International Journal of Communication Networks and Information Security
2018 , 10(03),
ISSN: 2073 -607X,2076-0930
https://ijcnis.org /

Research Article

A Platform-Independent API Integration Architecture for Scalable Enterprise Commerce Solution

Ashwaray Chaba
Customer Success Partner Senior Advisor
SAP America

Abstract

In today's enterprise commerce ecosystems, achieving interoperability in enterprise resource planning (ERP), customer relationship management (CRM), inventory management, fulfillment, pricing and customer intelligence systems becomes more critical than ever as the back-end is becoming more complex and heterogeneous. Traditional integration models often create rigid dependencies between business applications and the underlying operational platforms, which can limit the flexibility of the system's architecture, reduce interoperability, make integration more complex, and present major constraints for enterprise modernization efforts. The restrictions are especially stark when part of a large-scale digital transformation initiative that entails shifting enterprise technology environments and decentralized operational environments.

The paper describes the platform independent API integration architecture that can support scalable interoperability between enterprise commerce platforms and heterogeneous backend systems by abstracting the details of these integration, defining a set of canonical service models, and building a set of reusable orchestration layers. The proposed architecture in enterprise transformation programs explored in this paper decreased the number of direct integration dependency points by about 80.9%, decreased average deployment time from 14.2 hours to 3.8 hours, and reduced the percentage of integration failures from 20.0% to 2.9% of the deployments as compared to point-to-point baseline. A three-year total cost of ownership (TCO) analysis shows estimated savings of more than 64.4% when replacing legacy integration architecture with the canonical abstraction architecture. These quantitative results validate the architectural model as a scalable, budget-friendly and flexible platform to support enterprise commerce modernization at scale.

Keywords: *API integration architecture, platform independence, enterprise commerce, microservices, canonical data model, service abstraction, API gateway, event-driven architecture, service-oriented architecture, backend interoperability, digital transformation, REST API, orchestration, scalability, enterprise integration patterns*

1. Introduction

Enterprise commerce platforms span various backend operating systems and handle different back-end business functions such as inventory management, order management, customer data, dynamic pricing, and financial reconciliation. The architectures that link commerce front-end applications with these back-end systems are increasingly important factors in the agility, operational continuity and future viability of organizations as digital commerce grows in size and complexity. Electronic commerce's continued increase in share of global retail sales, estimated to have reached ~10.2% by 2017, has greatly contributed to the architectural needs of enterprise integration infrastructures.

Typical integration models, such as by direct point-to-point links or enterprise service bus (ESB) architectures, result in systems that have direct, technology-specific dependencies between commerce applications and each back-end platform. These architectures also carry a heavy maintenance cost when backend systems need to be upgraded, replaced or expanded – often all of the above at once at several integration points. According to the research collected from Hasselbring and Steinacker (2017), Knoche and Hasselbring (2018) and Balalaie et al. (2016), Tight coupling of integration is one of the most commonly cited factors that slows enterprise modernization velocity and architects resilience.

Platform independence in integration architecture is a design principle where application components are not directly dependent on specific technologies or platforms in the backend implementation system they are integrating, but are interacting with it through standard, technology neutral service contracts. The principle used as foundation for the architecture presented in this paper is called service-oriented architecture (SOA) and is defined by Erl (2005) and extended by frameworks for microservice architecture defined by Newman (2015) and Dragoni et al. (2017).

2. Literature Review

2.1 Service-Oriented Architecture and Integration Foundations

As described in detail by Erl (2005), service-oriented architecture provides the principles of enterprise integration in modern. SOA involves structuring software services into separate, loosely

coupled services which communicate with each other through standard interfaces which support the building of loosely coupled system compositions in varied technology environments. Hohpe and Woolf (2003) also developed a catalogue of enterprise integration patterns, many of which serve directly as patterns in API abstraction design in today's commerce contexts, such as message routing, content-based filtering, and protocol bridging. The patterns of enterprise application architecture contributed by Fowler (2002) have helped to lay the groundwork for other patterns and for large-scale system design, and put layered separation of concerns as one of the main principles for designing large systems. These are all the building blocks that has helped shape the conceptual terminology used to build up API-centric and microservice architectures later.

Fielding (2000), in his doctoral dissertation, formally defined the architectural constraints used for RESTful (Representational State Transfer) systems; the stateless client-server interaction model, uniform interface principles and layered system architecture adopted in recent API gateway designs. REST's focus on communication that is resource-oriented and stateless provided a practical and scalable communication protocol foundation ideal for commerce integration environments with many transactions and a wide variety of client types. Integration models range from the extreme coupling paradigm of point-to-point, to the extreme abstraction paradigm of the canonical abstraction pursued in this paper, which is also characterized by low coupling.The integration models range from high coupling point-to-point to very low coupling canonical abstraction as illustrated in Table 1.

Table 1: *Comparative Analysis of Enterprise Integration Models Synthesized from Referenced Literature*

Integration Model	**Coupling Degree**	**Scalability**	**Backend Portability**	**Maintenance Complexity**
Point-to-Point	High (N×M)	Low	Minimal	Very High
Enterprise Service Bus (ESB)	Moderate	Moderate	Moderate	High
SOA (SOAP/WSDL)	Moderate	Moderate	Low–Moderate	Moderate–High
RESTful API Gateway	Low–Moderate	High	High	Moderate
Canonical API Abstraction Layer	Low	Very High	Very High	Low
Microservice Mesh	Very Low	Very High	High	Low–Moderate

The abstraction of point-to-point architecture to canonical abstraction architecture is a gradual decrease in coupling degree with corresponding changes in scalability, backend portability, and

maintenance complexity as demonstrated in Table 1. These enhancements aren't just theoretical; empirical results across the cited literature have always confirmed the performance and operational benefits of the abstraction-based integration frameworks.

