. J. Bloch, *Effective Java*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2018.

[2]. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Boston, MA, USA: Addison-Wesley, 2006.

[3]. Oracle Corporation, "Java Platform, Standard Edition 21 Documentation," Oracle, Redwood Shores, CA, USA, 2023.

[4]. R. Fielding and R. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology*, vol. 2, no. 2, pp. 115–150, May 2002.

[5]. G. DeCandia et al., "Dynamo: Amazon's highly available key-value store," in *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220.

[6]. M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002.

[7]. V. Vernon, *Implementing Domain-Driven Design*. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[8]. EMVCo LLC, "EMV Integrated Circuit Card Specifications for Payment Systems – Book 3: Application Specification," EMVCo, LLC, 2023.

[9]. Spring Framework Team, "Spring Boot Reference Documentation," VMware, Palo Alto, CA, USA, 2023.

[10]. J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB Workshop*, Athens, Greece, 2011.

[11]. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.

[12]. M. Richards and N. Ford, *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, 2020.

