Architecture for Extensible Multi-Carrier API Adapters in E-Commerce

From Integration to KPI-Based Evaluation

Bachelor Exposé

Submitted to
IMC University of Applied Sciences Krems

University of
Applied Sciences

Bachelor Programme Informatics

by

David Fodor BSc

for the award of academic degree
Bachelor of Science in Engineering (BSc)

under the supervision of
Rubén Ruiz, Torrubiano, Prof. (FH) Dr.

Submitted on January 15, 2026

DECLARATION OF HONOUR

I declare on my word of honour that I have written this Bachelor Exposé on my own and that I have not used any sources or resources other than stated and that I have marked those passages and/or ideas that were either verbally or textually extracted from sources. This also applies to drawings, sketches, graphic representations as well as to sources from the internet. The Bachelor Exposé has not been submitted in this or similar form for assessment at any other domestic or foreign post-secondary educational institution and has not been published elsewhere. The present Bachelor Exposé complies with the version submitted electronically.

David Fodor BSc
January 15, 2026

Abstract

E-commerce merchants, especially SMEs, often face fragmented shipping workflows due to heterogeneous carrier APIs and limited end-to-end visibility into shipping performance. This bachelor thesis investigates how an extensible, multi-carrier integration layer can be designed to reduce coupling to individual carriers while enabling a KPI-driven analytics dashboard that supports both operational monitoring (near real-time) and strategic evaluation (batch reporting). Building on a literature review of service-oriented integration patterns (e.g., adapters and API gateways) and monitoring/analytics architectures (e.g., hybrid real-time and batch processing reference models), the work formulates design criteria and research questions for Shopickup, a prototype system integrating Shopify with selected carriers. The intended outcome is a validated architectural design and a functional prototype that demonstrates pluggable carrier adapters and a dashboard capable of producing actionable shipping KPIs for merchant decision-making. At the time of writing, the thesis is a work in progress: the related-work and research foundation are completed, and the methodology and implementation/evaluation phases are starting.

Keywords: API Integration, KPI, E-Commerce, Shipping

Table of Contents

List of Tables

List of Figures

List of Abbreviations

KPI	Key Performance Indicator — a measurable value used to evaluate the success of an activity in meeting strategic or operational objectives.
REST	Representational State Transfer — an architectural style for networked applications, commonly used for web APIs.
SME	Small and Medium-sized Enterprises — businesses with a limited number of employees and revenue.
SOA	Service-Oriented Architecture — a design pattern where services are provided to other components by application components, through a network.
OTDR	On Time Delivery Rate — a KPI measuring the percentage of shipments delivered on or before the promised delivery date.
BI	Business Intelligence — technologies and strategies used by enterprises for data analysis and management of business information.
SLA	Service Level Agreement — a formal contract that defines the level of service expected. In shipping, commonly max delivery time.
ETL	Extract, Transform, Load — a data integration process that involves extracting data from various sources, transforming it into a suitable format, and loading it into a target system.
DRY	Don't Repeat Yourself — a software development principle aimed at reducing repetition of information or code.

Chapter 1 INTRODUCTION

E-commerce has revolutionized the retail landscape, enabling businesses of all sizes to reach domestic and global markets with unprecedented ease, meanwhile consumers benefit from unparalleled convenience and choice. Consequently, online shopping has seen strong growth over the past two decades, a trend accelerated by technological advancements and shifting consumer behaviors. As per Capital One's 2025 report, online sales accounted for 20.5% of total retail sales worldwide [1] and has been steadily climbing over the past decade. Similar trends can be observed in various regional markets and their respective research reports, such as in Eurostat's EEA specific data showing that more and more stores are adopting online sales channels, with 23.83% of all enterprises having one in 2023, compared to just 17.21% ten years prior [2], with the share of total sales following suit. On the individual level, Eurostat surveyed that 94% of EU citizens used the internet in 2024 and 77% have made online purchases in the past year [3], with younger demographics leading the way, showing that this growth is likely to continue in the foreseeable future.

