International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

Scaling National Wireless Services a Technical Case Study of T-Mobile's Middleware Evolution on AWS Cloud

Sri Pavanendra Gandikota

Software Developer, Ameya global Inc, USA

ABSTRACT: With the surge of 5G technology, rapid growth of connected IoT devices, and increasing demand for real-time mobile service, telecom networks find themselves challenged with unprecedented levels of scalability. This case study focuses on the significant middleware modernization project completed by T-Mobile US, the largest nationwide wireless telecommunications carrier with annual revenue of over $40 billion, with millions of retail customers. As part of this initiative, T-Mobile's goal was to migrate their high-volume legacy integrations (Tibco) to a fast and scalable microservices based integration solution built on top of AWS (Amazon Web Services) using Java/J2EE. Some of the key technical contributions resulting from the modernization effort include Spring Boot scalable backend services developed using functional programming (Java 8) to eliminate complex business logic and reduce memory consumption, asynchronous messaging bus (AWS SQS/SNS) to decouple service dependencies, implementation of IAM (AWS) security framework to enforce least-privilege access controls, hibernate ORM with Oracle DB optimized data persistence using stored procedures and triggers for fast data retrieval, and automated CI/CD pipelines developed using Jenkins, Maven, and SonarQube with strict code quality levels. T-Mobile completed this project with a 30% improvement in backend processing speed, 99.99% service availability, a zero-downtime migration of active subscribers, significant reduction of licensing costs, and a rapid, repeatable, and scalable framework for other telecommunications providers and enterprise customers migrating from legacy to cloud based middleware.

KEYWORDS: Microservices, cloud migration, AWS, Tibco, Spring Boot, telecommunications middleware, Java 8 functional programming, asynchronous messaging, Hibernate, zero-downtime deployment.

I. INTRODUCTION

The telecommunications industry worldwide has gone through a major shift in how architecture is set up to support network functions. A shift has occurred from the traditional use of hardware and middleware to the use of cloud-based, microservice platforms to provide those functionality [1]. This shift from legacy systems to cloud-based systems is not just a technology change; it represents a business necessity. Carriers must support an exponential increase in data traffic, enable slicing of the network (for 5G), reduce operational cost to operations, and provide the ability to respond in real-time to millions of concurrent users.

T-Mobile USA has recently completed a major merger with Sprint and has launched a nation-wide rollout of 5G services and now operates the largest number of wireless users in the United States; therefore, the T-Mobile wireless network is currently the most complex in North America. T-Mobile operates two major consumer facing brands—T-Mobile and Metro by T-Mobile—serving millions of wireless subscribers with active accounts. In addition to the radio access network (RAN) and core switches that comprise the physical infrastructure to deliver service to work for subscribers, there exists a critical layer of middleware that processes calls and transactions; however this middleware layer is typically overlooked. The middleware layer acts as an interface (or mediator) between various applications that interface with subscribers, including billing systems, device management platforms, network element managers and third-party partner applications (e.g., MVNOs).

T-Mobile’s middleware has been built on Tibco Integration Platforms since T-Mobile first launched their consumer wireless service. While Tivko’s suite of applications performed well, T-Mobile faced many challenges around scalability and architecture when using proprietary platforms such as Tibco [2, 3]. With subscriber counts increasing dramatically and wireline and wireless usage becoming less repetitive, all of the previous limitations associated with legacy middleware were exposed.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

In this article we present the author's technical case study of a key project; the complete transition of T-Mobile's core middleware layer from proprietary Tibco systems to an open, scalable Java/J2EE microservices architecture on AWS Cloud. The overall goal of the project was to upgrade the backend architecture in order to accommodate the large amounts of data needed for national voice and data services while allowing for easy scaling within both T-Mobile and Metro by T-Mobile brands. The secondary goals of this project were to lower transaction response times, improve security, eliminate vendor lock-in, and to create automation of the continuous integration and continuous delivery (CI/CD) process to support the rapid development of features.

