International Journal of AI, Big Data, Computational and Management Studies
Noble Scholar Research Group | Volume 7, Issue 2, 185-190, 2026
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V7I2P127

Original Article

Migration Strategies for SOAP-Based Enterprise Integrations to REST and Event-Driven APIs

Viplove Goswami
Independent Researcher, USA.

Received On: 14/03/2026 ***Revised On: 15/04/2026*** ***Accepted On: 22/04/2026*** ***Published On: 30/04/2026***

Abstract: *For many organizations, modernizing their legacy enterprise architectures has become a strategic priority to enable cloud-native computing, mobile-first delivery, and real-time data processing. SOAP was the industry-standard for structured and reliable communication in SOA for years. It is a protocol that uses XML. Yet the complexity, bulkiness and tight coupling of SOAP based integrations has increasingly been a constraint in agile development. The paper examines the measures that need to be taken in order to migrate the legacy SOAP services to REST and Event Driven Architectures. Through a synthesis of contemporary research, this paper explores incremental models (e.g., Strangler Fig, Leave and Layer), technical issues such as the mapping of XML-centric actions to resource-oriented actions, and the evolution of security protocols from WS-Security to OAuth 2.0/JWT. In addition, a pivotal transformation emerges as the Infrastructure transcends synchronous ACID transactions towards an eventual consistency, event-driven model through the Transactional Outbox pattern and Change Data Capture (CDC). The results show that effective migration requires a holistic view of all technical re-architecture, organizational readiness, observability, etc., for business not to be interrupted and system to be long term in nature.*

Keywords: *Enterprise Integration, SOAP To REST Migration, Event-Driven Architecture, Strangler Fig Pattern, API Modernization, Microservices, Data Consistency, Oauth 2.0, JSON Web Tokens.*

1. Introduction

For over 25 years, systems have evolved toward decentralization, flexibility, and real-time action. In the late 1990s, a new technology called the Simple Object Access Protocol (SOAP) emerged, which provided an exceptionally formalized, XML-based messaging protocol that could dependably communicate between distributed systems. SOAP (and WSDL) enabled enterprises to build robust SOA (service-oriented architecture) where strict contracts and transaction guarantees could be developed for complex situations. These integrations were a success in the era of monolithic enterprise systems, where a high level of vertical integration and high standards were viewed as assets rather than weaknesses.

However, the rise of web-scale applications, mobile platforms, and the Internet of Things (IoT) in the digital economy exposed the shortcomings of SOAP. The lengthiness of XML, costly performance of complex security standards (WS-Security) as well as a tightly coupled client-server architecture slow down the agility that contemporary software require. In 2000, undirected evolution pushed for an alternative strategy, the introduction of Representational State Transfer (REST), which favored lightweight principles from the world-wide web: statelessness, resource identification and uniformity. By utilizing a more efficient data format, JSON, and the easy-to-use REST architecture, RESTful APIs can easily scale when required. These characteristics which SOAP lacks make RESTful APIs the new standard for public APIs and cloud-native services within a short span of time.

There has been evolution in recent years to go toward Event-Driven Architectures (EDA), where systems communicate with one another by producing and consuming events rather than communicating directly using the request-response approach in synchronous fashion. This change is caused by the challenge of decoupling services while achieving real-time reactivity Today's enterprise faces not just a choice but the challenge to migrate seamlessly from legacy SOAP-based integrations to one of these with little or no disruption to mission-critical operations.

It will focus on "the imperative" that allows the scientific migration of support services. It studies what architectural patterns can enable such incremental modernization, which security transitions can be needed to secure distributed endpoints and which data management techniques can be applied to maintain consistency in an asynchronous world. This study seeks to develop a nuanced framework for technical leaders and architects who are treading the challenging API modernization journey, based on industry case studies and empirical data.

2. The Technical Foundation and Legacy of SOAP

In order to comprehend the migration process, it is crucial to begin with the analysis of the technical features.

SOAP was created as a protocol to allow structured communication over various transports such as HTTP, SMTP, JMS, etc. The use of XML allowed for extensibility and strong typing which has been outlined in the WSDL file. This contract-based approach gave the client interacting with a service a precise idea of what parameters were required and what responses to expect, resulting in quite reliable enterprise-to-enterprise (B2B) interactions.

The WS-* standard is supported, which make SOAP a powerful web protocol to use. The enterprise-grade requirements include WS-ReliableMessaging for guaranteed delivery, WS-AtomicTransaction for complex distributed transactions requiring ACID properties, and WS-Security for message-level encryption and signatures among others. SOAP was a more formal protocol compared to REST. It was particularly beneficial for industries like finance, healthcare, and others that required multiple legacy databases and strict regulatory compliance for each transaction.