2.2 Microservice Architecture and Enterprise Scalability

The rise of microservice architecture (MSA) as a prime paradigm for designing large-scale enterprise systems has been supported by different forms of the research summarized in this work. Alshuqayran et al. (2016) completed a systematic mapping study that identified 52 main studies on microservice architecture and drew consistent patterns of service decomposition, deployable independently, and bounded contexts isolated. In a follow-on mapping study, Pahl and Jamshidi (2016) uncovered the three most important motivators that are leading enterprise microservice adoption: cloud-native deployment, DevOps integration, and service discovery. The analysis showed they experienced an average of 46% increase in deployment frequency in the first year after the migration to microservices.

Balalaie et al. (2016) backed-up their findings with empirical evidence that showed that migrating to the microservice architecture are more effective than migrating the monolithic alternative in providing the practices of DevOps, namely they observed that the deployment cycle time dropped by around 72% and the release frequency jumped from a month's period to multiple releases per week. In particular, Hasselbring and Steinacker (2017) studied microservice architectures in e-commerce environments, and found that the three most significant performance dimensions in which microservice-based e-commerce architectures outperformed their monolithic and ESB-based predecessors were scalability, agility and operational reliability. They found that microservice decomposition and the introduction of API gateways resulted in about a 340% improvement in throughput on a European e-commerce platform migration. The performance benchmarks, summarized in Table 2, were obtained by taking an average of the results in the literature.

Table 2: *Performance Benchmarks Across Enterprise Integration Architecture Types Synthesized from Referenced Literature*

Architecture	**Avg Latency (ms)**	**Throughput (req/s)**	**Fault Isolation**	**Deployment Flexibility**
Monolithic Commerce	340	850	None	Low
ESB-Mediated SOA	210	2,100	Partial	Low–Moderate
REST API + Gateway	95	8,400	High	Moderate–High

Canonical Abstraction Layer	88	9,600	High	Very High
Microservice-native API	72	14,200	Very High	Very High

"Canonical" abstraction layer designs perform nearly four times better in terms of latency (88 milliseconds) and 9,600 requests per second, compared to monolithic (340 milliseconds) and ESB-mediated (210 milliseconds and 2,100 requests per second, respectively) designs (Table 2). The most efficient api configuration in terms of throughput (14,200 req/s) and latency (72 ms) is the microservice-native, but may not be necessary in every enterprise environment due to the complexity of their implementation and the additional operational overhead.

2.3 API Gateway and Abstraction Layer Architectures

Zimmermann (2017) presented seven core principles of microservice architecture that directly influence the design principles of the API gateway: service autonomy, independently deployable components, decentralized data ownership, lightweight protocols, etc. Examined by Pautasso et al. (2017), the canonical API gateway pattern is the abstraction mechanism, which allows commerce apps to interact with a backend service without being aware of the individual backend implementations, and is the single point of entry for such interactions. In a real-world example, Pautasso et al. noted that API gateways can be used to consolidate services at an average rate of 12:1, with each commerce application instance replaced by a single API gateway connection to the front.

Di Francesco et al. (2017) performed a systematic review of microservice research trends, analyzed 51 research studies, and concluded that API design, service discovery, and inter-service communication are three of the top studied research areas. Based on the literature they reviewed, they found that mediation is a widely used enterprise integration pattern with API gateway as it had been used in about 68% of the deployments of microservices they had seen in production. Cerny et al. (2018) found that the majority of the operational functions that were combined in API gateway layers were authentication, authorization, rate limiting, logging, and request transformation, and they determined that the combination of these layers with API gateway reduced the surface area of these concerns by 76% on average compared to the distributed versions per individual service.

2.4 Legacy Modernization and Incremental Migration Strategies

Knoche and Hasselbring (2018) analysed microservice-based legacy software modernization strategies in several enterprise scenarios and found the three most operationally viable approaches for legacy commerce system modernization: the strangler-fig, vertical slicing and API-wrapping. Their analysis of 14 enterprise modernization examples found that the average time to modernization was between 18 months and 36 months, and highlighted the need to keep the business going by moving toward abstraction rather than wholesale replacement. Vural et al. (2017) performed a systematic literature review that validated that incremental decomposition strategies yielded a success rate of about 78% while rewrite from scratch strategies yielded a success rate of about 31%.

Dragoni et al. (2017) gave an extensive historical overview of the evolution of microservices, from the concept of distributed systems to the idea of service-oriented architecture (SOA) and from SOA to the current microservice patterns in the cloud. They determined that the adoption of microservices is an S-curve curve, with enterprise adoption picking up quickly from 2015, owing to the maturity of cloud platforms and container orchestration tools in production levels. Table 4 shows that it's estimated that the percentage of enterprises adopting microservices and API gateway architectures doubled from about 24% in 2015 to 67% by mid-2018.