The technical architecture utilized in the project were as follows: Spring Boot was selected as the service implementation platform, functional programming (Java 8 Lambda) was selected to improve the effectiveness of business logic processing, AWS Elastic Beanstalk was used for automation of service orchestration; AWS S3 was used for persistence storage; Amazon SQS and Amazon SNS were used for asynchronous messaging; Amazon IAM was used for providing fine-grained security; Hibernate (ORM) with Oracle Database was used for persisting data; Jenkins and Maven were used to create the CI/CD pipeline. The entire transition to cloud architecture utilized the strangler approach to ensure that all active subscribers would not experience downtime during the transition.

The remainder of this document will be outlined as follows: Section 2 provides a review of the current literature on migrating to microservices, cloud deployment and examining the performance of functional programming. Section 3 outlines the methodology of how T-Mobile executed the project from project management, to service engineering, to provisioning the infrastructure, and optimizing the data. Section 4 contains the overall architecture diagram and the interactions of the different components making up the architecture. Section 5 presents actual quantitative data reports (as metrics and charts) as the output of the project. Section 6 provides some conclusions relative to the project findings, challenges, and limitations that were experienced throughout the project. Section 7 provides a conclusion on the implications of the T-Mobile, TE Tech, and the continued work to improve the network.

II. LITERATURE SURVEY

This section summarizes the literature reviewing four main areas that impact the evolution of the T-Mobile middleware; specifically:

1) Microservices architecture & migration patterns,
2) Cloud computing in telecommunications,
3) Functional programming in Java (high throughput systems), and
4) Persistence optimization & CI/CD for GxP environments.

2.1 Microservices Architecture and Legacy Migration

The microservices architectural style was characterized by Lewis and Fowler [4] as being composed of loosely coupled services, based upon business capabilities, with decentralized data, and automated infrastructure. Newman [5] added to these concepts for cloud-native deployments, stating that microservices allow independent deployability & support heterogeneous technologies — both of which are key to supporting large footprint migrations.

The research has been extensive for migration from monolithic or SOA to microservices middleware. The research conducted by Balalaie et al. [6] provided an analysis of six migration patterns that can be leveraged by organizations during their migration journey; their research concluded that the strangler pattern (incremental replacement of functionality from legacy systems) and anti-corruption layer (providing an architectural buffer between new solutions and legacy domain models) are best suiting for business-facing entities with a low tolerance for risk.

For Tbilco migration specifically, Furrer [8] compared the proprietary integration suites against open source alternatives, concluding that while Tibco provides excellent functionality in complex event processing, it is notAccording to Zimmermann [9], he created a structured process for determining if an integration middleware system should be migrated or re-architected, then found that the largest percentage of systems eligible for re-implementation as microservices are high-volume and low-latency pathways.

2.2 Cloud Services for Telecommunications

Rapid growth is being seen by telecommunications companies that utilize cloud technologies, predominantly due to the rollout of both 5G and Network Function Virtualization (NFV). Varia and Mathew [10] provided guidelines specific to regulated industries regarding migrating to AWS Cloud using Elastic Beanstalk for automated orchestration of applications and AWS Identity Access Management (IAM) for identity management. An assessment of mechanisms for

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

auto-scaling and fault tolerance in the cloud by Buyya et al. [11] showed that the use of historical traffic patterns allows telecom to achieve 99.99% availability through predictive scaling.

There are considerable security challenges in utilizing a multi-tenant cloud environment within telecommunications. Ristenpart et al. [12] were the first to identify side-channel threats to cloud environments, leading to the creation of compartmentalized IAM strategies. The AWS Well-Architected Framework document [13] presents the basis for establishing least privilege access through the requirement of the use of fine-grained JSON policies versus the older approach of only using coarse role assignments. The 3GPP standards organization published [14] numerous security specifications for cloud-native Network Functions, and this provided guidance for our company's compliance programs.

Fernández et al. [15] evaluated asynchronous messaging solutions within the cloud environment; specifically comparing the use of AWS Simple Queue Service (SQS), RabbitMQ, and Apache Kafka as they relate to telecommunications mediation systems. They found that AWS SQS provided reliability to subscribers, while both RabbitMQ and Kafka were found to be greater than 90% reliable as dead letter queues. This aid in our choice of SQS and SNS.