We have watched the rise of giants like Amazon and the Alibaba group, who have set new standards for customer experience, logistics, and supply chain management on a global scale. These large enterprises have proven that efficient e-commerce operations can drive significant revenue growth and to this day in Europe, larger enterprises are more likely to sell online than smaller ones, with 24.44% of turnover coming from online sales in large companies, compared to just 14.99% for medium and 9.49% for small enterprises [2]. However, the e-commerce boom is not limited to large players, small and medium-sized enterprises (SMEs) have more opportunities and tools under their disposal than ever before to establish and grow their online presence. Webshop engines like Shopify, Wix and Wordpress based solutions like WooCommerce have democratized access to e-commerce, allowing SMEs to set up online stores with relative ease and minimal upfront investment. This lowers the barrier to entry for many small businesses and aspiring entrepreneurs, leading to a remarkable number of 24.1 million online stores globally [4]. We can also see that most of the online stores are created with a just a handful of these popular platforms, with Shopify alone powering 23.6% of them [4]. These giant platforms can not by themselves cover all the needs of the diverse e-commerce landscape across different regions, industries and business models, thus they need some help from third-party applications and services to fill in the gaps. This thesis project, referred to as Shopickup, comes into play, to solve a major pain point for SMEs in the e-commerce space: the integration and management of shipping carriers.

1.1 Motivation and Problem Statement

While e-commerce platforms provide solid foundations for online sales, due to their general-purpose large-scale international nature, they can not possibly cover all shipping providers of all regions they operate in. SMEs similarly often struggle with fragmented shipping workflows, where order data must be manually transferred between sales platforms like Shopify and heterogeneous carrier systems like GLS, DPD, DHL and many others. This fragmentation leads to operational inefficiencies, data redundancy, and a lack of real-time visibility into shipping performance. To address these challenges, webshop engines allow third-party applications to extend their core functionalities via well-defined APIs.

Multiple such third-party applications already exist, but they most often have limited carrier or region support, or are built for specific use-cases, such as label generation only. They typically rely on platform-specific plugins that are difficult to maintain or monolithic middleware that lacks the flexibility to adapt to new carrier standards rapidly. This project addresses the lack of a unified, extensible integration layer that can:

1. Normalize data across diverse carrier Application Programming Interfaces (APIs) without completely custom code for each new provider.
2. Provide a scalable analytics framework that transforms raw shipping data into actionable business intelligence.

There is a clear need for a more comprehensive, flexible, and scalable solution that can integrate multiple carriers, is easily extensible to support new ones, and provides meaningful insights into shipping performance via key performance indicators (KPIs). The motivation for this project also stems from practical experience supporting a Shopify-based online store.

1.2 Research Objectives

The primary objective of this thesis is to design develop and evaluate a scalable software architecture that unifies e-commerce shipping operations. Research to be done can be divided into two main topics, each of which with their main and related sub-research questions (MRQ and SRQ, respectively). Inevitably, to present the whole picture, I will have to cover some of the other aspects of the implementation as well, but the focus will remain on the most relevant technical aspects.

1.2.1 Architecture and Extensibility

The first part of this study focuses on the structural design of the Shopickup application, going into more detail about the architecture and extensibility aspects.

MRQ1. What is the most appropriate architecture for a common, extensible API adapter that integrates multiple shipping carriers with heterogeneous APIs, data formats and workflows, and what are the advantages and disadvantages of the candidate architectures?

1.2.2 Evaluation and Analytics

The second part of this study focuses on the usability and effectiveness of the proposed system.

MRQ2. How can a data analytics dashboard be designed such that merchants receive actionable insights into shipping performance and customer preferences while balancing usability, scalability, and real-time needs?

> **SRQ2.1.** How can the effectiveness of such a system in shipping operations be evaluated, which KPIs are most relevant to merchants?

1.3 Scope and Limitations

This study focuses on the development of a functional prototype integrating Shopify with a selected set of international shipping providers (e.g.: GLS, Austrian Post, MPL and others) to demonstrate the proposed architecture. While the system is designed for extensibility, full integration with all major global carriers is beyond the scope of this bachelor thesis, but potentially open sourcing the shipping carrier adapter framework will allow future contributions to expand its capabilities if desired. Implementation of the analytical dashboard and the overall application will be evaluated on our own e-commerce store's data in-depth and to a lesser extent on surveys and feedback from other business owners using the system.