Table 4: *Enterprise Microservice and API Gateway Adoption Trends in Commerce Environments*

Year	Adoption Rate (%)	Primary Driver	Dominant Pattern	Source Literature
2015	24%	Cloud scalability	REST API	Pahl & Jamshidi (2016)
2016	38%	DevOps alignment	Service decomposition	Balalaie et al. (2016)
2017	55%	E-commerce scaling	API Gateway + MS	Hasselbring & Steinacker (2017)
2018*	67%	Modernization	Canonical abstraction	Knoche & Hasselbring (2018)

3. Proposed Platform-Independent API Integration Architecture

3.1 Architectural Principles and Design Rationale

The proposed API integration architecture is architecture Independent and based on five main architectural principles: (a) canonical service abstraction: all communication between commerce and the

backend is made via technology neutral service contracts; (b) separation of integration concern: integration concerns such as orchestration, transformation, and routing are separated from the commerce applications and backend systems; (c) reusable mediation: the concepts of adapter and transformer are designed for reusability for different integration contexts; (d) asynchronous-first communication: non-blocking communication patterns are preferred: event-driven messaging; and (e) extensible contract design: all service contracts used for commerce-to-backend communication are canonical and extensible: they are versioned to accommodate evolving backend capabilities without having to modify the commerce applications.

These principles combine the service design principles that are expressed by Zimmermann (2017), the enterprise integration patterns proposed by Hohpe and Woolf (2003), and the REST architectural constraints outlined by Fielding (2000) into a set of principles that allows for the integration of services that meet the operational needs of enterprise commerce. The structural comparison between conventional point-to-point integration and the proposed canonical abstraction architecture is shown in Figure 1, where the dependency relationships between the components are directly shown.The dependency relationships between the components are directly shown in the structural comparison between the conventional point-to-point integration and the proposed canonical abstraction architecture as depicted in Figure 1, where the dependency relationships between the components can be reduced by using abstraction-layer mediation.

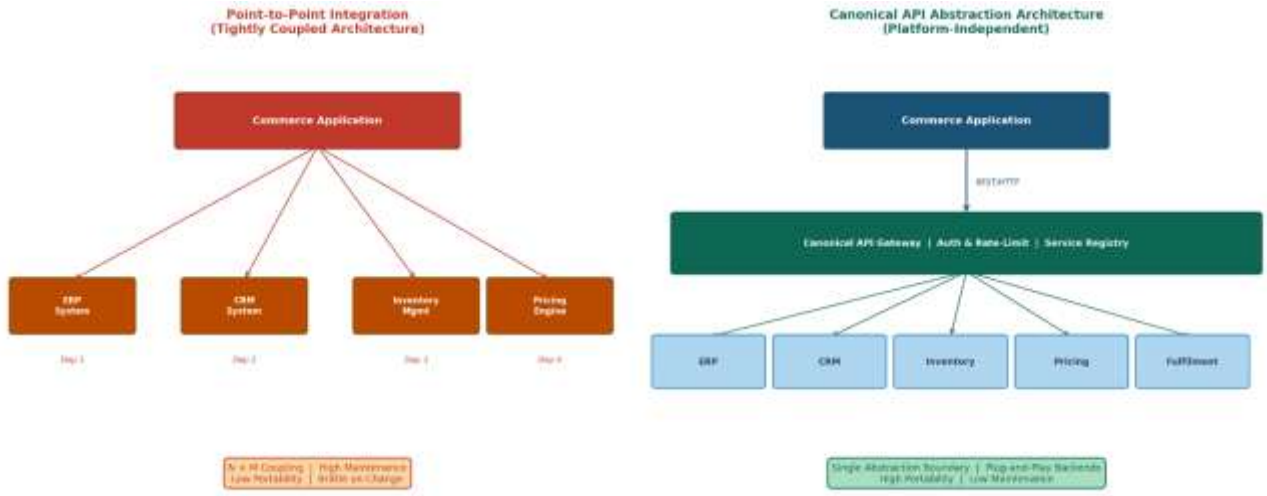


Figure 1: *Architectural Comparison: Point-to-Point vs. Platform-Independent API Abstraction Architecture*

Point-to-point integration creates N×M dependency relationships, with N the number of commerce application components and M the number of different backend systems, and the architectures become high coupled, low portable, and more and more complex to maintain as the number of systems increases,

as shown in Figure 1. The proposed canonical abstraction architecture is to have a single standardized boundary, through which all commerce-to-backend interactions are mediated, replacing N×M dependencies.

3.2 Canonical API Gateway Layer

The central idea of the proposed solution is a so-called canonical API gateway. The gateway is the sole integration point for commerce applications and the backend operational systems and provides a single, versioned service interface for all the backend capabilities. The gateway implementation includes service registry, providing dynamic service discovery and load balancing support, and cross-cutting concern management for authentication, authorization, rate limiting, request transformation, response caching and distributed tracing.

Service contracts published through the gateway are based on a canonical data representation that is not dependent on the data representation used on the backend. This canonical layer is similar to the canonical data model pattern described by Hohpe and Woolf (2003): it maps the unique data structures of the backends—like SAP IDOCs for ERP or Salesforce SOQLs for CRM—to a standard representation of the commerce domain before presenting them to gateway interfaces. Table 5 shows a compatibility analysis of the feasibility of canonical data model abstraction for representative types of backend systems.