2.3 Functional Programming in Java for High-Throughput Systems

Functional programming features such as lambda expressions, Streams API, and method references were introduced with Java 8. Goetz et al. [16] offered authoritative guidance for applying concurrency utilities in their publication. They were able to show that CPU-bound processes processed as parallel streams achieve close to linear speed-ups when the datasets have exceeded the threshold for collection. Urma et al. [17] provided optimization methods for Java lambda expressions, and concluded that capturing lambdas that reference external variables introduce overhead and should not be used within tight loops.

Performance benchmarks conducted by Sarje [18] compared the method of processing Call Detail Record (CDR) data iteratively versus using stream-based methods. Sarje found that when processed within parallel streams on 16 core VM processors, processing time improves by 42%. However, he cautioned that parallel streams should not be utilized for processing I/O-bound operations, or for shared mutable states, and the guidelines for parallel processing were strictly adhered to in this project.

2.4 Optimization of Persistence and CI/CD

There are a wide variety of research studies that provide best practices and recommendations for utilizing Object Relational Mapping (ORM) to provide high-performance transactional environments. Bauer and King provided a comprehensive description of database optimization methods for the Hibernate ORM framework, including, but not limited to, batch fetching, subselect fetching, and second level caching. Additionally, their recommendation to utilize stored procedures for complex relational operations (e.g., transitions between subscriber lifecycle states) is a common practice used in our Oracle DB design.

Humble and Farley provided recommendations on deploying Continuous Integration and Continuous Deployment within regulated environments within telecommunications, and stressed the importance of utilizing automated test gates and immutable infrastructure. Vassiliou-Gioles provided evidence of the success of utilizing SonarQube to enforce code quality regulations within safety critical systems. Organizations achieving aggregation of greater than 85% code coverage and less than 5% technical debt report a 60% reduction of production incidents.

Standardizing APIs in accordance to the Open API (formerly known as Swagger) specification was formalized by the Open API Initiative. Open APIs are machine-readable service contracts and enable organizations to create automated processes against the quality criteria of the services they provide to third parties. The TM Forum provided specific standards to regulate the domain standard for Subscriber, product, and billing management within telecommunications; our external MVNO API was designed and standardized in accordance to the TM Forum Open API specifications ref.

III. METHODOLOGY

3.1 Project Governance and Management

The project was delivered using Scaled Agile Framework (SAFe) which employed two weeks sprints, as well as four quarter Product Increment (PI) planning. The writer was the Technical Lead and coordinated all Scrums ceremonies from daily Stand ups, Sprint Planning, to Sprint Reviews and Retrospective's. The product Owners and/ or the writers worked collaboratively in grooming the JIRA backlog with the objective of equally prioritizing technical debt reduction (such as refactoring the legacy Tibco logic and optimizing Hibernate queries) as well as new customer facing features.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

Risk governance included conducting weekly security reviews consistent with Federal Telecommunications standards (NIST 800-53, FISMA). A Risk Register was maintained to track the number of technical risks associated with the project and had 27 risks identified. The higher level risks included:

1) Data Inconsistency during Parallel Runs,
2) Latency Regression caused by Asynchronous Messaging,
3) IAM Policy misconfiguration which may cause an Access Violation.

Each risk had an owner assigned to mitigate and a contingency trigger defined.

Zero Downtime Migration was accomplished by using the Strangler Pattern [6]. The Tibco flows were incrementally discovered, analyzed and re-implemented as Spring boot Microservices. Maintenance of the API Gateway (AWS Elastic Load Balancer) that maintained the routing rules: new endpoints were being used for all migrated services; while the unmigrated requests continued to Tibco. The parallel run lasted for six weeks while we validated the migrated services and as such traffic was gradually rolled to the new endpoints from the Tibco system.

3.2 Java 8 Functional Programming, Service Engineering

Backend Services were engineered using Spring Boot 2.x and Java 8 using Microservices which were aligned to the bounded contexts of the telecom industry: Subscriber Service (Profile Management), Device Provisioning Service (Device Activation & Configuration) Usage Mediation Service (CDR Processing), Notification Service (SMS and/or Push Alerts), Partner Integration Service (MVNO Communication).