1.4 Thesis Structure

Throughout this thesis, the standard IMRAD structure (Introduction, Methods, Results, and Discussion) is followed, mirroring the process of scientific discovery, as well as my journey of development along the way. The remainder of this thesis is structured as follows, as per the IMC Krems guidelines:

Chapter	**Title**	**Contents (summary)**
2	Related Work	Reviews prior work in API integration, middleware for logistics, and related integration patterns; positions this thesis relative to existing solutions.
3	Methodology	Details the theoretical framework, architectural choices, design criteria and evaluation methodology used for the proposed system.
4	Implementation and Results	Describes the Shopickup prototype implementation, integration with selected carriers and platforms, and presents experimental results.
5	Discussion	Interprets results, discusses limitations, trade-offs and lessons learned.
6	Conclusion and Future Work	Summarises findings and outlines potential directions for further development and research.

Table 1.1: Thesis structure and chapter summaries

Chapter 2 Related Work

Finding existing literature that directly addresses the design of an extensible, multi-carrier shipping integration layer for SMEs is challenging. Most scientific work either focuses on logistics and supply chains (routing, last-mile optimization, warehouse management) or on general software architecture and integration patterns (SOA, microservices, API gateways). There is a clear gap when it comes to shipping integrations at the API level between e-commerce platforms and parcel carriers such as DHL, GLS, or national postal services. This chapter therefore proceeds in two steps. First, it reviews general software architecture approaches for integrating heterogeneous APIs and argues how these can be applied to the domain of multi-carrier shipping. Second, it briefly positions existing commercial and open-source solutions relative to the architectural patterns discussed, thereby highlighting the research gap addressed by this thesis.

2.1 Software Architecture for API Integration

The core technical problem of this thesis is the integration of multiple heterogeneous carrier APIs into a single, unified application in a way that remains extensible when adding new carriers or changing existing ones. This problem is a specific instance of the broader topic of enterprise application integration and service-oriented architecture (SOA).

2.1.1 Service-Oriented Architecture as Theoretical Foundation

Let us first establish a theoretical foundation based on SOA principles. What better way to start than with a definition? While there is evidently no single agreed-upon definition of SOA [5], for the purpose of introducing it we turn to the book of Juric et al., which provides a concise and practical definition:

> "In short, it is an Enterprise Architecture where applications are designed to provide coarse-grained services, which are consumed by Business Processes or other integration applications. Service-Oriented Architecture is both a design concept and an architecture. The design concept in SOA is about designing applications/systems that have well defined self-describing access interfaces, with the services being composed into business processes. The architecture is about having simple mechanisms to use these access-interfaces for Integration of the Enterprise."

It is easy to see how this thesis's projects calls for use of SOA practices, specifically the architectural approach of simple mechanisms for using different access-interfaces. Papazoglou emphasizes that it "captures the logical way of designing a software system to provide services to end-user applications or other services distributed over a network" [6]. The key principle is loose coupling, achieved by a clear separation between service interfaces and service implementations. In an SOA-based design, an application does not directly depend on the internal details of another application, instead it interacts with well-defined service contracts. For the problem at hand, this translates to the idea that our core system should not be tightly coupled to the internal details of DHL's, GLS's, or Austrian Post's APIs, as these are likely to change over time and differ significantly between providers anyway. Instead, it should rely on a stable, domain-specific interface (e.g., "create shipment," "track shipment," "cancel shipment") outlining the capabilities required for users of our system, while carrier-specific logic is encapsulated in separate components behind these interfaces and abstractions.

A systematic literature review by Niknejad et al. synthesizes 103 primary studies and concludes that much of the case studies and implementations of SOA show the positive impact of the before mentioned characteristics and that it has "proven to be a key paradigm in numerous industries" [5]. It also highlighted however, that the biggest problem with SOA adoption is the topic of governance and the lack of empirical studies on measuring its success, however these are out of scope for this thesis. Applying these principles to this project, we can outline the following high-level architectural guidelines:

- The core business logic (e.g., order management, user interface) interacts with a generic shipping service interface rather than individual carrier APIs.
- Each carrier integration is treated as a service that implements this generic interface.
- Adding or replacing a carrier should require only changes in the implementation of the corresponding service, not in the rest of the system.