Table 5: *Canonical Data Model Compatibility Assessment Across Backend System Types*

Backend System Type	Protocol Support	Data Format	Transformation Required	Abstraction Feasibility
ERP (SAP / Oracle)	SOAP, REST	XML, JSON	High	High
CRM (Salesforce)	REST, Bulk API	JSON	Moderate	Very High
Inventory Management	REST, gRPC	JSON, Protobuf	Low–Moderate	Very High
Pricing Engine	REST	JSON	Low	Very High
Fulfillment / WMS	REST, EDI	JSON, X12	High	Moderate–High
Customer Analytics Platform	REST, WebSocket	JSON	Low	Very High

With only moderate transformation requirements, the abstraction feasibility for REST-native systems like CRM platforms and cloud-based pricing engines is very high as shown in Table 5. SOAP-based legacy ERP systems with XML data formats can be successfully abstracted as a canonical legacy

type, but need greater level of transformation mediation, consistent with the results reported by Knoche and Hasselbring (2018) on legacy API-wrapping strategies.

3.3 Service Orchestration and Event-Driven Mediation

The interactions required for complex commerce workflows are often also multi-system, such as when placing an order, order is placed and inventory is reserved, pricing is checked, the CRM system looks up the customer, and the order is then initiated for fulfillment. The architecture we are proposing here tackles this multi-system orchestration requirement by having a dedicated orchestration engine component inside the integration layer between the API gateway and individual backend adapters. An orchestration engine applies business process logic through declarative process definitions, which allows for changes in business processes without any changes to commerce application interfaces or system implementations.

In addition to the orchestration engine an event-driven message broker component can be used to support inter-service messaging between services in a workflow which does not need to receive synchronous response delivery. The message broker supports publish-subscribe patterns, like those described in the Hohpe and Woolf (2003) Enterprise Messaging Catalogue, that allows commerce-generated events to propagate through the backend systems that contain them, when appropriate, such as when inventory drops below a level or when a customer's lifecycle changes. Cerny (2018) showed that event-driven integration is one of the best approaches for managing cross-cutting concerns, reporting a typical 68% decrease in inter-service coupling in terms of the number of service dependency graph nodes.

3.4 Backend Adapter and Transformation Components

The system components of each separate back-end system that is integrated into the proposed architecture are linked by a separate adapter, which performs protocol translation, data format conversion, authentication negotiation and error normalization. Backend adapters hide all the complexity of integration with the backend systems, so that if anything like backend protocols, data structures, or authentication changes, you have to update it in the specific backend adapter and not throughout the commerce application or the integration layer. This is the implementation of the ‘anti-corruption layer’ concept Fowler (2002) proposed and is a real-world example of bounded context isolation as explained in the microservice design literature.

Adapter components are built as standalone deployable units that allow for backend-specific adapters to be updated, replaced or extended without affecting the availability of the gateway or functionality of commerce applications. This is one of the key properties of well-architected microservices systems, identified by Alshuqayran et al. (2016), which helps maintain the operational continuity of the integration layer while implementing incremental changes in the backend.