Regardless of these strengths, the architectural rigidity of SOAP caused much operational friction. The payloads used by SOAP envelope, which has a header and a body, became quite verbose, which lead to consuming heavy bandwidths and consuming CPU cycles for serialization and parsing. This performance penalty was especially severe in high-volume settings or on mobile devices with limited processing power. The fact that SOAP is a stateful protocol i.e. it keeps track of the session across multiple requests, hampers scalability. In the cloud, you want your services to be stateless so that they can elastically load balance across multiple services. As businesses adopted microservices and agile methods, the need for a lightweight and flexible integration pattern became evident.

3. The Paradigm Shift to Restful Resource Orientation

The shift to Representational State Transfer (REST) was a shift away from the operation-focused perspective of SOAP towards a resource-focused view of the system. Essentially REST is an architectural style that uses existing protocols, mainly the HTTP protocol, to manage the transfer of resourcing identified by URIs. Through the use of standard HTTP methods such as GET for retrieval, POST for creation, PUT for updates, and DELETE for removal, REST created an interface that was convenient and usable.

The key factor that made REST successful was the use of JSON (JavaScript Object Notation) as the data format test. The size of JSON is much less than XML, giving a payload three or four times smaller. This means that response time is faster and costs less through the network. Studies from different sources have shown that RESTful web services yield better throughput and system behaviour under load than SOAP web services. They confirm earlier disclosure experiments. The performance testing is carried out in a compiled Java or C# language.

In addition, the stateless nature of REST made scaling infrastructure easier. The server does not persist a client context between requests, meaning any instance of a service can handle any request, allowing for horizontal scaling and increased resilience. Nevertheless, this “statelessness” creates a challenge for legacy migrations regarding how to implement handling for session-based workflows that are inherently present in SOAP architecture. The usual solutionwas to externalize state to distributed caches or rely on token-based authentication mechanisms which carried the context within the request.

4. The Rise of Event-Driven Architectures and Asynchronicity

Though REST brought modernity to synchronous communication, the increasing complexity of distributed systems started requiring more asynchronous communication pattern. In the case of Event-Driven Architecture (EDA) systems do not wait for direct responses to requests. Instead, they emit events about state changes to the event broker. A broker, such as Apache Kafka, RabbitMQ, or AWS EventBridge, is a highly scalable intermediary that decouples the producers from the consumers.

EDA enables decoupling and has multiple edge against traditional request-response systems The producers we use do not know who consumes their events, so they can evolve their service in an independent way, adding new consumers without modifying the old one. Asynchronous processing increases scalability. Traffic spikes at one place in the system are buffered by the message broker. This buffering would prevent cascading failures in the system architecture. Moreover, EDA can process data in real-time. Hence, it is fit for modern-day uses like IoT telemetry, fraud detection, and inventory visibility (in e-commerce).

The move to EDA, however, introduces a challenge called eventual consistency. Despite the presence of WS-AtomicTransaction in SOAP that provides immediate transactional integrity, event nodes typically eventually settle to a consistent state as they receive and process events from other event nodes. To manage a distributed transaction, we have to use a pattern like saga pattern and Separate the logic of modifying of data and fetching data by using CQRS. For organizations transitioning from SOAP, this embodies not simply a technical switch but a fundamental transformation in the modeling and handling of business processes.

5. Strategic Patterns for Incremental Modernization

The migration of mission-critical enterprise integrations is rarely a "big bang" event. Such approaches are fraught with risk, including long release cycles, hidden dependency failures, and significant business disruption. Instead, research points toward incremental modernization patterns that allow for continuous value delivery and risk mitigation.

5.1. The Strangler Fig Pattern

The Strangler Fig Pattern, offered by Martin Fowler, is the main approach used to tear down legacy monolithic systems. The strangler fig tree grows around another tree,

eventually replacing it. In this pattern, you build brand new RESTful or event-driven services around your existing legacy SOAP services. An API gateway or proxy layer is introduced to reroute incoming requests to either the legacy system or to the new modern transactional components.

At first the proxy send most of its traffic to the old soap endpoints. As features are extracted and rebuilt as microservices, the Traffic Redirector gradually directs traffic to the new REST interfaces. The team is able to deliver value gradually while ensuring the legacy system remains functional, thanks to this phased approach. The last phase takes place when the legacy feature is already replaced, and then the SOAP services and proxy layer can be fully shut down.