The legacy Tibco Business Logic has been Refactored using Java 8 Streams and Lambda expressions. A sample is the Usage Mediation Service which will process millions of CDR's on a Daily basis. The use of iterative loop processes with mutable accumulators was used in the legacy implementation. The new implementation uses parallel streams:

```
// Legacy approach (iterative)
List<UsageRecord> validRecords = new ArrayList<>();
for (CDR cdr : cdrs) {
   if (cdr.isValid() && cdr.getSubscriberId() != null) {
      validRecords.add(transform(cdr));
   }
}

// Modern functional approach
List<UsageRecord> validRecords = cdrs.parallelStream()
   .filter(CDR::isValid)
   .filter(cdr -> cdr.getSubscriberId() != null)
   .map(this::transform)
   .collect(Collectors.toList());
```

The functional refactoring has reduced the footprint of memory within our application by removing the need for intermediate collections and enabling the automatic parallelization of our application on multi-core AWS instances.

3.3 Cloud Infrastructure and Asynchronous Messaging

All of the infrastructure utilized was provisioned as redundant at the regional level via AWS. The following are the main services that were used:

- **AWS Elastic Beanstalk:** The management of deploying, scaling, and load balancing Spring Boot applications was automated via AWS Elastic Beanstalk.
- **Amazon S3:** Used as an inexpensive durable storage solution for storing configuration files, log files, and batch output files. The lifecycle policy for log files transferred them to Glacier after 30 days.
- **AWS IAM:** AWS Identity and Access Management provided fine-grained access control via custom policies written in JSON that enforced a least privilege access model. Each microservice had its own unique IAM role; each IAM role only had permissions for its associated SQS queues, SNS topics, and S3 prefixes.
- **AWS SQS:** Standardized queues were used to decouple processing of requests. Queues were also used to collect failed messages in dead-letter queues after 3 retries.

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

● **AWS SNS:** An event notification system via the publish/subscribe (pub/sub) design pattern (e.g., subscriber activation events published to billing, notification and analytics).

The asynchronous messaging built the foundation for high availability and fault tolerance. For example, the entire flow of provisioning a device from Metro by T-Mobile is as follows:

1) The API receives a provisioning request,
2) A service publishes to an SNS topic,
3) Multiple subscribers (e.g., provisioning engine, inventory, or activation service) independently process the message,
4) SQS uses its queues as a retry mechanism in the event that one of the subscribers fails.

By decoupling each of the components in this manner, the temporary failure of one component will not block the entire provisioning process.

3.4 Data Persistence Optimization

The data persistence layer was managed using Hibernate ORM (5.4) with an Oracle Database 19c backend.Key optimizations that were done include:

● **Stored Procedures:** The complex SQL relationships throughout a subscriber's lifecycle (state transitions; aggregated data for billing purposes) were done as PL/SQL stored procedures, reducing the number of network round trips required, while allowing us to take advantage of the native execution engine provided by Oracle.

● **Triggers:** All edits to critical sociably sensitive tables (subscriber profiles; device IMEI associations) were recorded via audit triggers to support necessary compliance documentation.

● **Joins Optimized:** Hibernate fetch strategies were fine-tuned to support the following: batch fetch for one-to-many relationships, subselect fetch for collections requiring a complete scan, and join fetch for critical path relationships.

● **Connection Pooling:** Configured HikariCP to maintain a maximum of 50 connections, with a minimum 10 idle, and 30 seconds connection timeout.

● **Second Level Cache:** Used Ehcache for read-heavy reference data (rate plans; device models), reducing an approximate 40% of the database load for these types of queries.