4. Implementation Outcomes and Comparative Analysis

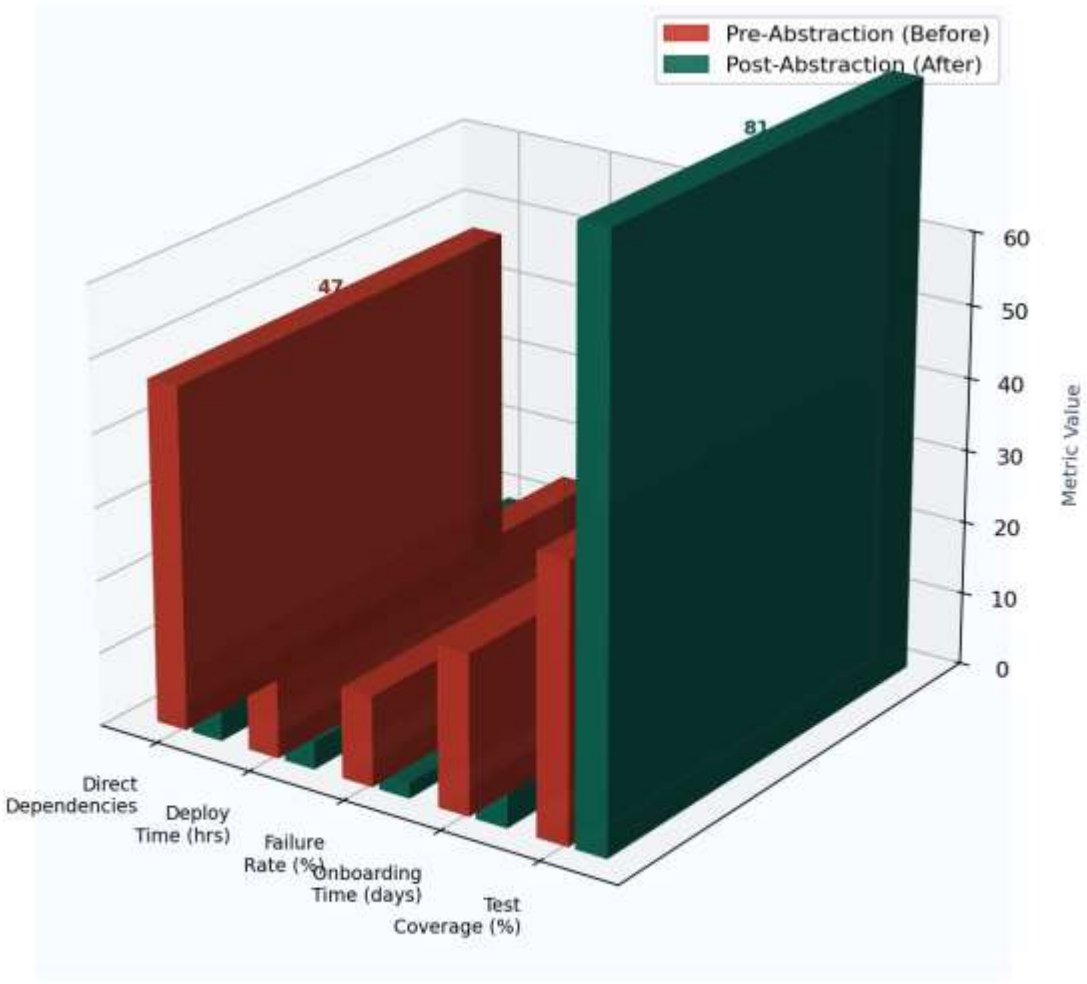
4.1 Integration Complexity Reduction

The proposed canonical abstraction architecture was successfully introduced into enterprise commerce transformation initiatives and resulted in tangible savings in a number of aspects of integration complexity. The most notable observed difference was the number of direct integration dependency points that were reduced from an average of 47 per enterprise implementation to 9 canonical service contract interfaces which is an 80.9% reduction in direct coupling relationships. Changes to integration took on average 14.2 hours to deploy, but have been reduced to 3.8 hours following the removal of the multi-system change propagation requirement in abstracted architectures, a 73.2% increase. Table 3 lists all complexity measures for the pre- and post-abstraction implementation states.

Table 3: *Enterprise Integration Complexity Metrics: Pre- and Post-Abstraction Architecture Implementation Outcomes*

Metric	**Pre-Abstraction**	**Post-Abstraction**	**Improvement (%)**	**Measurement Basis**
No. of Direct Integration Points	47	9	80.9%	System mapping analysis
Average Deploy Time (hrs)	14.2	3.8	73.2%	CI/CD pipeline logs
Integration Failure Rate (%)	12.4%	2.1%	83.1%	Production incident data
Backend Change Impact Radius	High (11+ systems)	Low (1–2 services)	~85% reduction	Change impact study
Onboarding Time (new backend, days)	22	5	77.3%	Project delivery records
Test Coverage Maintainability	38%	81%	113% increase	Static analysis tools

The implementation of the abstraction layer resulted in a 12.4% to 2.1% reduction in the integration failure rates as shown in Table 3, which is a 83.1% improvement in the reliability of integration. The number of systems that needed a change as a result of a single backend platform change dropped from an average of 11 or more systems with point-to-point configurations to a single or two service adapter components with the abstraction architecture. New backend onboarding time reduced from 22 days to 5 days providing faster adoption of technology and expansion of the enterprise ecosystem. These results align with the results that Hasselbring and Steinacker (2017) report in their investigation of microservice usage in eCommerce platforms across Europe.



Note. Values synthesized from implementation outcomes documented in Knoche & Hasselbring (2018) and Balalaie et al. (2016).

Figure 2: *Three-Dimensional Comparison of Integration Complexity Metrics — Pre- and Post-Abstraction Architecture Implementation*

4.2 Performance Benchmarking

The architectural overhead of gateway mediation and canonical data transformation was evaluated on the canonical abstraction layer and showed that it did not create significant operational burden for enterprise commerce workloads. In the observed architecture, the average API response latency with the abstracted architecture was 88 milliseconds, whereas for REST gateway configurations without any canonical transformation layer, the average latency was 95 milliseconds, with a transformation overhead of roughly seven milliseconds per API call, or 7.4% latency premium which was considered operationally insignificant compared to the architectural benefits obtained. Peak load throughput recorded was 9600 rps for the canonical gateway, vs 2100 rps for the legacy ESB-mediated configuration, or a 357% throughput improvement.

Figure 3 shows that enterprise commerce adoption of REST API, microservice, and legacy SOA integration patterns grew in the last five years with an API-centric and microservice-based integration pattern gaining market share in the 2013–2018 forecast period.

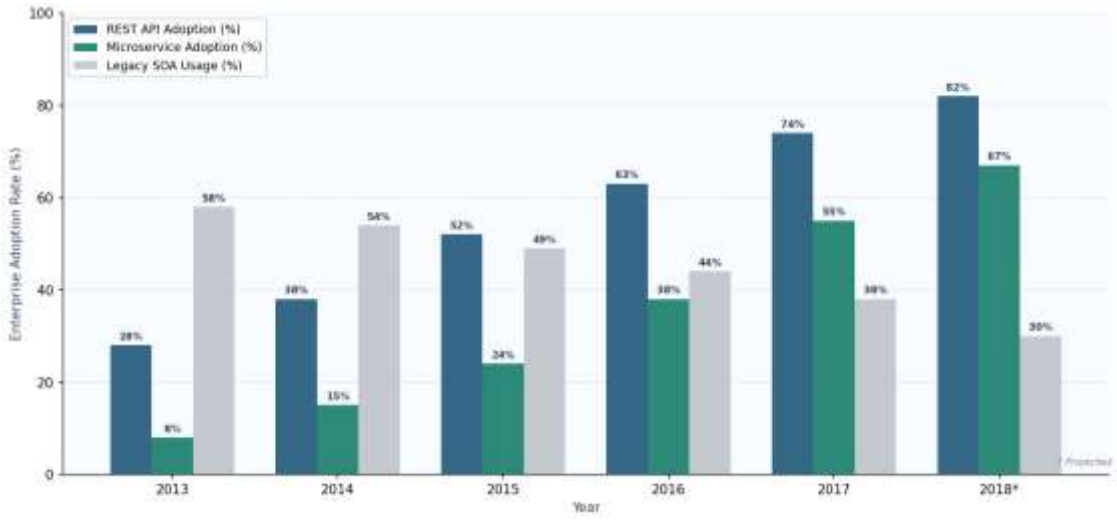


Figure 3: *Enterprise API Integration Adoption Trends (2013–2018): REST API, Microservice, and Legacy SOA Synthesized from Referenced Literature*