This conceptual foundation is important, but it does not yet specify concrete implementation structures or design patterns.

2.1.2 Empirical Evidence of REST API Heterogeneity

In practice, most modern integrations are realized via RESTful web APIs over HTTP. A common assumption in industry is that REST APIs are similar, implying that integration should be straightforward across different providers. However, this is not necessarily the truth in most cases. Neumann et al. concluded through an extensive empirical study of 522 public web service APIs that there is “high diversity in services, including differences in adherence to best practices, with only 0.8% of services strictly complying with all REST principles.” [7] and that even among “REST” APIs, there is only limited compliance to strict REST constraints. In other words, there is no real homogeneity at the API level—each provider does things slightly differently. One very common example many developers encounter day by day is for example the variety of authentication methods or the lack of a variety of available REST methods (GET, POST, PUT, DELETE, etc.) for operations that would logically map to them.

Translating this to the parcel-shipping context, even if DHL, GLS, Austrian Post, and others expose “REST” APIs, their URL structures, authentication, request bodies, responses, and error handling differ in non-trivial ways. A naive point-to-point design that integrates each carrier directly into the core application will inevitably accumulate a large amount of carrier-specific branching logic, making the system brittle and hard to maintain. The empirical heterogeneity demonstrated by Neumann et al. thus again provides a strong justification for an explicit abstraction and normalization layer aligning with SOA guidelines, rather than relying on ad-hoc HTTP calls scattered throughout the codebase.

2.1.3 Adapter Pattern for Carrier Abstraction

The challenge of integrating heterogeneous systems with incompatible interfaces is not new to software engineering. The Adapter pattern is a well-established structural design pattern introduced in the foundational work “Design Patterns: Elements of Reusable Object-Oriented Software” by Gamma et al. all the way back in 1994 [8]. It works by introducing an intermediary adapter that implements a target interface (the one the client expects) while internally delegating to an adaptee (the existing component with an incompatible interface).

This pattern has two primary implementation variants, one commonly referred to as Class Adapter, which is inheritance based, and the other as Object Adapter, which is composition based. In the context of this project, the latter one is more appropriate, as we are dealing with external carrier APIs with no benefit of inheritance. The following is a simplified illustration of how the Adapter pattern can be applied to the multi-carrier shipping integration problem:

```
interface CarrierAdapter {
    createLabel(order): Label
    trackShipment(trackingId): TrackingInfo
    cancelShipment(trackingId): CancellationResult
  }
```

Listing 2.1: Carrier adapter implementations

For each carrier, a dedicated adapter implements this interface:

```
class DHLAdapter implements CarrierAdapter { ... }
class GLSAdapter implements CarrierAdapter { ... }
class AustrianPostAdapter implements CarrierAdapter { ... }
```

Listing 2.2: Carrier adapter implementations

Each adapter is responsible for:

- Translating the generic Shopickup domain model into the carrier-specific request model (e.g., mapping address fields, parcel dimensions, and service options).
- Calling the carrier's API, including managing authentication, endpoints, and error conditions.
- Translating the carrier's response into a normalized internal representation suitable for downstream processing and analytics.

This pattern has several benefits, which align closely with the quality attributes targeted by this thesis:

- Encapsulation of change: When a carrier changes their API (e.g., URL, auth method, required fields), only the corresponding adapter needs modification, not the rest of the system.
- Extensibility: Adding a new carrier requires implementing a new ‘CarrierAdapter‘ without altering existing adapters or the core logic, directly supporting the Open/Closed Principle.
- Testability: Each adapter can be unit-tested in isolation with mock carrier APIs.

The Adapter pattern therefore provides a concrete, implementation-level mechanism to realize the carrier-agnostic abstraction suggested by SOA and motivated by REST API heterogeneity.