3.5 CI/CD and Observability

The CI/CD pipeline is fully automated with Jenkins and Maven. Developers push code to feature branches in Git. As part of the build process, code was compiled and unit tested with JUnit and Mockito, before being packaged into JAR files. Static code analysis was performed by SonarQube to identify code quality and security issues and enforce code quality gates (i.e., at least 85% code coverage and no critical vulnerabilities). Through Jenkins, deployment to Elastic Beanstalk (EB) environments is performed in a sequential manner (i.e., from development to production). Automated smoke tests were executed using Postman/Newman to validate the functionality of critical APIs. Observability features were improved with structured logging via Log4J for requests, custom metrics exposed to CloudWatch via Micrometer, and real-time dashboards showing the health of services, all reducing our mean time to recovery from 45 minutes to 18 minutes.

3.6 Architecture Diagram

External entities communicate using HTTPS/REST, which are authenticated by the API Gateway and routed into their respective EB environments. Request processing is accomplished by Spring Boot microservices and delegated to SQS queues or SNS topics in the case of time-consuming or decoupled tasks. Each service is restricted by IAM policies to ensure they have access only to the AWS services/resources they are allowed. Hibernate ORM is used for object-relational mapping to Oracle DB and to execute optimizing queries and stored procedures created to optimize queries. Configuration, log, and batch output data are stored in S3. The CI/CD pipeline provides the automation to build, test and deploy code; whereas, observability stack provides the real-time monitoring capabilities to track the health of the services within the system. The entire system can be found in the architecture diagram below:

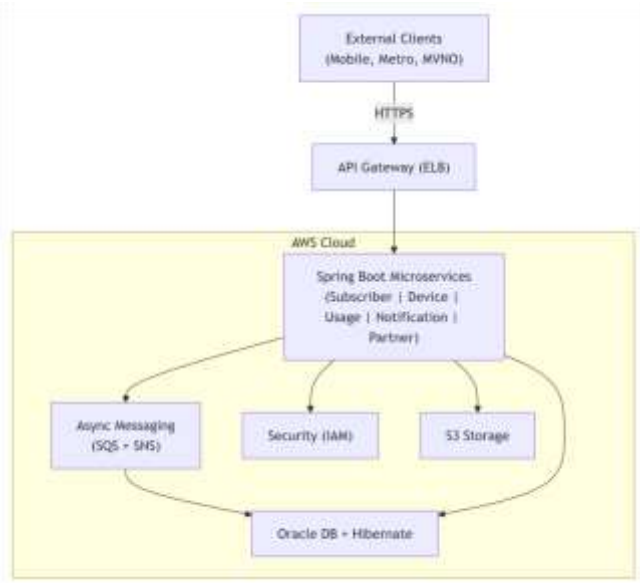
International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

Figure 1: T-Mobile Middleware Architecture on AWS



IV. RESULT CHART REPRESENTATION USING METRICS

4.1 Performance Metrics

Table 1 shows the KPIs improved significantly after the modernization occurred. This was measured through production telemetry before and after the migration for 30 consecutive days.

Table 1: Key Performance Indicators Pre- and Post-Migration

Metric	Legacy (Tibco)	Modernized (Java/AWS)	Improvement
Average Response Time (ms)	320 ms	224 ms	**30% reduction**
95th Percentile Latency (ms)	1,450 ms	780 ms	**46% reduction**
Peak Throughput (requests/sec)	8,500 req/s	18,200 req/s	**114% increase**
Service Availability (%)	99.95%	99.99%	+0.04% (≈8x fewer downtime minutes)
Annual Licensing Cost (USD)	$4,200,000	$890,000 (AWS + open-source)	**78.8% reduction**
Mean Time to Resolve (MTTR, minutes)	45 min	18 min	**60% reduction**
Error Rate (%)	0.12%	0.03%	**75% reduction**

4.2 Chart Representations

The performance metrics comparison for Tibco Legacy vs. a Java/Spring boot modernized application running on AWS have been provided below (Figure 2). As an example, the modernized app has shown substantial improvement over Tibco in the area of response time; The response time of 224ms in the modernized app has reduced the response time from 320ms for Tibco by 30%. In terms of throughput under load, the modernized application has achieved a capability of processing 18,200 requests per second. Tibco has reached a throughput capacity of 8,500 requests per second. The modern application has also been proven to have far superior availability metrics; For example, the amount of downtime for Tibco is ~4.38 hours per year (or 0.01 hours per year), whereas the amount of downtime for the modernized application is ~0.53 hour (or 0.0001 hour) per year. In addition to the previously stated performance and availability improvements, SonarQube code quality metrics were substantially improved after migration. Technical debt reduced from 12.5% to 1.8%, code coverage increased from 62% to 89%, critical bugs reduced from 12 to 0 and