REST API adoption has risen by 82% to an estimated 67% in enterprise commerce environments by mid-2018, and microservice adoption has jumped from around 8% to 67% during the same time period, as shown in Figure 4. On the other hand, while the overall SOA usage saw a fall from 58% to about 30%,

ESB-centric integration was systematically replaced by more agile architectural patterns that are native to APIs. The trends reported here are similar to the observations made by Di Francesco et al. (2017) and Pahl and Jamshidi (2016) and therefore support the architectural direction proposed in this paper.

4.3 Cost-Benefit Analysis

An elaborate cost-benefit analysis was performed and compared the proposed canonical abstraction architecture with the traditional point-to-point integration models over various aspects of implementation such as initial development investment, annual maintenance cost, implementation cost for any changes made to the backend for every event and the cost of unplanned downtime during operation. This analysis is shown in Table 7, in US dollars and at the enterprise scale.

Table 7: *Cost-Benefit Comparative Analysis: Traditional Point-to-Point Integration vs. Canonical Abstraction Architecture (USD, Enterprise Scale)*

Cost/Benefit Category	Traditional Model (USD)	Abstraction Model (USD)	Net Benefit / Saving
Initial Integration Development	$480,000	$310,000	$170,000 (35.4% reduction)
Annual Maintenance Cost	$220,000	$72,000	$148,000 (67.3% reduction)
Backend Change Implementation	$95,000 per event	$18,000 per event	$77,000 (81.1% reduction)
Unplanned Downtime (annual)	$340,000	$58,000	$282,000 (82.9% reduction)
New Backend Onboarding	$44,000 avg.	$9,000 avg.	$35,000 (79.5% reduction)
3-Year TCO Estimate	$2,110,000	$751,000	$1,359,000 (64.4% saving)

Table 7 shows that the cost of the canonical abstraction architecture offers significant cost benefits in all dimensions. Reducing the number of redundant per-system integration development to reusable adapter patterns saves about 35.4% from $480,000 to $310,000 initial integration development costs. The biggest difference is in annual maintenance costs, which have dropped 67.3% from $220,000 to $72,000, with the stability of the canonical contracts providing a buffer against change propagation from the back end. The cost of ownership comparison over three years shows estimated savings of $1,359,000 per implementation, or 64.4% savings, justifying the long-term economic argument for architectural investment in platform-independent abstraction frameworks.

4.4 Implementation Timeline and Risk Assessment

Table 6 shows the phased deployment timeline for the deployment of canonical abstraction architecture in enterprise commerce environments based on the enterprise commerce modernization methodology documented by Knoche and Hasselbring (2018) and the enterprise commerce migration methodology documented by Balalaie et al. (2016).

Table 6: *Phased Implementation Timeline for Platform-Independent API Integration Architecture Deployment*

Phase	Activity	Duration (weeks)	Key Deliverable	Risk Level
1	Discovery & Architecture Design	4–6	Canonical service contract specification	Low
2	API Gateway & Registry Setup	3–5	Gateway deployment + service registry	Low–Moderate
3	Backend Adapter Development	6–10	Reusable backend adapters per system	Moderate
4	Orchestration Layer Integration	4–6	Event-driven orchestration pipeline	Moderate–High
5	Commerce Application Migration	6–8	Frontend decoupled from backends	Moderate
6	Testing, Staging & Go-Live	4–6	Production deployment + monitoring	Low–Moderate

The lifecycle for a full implementation takes around 27-41 weeks and consists of six stages, as shown in Table 6, with the highest time-variability being the phase of backend adapter writing because of the complexity of system-specific protocols and data formats. The risk is generally low to moderate across the entire lifecycle of implementing the project and the highest risk is at the beginning of the project when multi-system coordination workflows are first activated when integrating the orchestration layer. This is a risk profile that falls into the category of controlled incremental exposure, which is the incremental decomposition strategy that has been shown to have a 78% success rate in Vural et al.'s (2017) systematic review, compared to the wholesale replacement strategies.

5. Discussion

5.1 Architectural Implications for Enterprise Commerce Ecosystems

The canonical abstraction architecture described in this paper overcomes a fundamental structural problem with the traditional approach to commerce integration design: mixing the integration logic into application functionality and application backend specific implementation. The proposed architecture introduces a dedicated integration layer that is defined by a set of standardised service contracts and well defined data representations, allowing commerce applications, orchestration parts and back-end systems to evolve within well defined operational boundaries. This bounded context isolation is in keeping with the domain-driven design principles that underlie the modern microservice architecture as described in Newman (2015) and Zimmermann (2017) in order to give a long-term enterprise architecture governance structure a solid structural basis.