5.2. The Leave and Layer Pattern

In situations where direct refactoring of the legacy system is too complicated or costly, we have Leave and Layer. The approach involves leaving the legacy core untouched and layering a modern event-driven architecture on top. The organization can produce events to an event bus (for example, Amazon EventBridge) via database triggers or Change Data Capture (CDC) on the legacy application.

Consequently, new cloud-native functionality can be engineered as extensions to the application, without risk of dislodging existing application code. The approach suits organizations needing quick, low-risk outcomes, or those working with unfamiliar legacy technology where significant refactoring is not an option.

5.3. Wrapper and Adapter Methodologies

A common intermediate step in migration involves the use of wrappers and adapters. A wrapping methodology makes the interactive functionalities of legacy systems accessible as web services by creating a RESTful façade over the existing SOAP endpoints. This translation layer converts incoming JSON requests into the required SOAP XML format, allowing modern clients to interact with the system using standard RESTful principles while the backend logic remains unchanged. While this does not remove the performance overhead of SOAP, it facilitates the modernization of the integration surface and allows for the gradual decoupling of client applications from the legacy protocol.

6. Core Challenges in Protocol and Data Mapping

The technical execution of a SOAP-to-REST migration involves a fundamental re-engineering of how data and operations are represented. This process is far more complex than a simple format conversion from XML to JSON.

6.1. From Verbs to Nouns: Resource Mapping

SOAP services are inherently operation-centric, with WSDLs defining verbs such as GetOrder History, Create Customer, or Calculate Tax. REST, however, is resource-centric, requiring these operations to be mapped to nouns and HTTP verbs. For instance, the SOAP operation Lookup City with a zip code parameter should be redesigned as a GET request to the /cities/{zipcode} resource. This transformation requires a deep understanding of the underlying domain model to identify the core resources and their relationships.

6.2. XML to JSON: Structural and Semantic Nuances

Though JSON is the most preferred format for APIs, since it is much simpler and smaller in size, many things can go wrong while switching from XML. The structure of XML supports mixed content, namespacing, and hierarchical complexity which do not easily translate to JSON's key value and array syntax. For instance, it's natural to represent a paragraph of text with embedded markup in XML but messy with an array in JSON.

Organizations should set rules about handling xml attributes and namespaces during migration. Whilst automated "no-code" conversion tools can speed up the process, large manual validation is often still necessary. We often see the 'silent errors' such as missing rows and distortion of numerals (diff. rounding) when shifting data from legacy denormalized schemas to cloud solutions.

6.3. Error Handling and Fault Mapping

SOAP and REST handle errors in fundamentally different ways. SOAP relies on standardized XML-formatted fault messages, where each message contains a fault block describing the error. REST APIs, by contrast, utilize HTTP status codes (e.g., 400 for client errors, 401 for unauthorized, 500 for server failures) paired with a structured JSON response body to communicate the nature of the error.

A successful migration requires the creation of error code mapping rules to ensure that legacy SOAP faults are translated into the most appropriate HTTP status codes. This ensures that modern clients can react correctly to failures and that traceability is maintained across the migration. Without robust error mapping and logging, a single unhandled error during the transition can cause significant downtime and frustrate development teams.

7. Data Consistency and Reliability in Distributed Systems

Maintaining data consistency is perhaps the most significant hurdle when migrating from the centralized, ACID-compliant world of SOAP integrations to the decentralized world of microservices and EDA.

7.1. The Dual-Write Problem

A common issue in distributed architectures is the "dual-write" problem, which occurs when a service must perform two actions: update its local database and notify other systems of the change via an event. If the database update succeeds but the event publication fails, or vice versa, the system falls into an inconsistent state.

7.2. The Transactional Outbox Pattern

The dual-write problem is primarily addressed through the Transactional Outbox architectural pattern. The service writes the business data and the event payload into its own

database in one atomic transaction instead of calling the message broker directly. The event is saved in an outbox table. An outbox processor is typically a dedicated background process that reads these events from the outbox table and publishes them on the messaging broker.

This pattern relies on the transaction guarantees of the local database. An event will never be published unless the business data is persisted. This approach is especially applicable for migrating legacy SOAP services that utilized distributed transactions (XA) as a way to ultimately achieve consistency without incurring the cost of complex distributed locking mechanisms.

7.3. Change Data Capture (CDC) and Eventual Consistency

Change Data Capture (CDC) is essentially a more sophisticated version of the outbox pattern. CDC tools like Debezium capture changes in real-time using the transaction log instead of a background worker polling the database. This approach lessens the burden on the database while ensuring synchronization between the legacy system and the event-driven layer in almost real time.