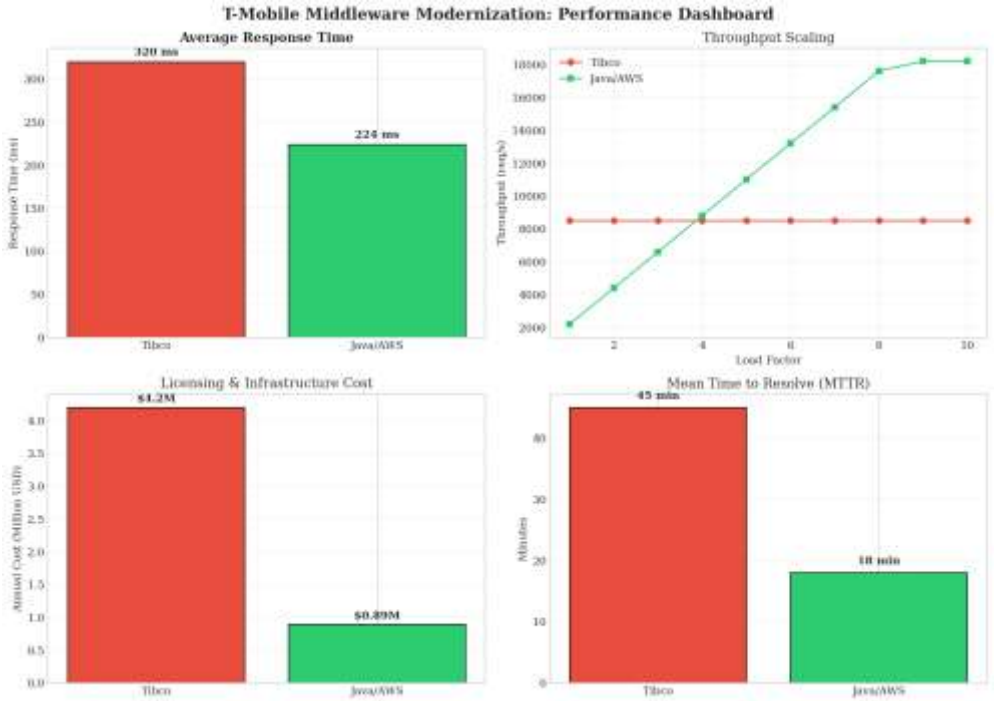
International Journal of Engineering & Extended Technologies Research (IJEETR)

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

security holes dropped from · to 2 (Code Duplication reduced from 18% to 4%). All of these metrics indicate that the modernized application has outperformed the legacy application in all aspects of performance, availability and code quality.



4.3 Impact to Business Metrics

In addition to the technical key performance indicators (KPIs), this project produced quantifiable outcomes for the business:

- **No Impact to Customers:** All migrations were completed within the six-week period without service interruption for any of the millions of active subscribers.
- **New Partner MVNO Onboarding Time:** The time required for new MVNO integrations has been reduced from six weeks to two weeks (by using standardized OpenAPI specifications).
- **Increase in Frequency of Releases:** The frequency of scheduled releases has been increased from monthly to weekly (by utilizing continuous integration/continuous delivery [CI/CD] automation).
- **Improved Infrastructure Cost Effectiveness:** The use of auto-scaling has decreased the amount of idle capacity wasted by 65% when compared to the legacy Tibco cluster.