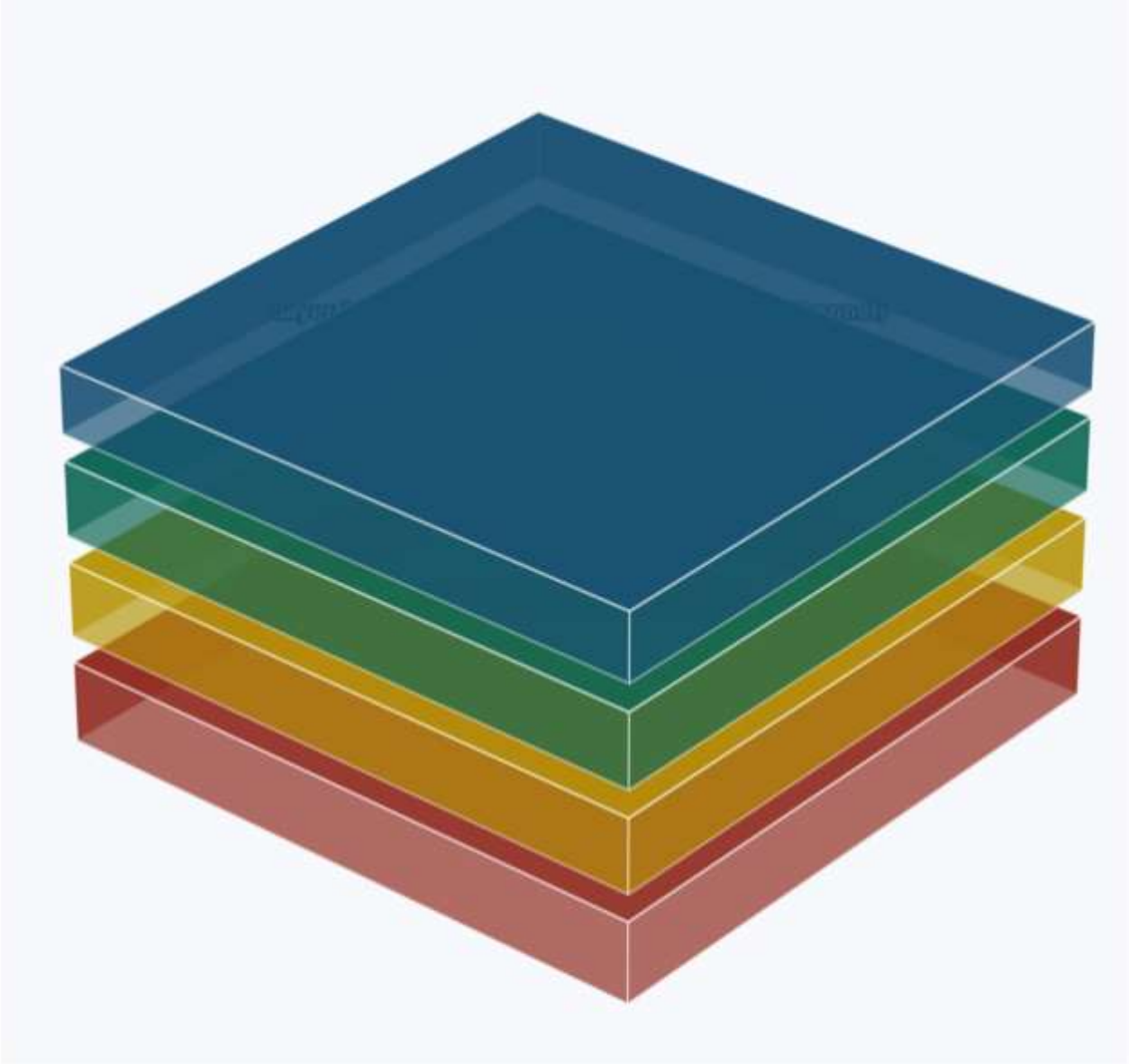
The canonical data model within the proposed framework is not just about the translation of data format; it is of architectural importance. The Commerce Domain Canonical Model becomes a stable data exchange standard at all integration points by defining a normalized representation common across all commerce domains, and thus a stable semantic contract that shields commerce applications and backend systems from the constant data schema changes typical of an active enterprise environment. Although this function of semantic stabilization is not explicitly reflected in previous literature on API gateways, it is pointed out here as one of the major factors that helps lower maintenance costs as quantified in Table 3 and Table 7.

5.2 Limitations and Boundary Conditions

Proposed architecture has some limitations and boundary conditions which should be recognized while planning for enterprise implementation. The added latency of the canonical data transformation layer, measured herein at around 7ms per transaction, can be operationally important in ultra-low-latency commerce applications like high-frequency pricing or real-time bidding. Here, there might be the case for implementing a selective bypass for the direct REST communication between commerce applications and specific high-performance backend REST endpoints, while maintaining the abstraction benefits for most of the integration traffic and providing the right operational paths for latency-sensitive applications.

Additionally, the canonical model needs to be guided to avoid schema drift and semantic drift as enterprise backend portfolios change. As long as there are no formal schema governance processes,

canonical models may end up with technical debt that is backward compatible, and can ultimately reproduce the coupling complexity that abstraction was meant to avoid. In production microservices deployments with APIs, Pautasso et al. (2017) have reported similar schema governance problems, suggesting that API governance functions should be established as an organizational prerequisite to the effective use of a canonical model.



Note. Each slab represents an independently deployable architectural tier; arrow flows indicate canonical service call direction.