A change in organizational thinking is key for eventual consistency to happen. It's Anticipated That Developers Design For Idempotency. That Is, Receiving The Same Event More Than Once (As Is Common With "At-Least-Once" Delivery) Must Not Cause Duplicate Actions. This commonly involves embedding unique event IDs and checking them against a distributed cache (Redis) before processing.

8. Security Migration: From WS-Security to OAuth 2.0 and JWT

Securing enterprise integrations is a multifaceted challenge that evolves significantly during a migration. Legacy SOAP services rely heavily on WS-Security, which provides message-level security controls. This model is valuable for long-running B2B workflows where messages may pass through multiple intermediaries, as the security is embedded in the message itself rather than just the transport channel.

8.1. Token-Based Authentication and Authorization

Modern architectures that are RESTful and event-driven enjoy a security model that is token-based and founded on OAuth 2.0 and JSON Web Tokens (JWT). The simplest description of OAuth 2.0 could be that is an authorization framework that enables applications to obtain limited access to user accounts. JWTs are small and self-contained objects used for securely transmitting information to and from parties. This also allows services to certify identity and authorization independently of a database.

Token-based authentication makes the system more scalable and does not load the server as the state is sent on the token instead of saving on the server. Nevertheless, if encryption and signature verification are not enforced correctly, JWT implementation can allow outsiders to modify tokens or misuse their privileges. To combat this, organizations must use short-lived access tokens, refresh tokens and apply robust revocation strategies during secure lifecycle.

8.2. Bridging the Security Gap

During the migration phase, enterprises often face the challenge of bridging WS-Security and OAuth 2.0. This is frequently addressed by placing an API gateway at the edge of the network. The gateway accepts modern OAuth tokens from clients, validates them, and then performs a token exchange or credential mapping to call the backend legacy SOAP services using the required WS-Security headers.

This proxy-based security layer enables the implementation of Zero Trust Architecture (ZTA) even for legacy systems. By enforcing continuous verification, micro-segmentation, and least-privilege access, organizations can protect vulnerable legacy infrastructure from lateral movement by attackers while gradually transitioning to modern security postures.

9. Performance Engineering and Observability

The performance benefits of moving away from SOAP are often the primary driver for migration. Research indicates that RESTful APIs using JSON can result in response time improvements of 40-50% compared to equivalent SOAP services. However, achieving these gains requires careful performance engineering across the entire stack.

9.1. Benchmarking and Infrastructure Selection

Choosing a suitable messaging framework is essential for event-driven architecture. An examination of message brokers shows that Apache Kafka offers the highest throughput, with benchmarks of 1.2M messages/sec, at the cost of operational complexity. Compared to spark, a pulsar offers better multi-tenancy and geo-replication, at the cost of slightly lower performance. Elsewhere, serverless solutions allow elastic scaling for bursty workloads, but they could lead to cold start latency.

Software programming language of choice affects the performance of the software. Compiled languages (Java, C#) tend to be more robust and responsive than interpreted languages (Python) under sustained load, although the latter may be favoured for use in rapid prototyping.

9.2. The Need for Distributed Observability

As systems transition from a few monolithic SOAP services to a mesh of RESTful and event-driven microservices, observability becomes paramount. Organizations must move beyond basic logging to implement comprehensive monitoring frameworks that include distributed tracing and real-time telemetry. This is essential for diagnosing performance bottlenecks and understanding the flow of events across complex, asynchronous pipelines.

10. Organizational Readiness and Cultural Shift

Technical re-architecture is only one component of a successful migration; the organizational and human factors are equally critical. Qualitative research into enterprise migrations highlights that stakeholder concerns often center on security risks, performance reliability, and the potential for business disruption during the transition.

10.1. Addressing the Skill Gap

Migration to REST and EDA requires a fundamental change in how teams develop, deploy, and monitor software. The shift to a microservices-based architecture necessitates a deep understanding of containerization (Docker/Kubernetes), DevOps practices, and CI/CD pipelines. Many organizations struggle with a lack of personnel skilled in these areas, which can lead to project delays or suboptimal architectural choices.

10.2. Leadership and Governance

A collaborative and innovative organizational culture is essential for overcoming the challenges of microservices adoption. Leadership commitment is the most crucial criterion for readiness, as migration projects often require significant upfront investment and a long-term perspective to achieve a positive ROI. Furthermore, enterprises must establish formal API governance frameworks to manage the lifecycle of their modern APIs, ensuring consistent naming, documentation (e.g., OpenAPI), and versioning policies.