2.1.4 API Gateway as a Unified Front Door

While the Adapter pattern addresses the structural coupling problem at the code level, it does not by itself solve several cross-cutting concerns that inevitably arise in a multi-carrier context:

- Authentication and authorization of external clients

- Rate limiting and throttling (e.g., to respect carrier rate limits)
- Central logging and monitoring of requests
- Versioning and gradual rollout of new functionality
- Caching of frequently requested data (e.g., tracking information)

In microservice and distributed architectures, these concerns are commonly handled using an API Gateway as an extra intermediary layer. Ochuba et al. have written an extensive review on API Gateway patterns [9] where they first argue that with the rise of SOA and microservices, and that API Gateways have become a foundational component in modern scalable, secure systems, particularly when integrating multiple backend services.

> "As enterprises increasingly adopt service-oriented designs, the need for robust intermediary layers that can handle diverse client protocols and varied backend requirements has become essential. In this context, API gateways not only provide a unified interface but also act as enforcement points for both technical and business logic" [9]

They also identified a number of roles and patterns an API Gateway typically plays such as Aggregator, Proxy, Adapter for roles and patterns such as Edge Caching, which could be very relevant for some functionalities of our use case [9]. Applied to Shopickup, an API Gateway would serve the following purposes:

- Expose a single REST API towards the e-commerce frontend (and potentially other clients such as ERP systems), with stable endpoints like ‘POST /shipments‘, ‘GET /shipments/id‘, and ‘GET /shipments/id/tracking‘.
- Internally, route requests to the appropriate carrier adapter service based on parameters such as the selected carrier, destination country, or configured routing rules.
- Provide centralized rate limiting per carrier (e.g., to avoid exceeding DHL's API quota).
- Apply caching for certain idempotent operations, such as repeated tracking queries within a short time window.
- Ensure consistent logging and monitoring across all carrier integrations.

By combining the Adapter pattern at the service level with an API Gateway at the edge, the project could offer a single, coherent interface to clients while encapsulating all carrier-specific complexity in backend modules. This architectural combination directly supports the goal of building an extensible, maintainable multi-carrier integration layer.

2.1.5 Extensibility as a Measurable Architectural Quality

Extensibility has been mentioned multiple times as a key design goal for this thesis, so it is worth discussing it in more detail, as it is not only a design intention but can be treated as a measurable quality attribute. The technical report by the Software Engineering Institute (SEI) at CMU discusses extensibility as a specific architectural concern and outlines how extension points and extension strategies should be identified and evaluated. In the following table I will present their view on what a system's extensibility can be assessed based on and how these criteria map to the thesis domain of multi-carrier shipping integration.

Criterion	**Thesis domain example**
Appropriateness of extension points	The `CarrierAdapter` interface (create/track/-cancel) and its method signatures cover the majority of required carrier behaviors and potential extension options.
Degree of encapsulation and coupling	Can adapters be added/removed without touching core business logic? Evaluates whether carrier-specific code is fully localized to adapter modules.
Cohesion of extension mechanisms	Adapter implementations contain only carrier-specific responsibilities (mapping, auth, error handling) and avoid leaking cross-cutting concerns.
Extension lifecycle: performance, testing, deployment, etc.	Ease of unit/integration testing of adapters in isolation; expected performance impact; deployment/update workflow for individual adapters.

Table 2.1: Extensibility evaluation criteria as defined by SEI and how they map to the thesis domain

In parallel, Petrillo et al. have done a literature review of 56 studies, showing which metrics are most commonly used by researchers. They found that coupling was one of the most frequently measured metrics and highlighted that generally low coupling and high cohesion are essential for extensible architectures. [10] Moreover, of these 56 papers, 6 had extensibility as their main focus, showing that it is an actively researched topic, especially with the rise of agile development practices and SOAs. These theoretical and metric-based perspectives support using extensibility, coupling, and cohesion as criteria when comparing different architectural options and could be used later in the evaluation chapter to assess the chosen design.