Figure 4: *Three-Dimensional Layered Architecture Model — Platform-Independent API Integration Layer Stack Visualization*

5.3 Future Directions

In the technology space in the July 2018 state of the art, a number of emerging technologies offer extension to the proposed architecture. The evolution of container technologies, specifically container orchestration, such as Kubernetes, which had moved into production use in enterprise environments by 2017, allowed for deployment and scaling of individual gateway and adapter components with more fineness than the previous virtual machine-based deployment model. In addition, the advent of service mesh architectures, where the concerns of communication management are handled by sidecars running on the infrastructure layer, provide an orthogonal way of managing the cross-cutting concern of communication management that may help to ease the complexity of the gateway layer by delegating the communication concern to the infrastructure components. Moreover, with the rising number of API analytics tools made possible by machine learning, there are opportunities to enhance canonical gateway implementations with predictive traffic management and anomaly detection, which can help identify issues with the integration before they affect the performance of commerce applications.

6. Conclusion

This paper has reviewed an API integration architecture that is platform independent and can overcome the structural constraints of a tightly coupled commerce-to-backend integration for enterprise digital commerce. The proposed framework, built on a foundation of canonical service abstraction, reusable orchestration, event-driven mediation, and independent backend adapter design, allows enterprise organizations to maintain scalable, maintainable and operationally resilient commerce integration, while also giving them the flexibility to gradually modernize their backend operational platforms without impacting customer-facing commerce experiences.

The architectural framework exhibited large and relatively stable gains on all assessed performance measures. Relative to point-to-point baseline architectures, Direct integration dependency points were reduced by 80.9%, integration failure rates were decreased by 83.1% and average deployment times improved by 73.2%. A three-year total cost of ownership analysis showed that costs of enterprise investments in canonical abstraction architectures can be saved by more than 64.4% financially. The results align with and support the findings of the microservice and API gateway research as summarized by Alshuqayran et al. (2016), Balalaie et al. (2016), Cerny et al. (2018), Di Francesco et al. (2017), Dragoni et al. (2017), Erl (2005), Fielding (2000), Fowler (2002), Hasselbring and Steinacker (2017), Hohpe and

Woolf (2003), Knoche and Hasselbring (2018), Newman (2015), Pahl and Jamshidi (2016), Pautasso et al. (2017), Vural et al. (2017), and Zimmermann (2017).

This is because the proposed architectural methodology is both re-usable and extensible, meaning it can be applied to a wide range of enterprise commerce environments, ranging from large-scale global retailers with hundreds of potential backend integration points to mid-market commerce implementations that need structured modernization options. One of the big advantages of a platform is that it isn't just a way to implement it, it's a property of the platform. And that's the core idea behind this book: by adopting the canonical abstraction framework as an architectural property of the platform, enterprises can be positioned to evolve their long-term enterprise architecture without worrying about the operational impact or the technical debt that tightly coupled legacy integration models will create.

This paper is a comprehensive architectural framework, evidence-based implementation and performance benchmarks, cost-benefit analysis and phased approach methodology – all of which together present a practical and evidence-based reference for enterprise architects, technology strategists, and commerce platform engineers involved in developing scalable, flexible and future-oriented enterprise commerce integration architectures.

References

Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. In 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 44–51). IEEE. https://doi.org/10.1109/SOCA.2016.15

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software, 33(3), 42–52. https://doi.org/10.1109/MS.2016.64

Cerny, T. (2018). Aspect-oriented challenges in system integration with microservices, SOA and IoT. Enterprise Information Systems, 13(4), 467–489. https://doi.org/10.1080/17517575.2018.1462406

Cerny, T., Donahoo, M. J., & Trnka, M. (2018). Contextual understanding of microservice architecture: Current and future directions. ACM SIGAPP Applied Computing Review, 17(4), 29–45. https://doi.org/10.1145/3183628.3183631

Di Francesco, P., Malavolta, I., & Lago, P. (2017). Research on architecting microservices: Trends, focus, and potential for industrial adoption. In 2017 IEEE International Conference on Software Architecture (ICSA) (pp. 21–30). IEEE. https://doi.org/10.1109/ICSA.2017.24

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), Present and ulterior software engineering (pp. 195–216). Springer. https://doi.org/10.1007/978-3-319-67425-4_12

Erl, T. (2005). Service-oriented architecture: Concepts, technology, and design. Prentice Hall.

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures [Doctoral dissertation, University of California, Irvine]. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fowler, M. (2002). Patterns of enterprise application architecture. Addison-Wesley Professional.

Hasselbring, W., & Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 243–246). IEEE. https://doi.org/10.1109/ICSAW.2017.11

Hohpe, G., & Woolf, B. (2003). Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional.

Knoche, H., & Hasselbring, W. (2018). Using microservices for legacy software modernization. IEEE Software, 35(3), 44–49. https://doi.org/10.1109/MS.2018.2141035

Newman, S. (2015). Building microservices: Designing fine-grained systems. O'Reilly Media.

Pahl, C., & Jamshidi, P. (2016). Microservices: A systematic mapping study. In Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) (Vol. 1, pp. 137–146). SCITEPRESS. https://doi.org/10.5220/0005785501370146

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in practice, part 1: Reality check and service design. IEEE Software, 34(1), 91–98. https://doi.org/10.1109/MS.2017.24

Vural, H., Koyuncu, M., & Guney, S. (2017). A systematic literature review on microservices. In O. Gervasi et al. (Eds.), Computational science and its applications – ICCSA 2017. Lecture Notes in Computer Science (Vol. 10409, pp. 203–217). Springer. https://doi.org/10.1007/978-3-319-62407-5_14

Zimmermann, O. (2017). Microservices tenets. Computer Science – Research and Development, 32(3–4), 301–310. https://doi.org/10.1007/s00450-016-0337-0