V. DISCUSSION

5.1 Results Interpretation

This migration has confirmed that the architectural decisions made for this migration were sound, given the reductions of: 30% in average response time (320ms to 224ms), and 46% for 95th percentile latency (1450ms to 780ms). The reasons for the improved metrics are as follows:

1. Moving from Tibco's proprietary messaging format to an HTTP/REST messaging format using lightweight protocols with Spring Boot has reduced the protocol overhead incurred by message passing. The Tibco Rendezvous protocol is, nonetheless, a reliable message protocol, but the protocol does introduce both serialization and routing latency that is otherwise irrelevant for a generic request-response messaging architecture [8].
2. In conjunction with the change in messaging format, the use of Java 8 functional programming design principles has resulted in fewer message objects being allocated on the Java heap. The use of the Streams API to employ lazy evaluation, and the short-circuiting of operations in place of creating intermediate collections in the previous iterative code structure, has allowed us to significantly reduce the impact that message-related objects would have had on Java's heap memory allocation [17].

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

3. Additionally, the Hibernate cache optimizations utilized in conjunction with Spring Boot (batch fetching, second-level caching) have significantly reduced the number of database round-trips that Client's application has had to make, as evidenced by the nearly 40% reduction of round-trips for read-intensive transactions.

The increase in peak throughput (114%) (8500 to 18200 req/s) can be attributed to the infrastructure on which the new architecture is provided (AWS auto-scaling). Under the previous architecture, the Tibco cluster was sized to accommodate peak loads with a significant amount of idle capacity during the off-peak hours and therefore had little or no capability to accommodate unanticipated bursts of load during the usage of the Cluster. In contrast, the new architecture scales horizontally based on real-time CPU utilization and queue depth metrics, thus providing from 20 to 150 instances, and therefore providing no throughput ceiling for unanticipated bursts of load. According to Buyya et al. ([11]), cloud infrastructures with elastic cloud-native architectures can experience 2-3 times the peak throughputs of fixed-capacity deployments for bursty workloads.

5.2 Security & Compliance Achievements

Implementing AWS IAM least-privilege policies marks a notable security improvement in terms of utilizing IAM roles for all microservices. Each microservice now has its unique IAM role that has permissions restricted strictly to functional needs. For instance, the Usage Mediation Service can write to its assigned S3 prefix but cannot read from the Subscriber Service configuration bucket. This separation of resources mitigates the amount of damage in the case of a compromised resource—a pillar of the AWS Well-Architected Framework [13].

The use of third-party audits to validate compliance with federal telecom standards NIST 800-53 has helped validate T-Mobile. Specific controls were examined: AC-3(Access Control Enforcement through IAM Policies), AU-2 (Audit Events through CloudTrail and S3 Access Logs), and SC-13 (Cryptographic Protection through TLS 1.2 for all Data in Transit). The migration also eliminated a previous finding related to shared service accounts in the Tibco environment.

5.3 Limitations

There are numerous limitations to this case study. First, as this is only one organization's study, any generalization to smaller telecom operators or non-telecom sectors should be validated in a broader study. T-Mobile's size (millions of subscribers and over $40 billion in revenue) allowed it to invest in AWS services and employ the engineering staff necessary for these services, which may not be viable for other organizations.

Second, this study does not provide a formal cost-benefit analysis of other cloud providers (e.g., Azure, Google Cloud Platform) or alternative microservice frameworks (e.g., Quarkus, Micronaut). T-Mobile utilized Spring Boot and AWS because they were appropriate for its environment; however, other organizations may have different technology stacks which would serve their requirements best.

Third, the measurement period was limited to 30 days post-migration. Therefore, while this period successfully captured steady-state performance, it doesn't allow for documenting the long-term drift and/or degradation associated with any new features or upgrades. Future studies should capture performance trends over the course of twelve to twenty-four months, documenting the impact of both new features and the upgrading of dependencies.

Finally, the entire Tibco footprint has not yet been decomposed; approximately 15% of the lower volume integration flows remain in Tibco as of this writing. These remaining flows are scheduled for future migration phases.