11. Conclusion

The transition from SOAP-based enterprise integration to REST and Event-driven APIs is a journey from the contract-driven world of the 1990s into the ‘real-time’ relationships of the 21st century. The new channel of communication is not merely a replacement of a temporary nature but also the upheaval of protocol designs that scale, secure and connect networks that manage multiple types of data.

This research has shown that modernization is full of challenges including complex resources mapping, asynchronous data consistency, and security standards evolution. Nevertheless, the modern world is rife with risks. Yet, by adopting gradual patterns like the Strangler Fig, enacting strong reliability patterns like the Transactional Outbox, and creating security layers based on proxies, we can gain from it.

Successful progress in these fronts requires a holistic understanding of the technical and organizational hard and soft readiness. With enterprises becoming more cloud-native and requiring real-time processing, the ability to combine old reliability with new agility will remain an important differentiator in fierce competition. The gradual evolution of integration is an ongoing process. The steps highlighted here can serve as a solid base for organisations to proceed with this major construction upgrade.

References

1. International Journal on Science and Technology. "The Evolution of APIs from SOAP to REST and GraphQL." 2025.
2. ResearchGate. "Modernizing Enterprise Web Services: RESTful APIs in a Cloud Context." 2025.
3. Venturelli, I. "The Evolution of System Integration: From SOAP to REST to Event-Driven Architectures." 2025.
4. Upadhyaya, B., Zou, Y., et al. "Migration of SOAP-based services to RESTful services." Symposium on Web Systems, 2011.
5. Goswami, V. (2026). A Comparative Study of Synchronous vs Asynchronous API Orchestration in MuleSoft-Led Enterprise Modernization. International Journal of Emerging Trends in Computer Science and Information Technology, 7(1), 101-104. https://doi.org/10.63282/3050-9246.IJETCSIT-V7I1P114
6. ResearchGate. "SOAP-based vs. RESTful web services: A case study for multimedia conferencing."
7. StepSoftware. "Modernizing Legacy Systems with the Strangler Fig Pattern."
8. Microsoft Azure Architecture Center. "Strangler Fig Pattern."
9. Future Processing. "The Strangler Fig Pattern for Legacy System Modernization."
10. Waehner, K. "Replacing Legacy Systems One Step at a Time with Data Streaming: The Strangler Fig Approach." 2025.
11. Curotec. "Modernizing a Legacy Application Using the Strangler Fig Pattern."
12. Google Cloud Documentation. "Event-Driven Architecture and Pub/Sub."
13. International Journal of Computer Science. "Event-Driven Architecture Patterns using Message Brokers in.NET."
14. ResearchGate. "Harnessing the Power of Event-Driven Architecture for Scalable Microservices." 2025.
15. arXiv. "Evaluation of High-Throughput Messaging Frameworks: Kafka, Pulsar, and RabbitMQ." 2025.
16. arXiv. "Building Secure Token-Based API Systems: OAuth 2.0 and JWT." 2025.
17. International Journal of Multidisciplinary Research. "Role of OAuth2 and JWT in Securing Distributed API Gateways." 2022.
18. International Journal of Innovative Research in Science. "Custom Authentication Frameworks for Mitigating JWT Vulnerabilities." 2025.
19. ResearchGate. "Comparative Study: SOAP vs. REST for Infra-Centric Data Transfer." 2025.
20. AWS Migration and Modernization. "Modernizing Legacy Applications with EDA: The Leave and Layer Pattern."
21. JETIR. "Strategies for Microservices Adoption and Migration from Legacy Systems." 2025.
22. Superblocks. "SOAP vs. REST: A Comprehensive Comparison."
23. DreamFactory. "Error Handling in SOAP to REST: Best Practices."

24. NoCodeAPI. "XML to JSON: 5 Steps to Modernize Legacy Systems."
25. International Journal of Multidisciplinary Research and Growth Evaluation. "Applying OAuth2 and JWT Protocols in Securing Distributed API Gateways." 2022.
26. Scott Logic. "Solving Data Consistency in Distributed Systems with the Transactional Outbox." 2025.
27. Dev.to. "The Transactional Outbox Pattern: Reliable Messaging in Distributed Systems."
28. Carr, J. "Transactional Outbox Pattern: Managing the Dual-Write Problem." 2026.
29. ResearchGate. "Security Architecture and Zero-Trust Models for Legacy Modernization." 2025.
30. International Journal of Science and Research. "Zero Trust Security Architecture for Legacy Systems." 2025.
31. ResearchGate. "Migrating Enterprise Legacy Source Code to Microservices: Statefulness and Data Consistency."
32. IEEE Xplore. "Readiness Criteria for Migrating from Monolithic to Microservices Architecture." 2024.