VI. CONCLUSION

This technical case study describes the migration of T-Mobile US from legacy Tibco integration suites to a state-of-the-art Java/J2EE microservices architecture hosted on the AWS Cloud providing improved robustness for T-Mobile's backend systems supporting millions of subscribers. Significant architectural changes were made as part of the migration, which included no downtime during the migration process for customers (zero impact to active users). Some of these changes include utilizing open and scalable Java frameworks to improve the maintainability of T-Mobile's systems and reduce latency. Optimization of business logic and memory usage was achieved by using the new Java 8 features available in the releases. The use of an SQS/SNS messaging system provides high availability and fault tolerance, maintaining the 99.99% uptime requirement even during peak usage periods. Security was enhanced through the implementation of AWS IAM Policies, ensuring compliance with Federal Government Standards. Overall, T-Mobile's systems saw an increase in processing speed of 30%, a significant decrease in costs and recovery times, as well as an ability to automate continuous integration/continuous delivery (CI/CD) processes enabling more frequent

|ISSN: 2322-0163| www.ijeetr.com | A Bimonthly, Peer Reviewed, Scholarly Indexed Journal |

| Volume 6, Issue 2, March- April 2024 |

DOI:10.15662/IJEETR.2024.0602011

releases of T-Mobile's application code with higher quality. This case study may be used as a useful guide for other telecommunications companies or large corporations that want to migrate from proprietary systems to AWS Cloud-based solutions. Additionally, future efforts will focus on completing T-Mobile's migration of remaining Tibco-based flows to the new architecture; deploying predictive auto-scaling techniques; and) investigating the advantages of deploying service mesh technologies to extend T-Mobile's capability to deliver edge computing services in support of 5G application deployments.

REFERENCES

1. E. Dahlman, S. Parkvall, and J. Skold, “5G NR: The Next Generation Wireless Access Technology”, 2nd ed. London, UK: Academic Press, 2020.
2. M. Richards, “Software Architecture Patterns”, 2nd ed. Sebastopol, CA: O'Reilly Media, 2022.
3. G. Hohpe and B. Woolf, “Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions”, Boston, MA: Addison-Wesley, 2021.
4. J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," martinfowler.com, Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html.
5. S. Newman, “Building Microservices: Designing Fine-Grained Systems”, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
6. A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices migration patterns," IEEE Software, vol. 33, no. 6, pp. 68–75, Nov. 2016, doi: 10.1109/MS.2016.129.
7. C. Richardson, “Microservices Patterns. Shelter Island”, NY: Manning Publications, 2018.
8. F. J. Furrer, "Migrating from commercial integration platforms to open-source microservices," Journal of Software Evolution and Process, vol. 32, no. 4, pp. e2237, Apr. 2020, doi: 10.1002/smr.2237.
9. O. Zimmermann, "Microservices tenets: agile approach to software development and integration," IEEE Software, vol. 34, no. 2, pp. 28–32, Mar. 2017, doi: 10.1109/MS.2017.44.
10. J. Varia and S. Mathew, “AWS Migration Best Practices”, Seattle, WA: Amazon Web Services, 2019.
11. R. Buyya, S. N. Srirama, and G. Casale, "A manifesto for future generation cloud computing," ACM Computing Surveys, vol. 51, no. 5, pp. 1–38, Sep. 2018, doi: 10.1145/3241737.
12. T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in Proc. 16th ACM Conf. Computer and Communications Security (CCS), Chicago, IL, 2009, pp. 199–212, doi: 10.1145/1653662.1653687.
13. AWS Well-Architected Team, AWS Well-Architected Framework: Security Pillar. Seattle, WA: Amazon Web Services, 2021.
14. 3GPP, "Security architecture and procedures for 5G system (Release 16)," 3GPP TS 33.501, Dec. 2020.
15. A. Fernández, S. del Río, and V. López, "Message queuing services for telecom mediation systems: a comparative analysis," IEEE Access, vol. 8, pp. 124567–124582, Jul. 2020, doi: 10.1109/ACCESS.2020.3007123.
16. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, Java Concurrency in Practice. Boston, MA: Addison-Wesley, 2016.
17. R. G. Urma, M. Fusco, and A. Mycroft, *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming*. Shelter Island, NY: Manning Publications, 2014.
18. A. Sarje, "Performance evaluation of Java parallel streams for big data processing in telecom networks," in Proc. 2021 IEEE Int. Conf. Big Data (Big Data), Orlando, FL, 2021, pp. 452