2.2 Architectural Styles for Multi-Carrier Integration

Based on the literature and the guiding principles discussed above, several architectural styles can be considered for implementing a multi-carrier integration layer. The following candidates are neither exhaustive of all possible options, nor are some of them seriously considered for implementation, but they still serve as useful points of comparison to understand trade-offs.

1. Point-to-point integration
2. Hub-and-spoke broker
3. Message-oriented service bus
4. API gateway-centric architecture
5. Event-driven microservices

The following subsections describe each of these in turn, discuss their advantages and disadvantages, and relate them to the goals of this thesis.

2.2.1 Point-to-Point Integration

In a point-to-point architecture, the application establishes direct connections to each external system. For the thesis project, this would mean that the backend directly invokes the DHL, GLS, Austrian Post, etc. APIs wherever shipping functionality is needed, often with branching logic such as:

```
if carrier == "DHL": call_dhl_api(...)
elif carrier == "GLS": call_gls_api(...)
...
```

Listing 2.3: Carrier dispatch example

This approach is simple and often used in early-stage prototypes because it has a low upfront cost and development time, by having no additional infrastructure or architectural components. Actually, this approach was initially used first in the project, up until two carriers were integrated, to quickly validate and develop other core functionalities, which I will discuss in future chapters. Clearly, this is not very scalable and like I have also experienced in my own small project. Literature also consistently identifies point-to-point integration as problematic at scale, even describing it as an expensive approach and bad extreme on the scale of integration [11]. Every new integration increases the number of dependencies and potential failure points, and changes in one system can ripple unpredictably through others. In the context of Shopickup, the main disadvantages of point-to-point integration, as experienced firsthand and reported in literature, include:

- High coupling: The core application must “know” about the details of every carrier API, violating SOA principles and making change localized only with difficulty.
- Poor extensibility: Adding a new carrier involves editing central business logic and testing the entire application, not just an isolated module.
- Limited reuse: Any normalization or error-handling logic gets duplicated across multiple code paths.

For these reasons, point-to-point integration can serve as a baseline or initial stage but is not a suitable architecture for the extensible multi-carrier platform targeted in this thesis.

2.2.2 Hub-and-Spoke (Central Broker)

A hub-and-spoke architecture introduces a dedicated broker component that sits between the core application and the external systems. The core application communicates only with the broker using a canonical data format. The broker then routes and transforms messages to and from individual spokes, which are the carrier adapters. In the projects context, the broker would:

1. Accept generic shipping requests such as `CreateShipment(order, carrier)` in a normalized format.
2. Select the appropriate carrier adapter based on the requested carrier or routing logic.
3. Transform the canonical request into the carrier’s specific format and invoke the carrier API.
4. Transform the carrier’s response back into the canonical format before returning it to the core application.

This architecture is closer to SOA principles and facilitates the use of Adapter and normalization patterns. It brings several benefits:

- The core application is decoupled from carrier details and depends only on the broker’s canonical interface.
- All data transformation is centralized in the broker, improving consistency.
- It becomes easier to add logging, basic rate limiting, or validation in one place.

However, this approach also has drawbacks. The broker can become a single point of failure and a performance bottleneck, especially if all traffic is routed synchronously through it, scaling is complex. In practice, hub-and-spoke architectures are often a stepping stone towards more distributed patterns such as service buses or microservices with API gateways. When contemplating the architecture for the system, after seeing the limitations of point-to-point integration, this pattern felt like a natural next step.

2.2.3 Message-Oriented Service Bus

A more decoupled architecture uses a message-oriented middleware or service bus, where components communicate via asynchronous messages, often in a publish-subscribe model via message queues or topics. This approach is widely applied in enterprise integration and IoT middleware, where the asynchronous nature of communication is not a limitation. In our practical example, a service bus architecture might work as follows:

- The core application publishes a `CreateShipment` message to a queue or topic, containing the order data and the selected carrier.
- One or more carrier adapter services subscribe to these messages. Each adapter handles messages for its specific carrier, performing the necessary API calls and transformations.
- Once a label is created, the adapter publishes a `ShipmentCreated` event, which the core application or other services consume to update their state or trigger notifications.

This approach has obvious advantages in terms of loose coupling, scalability, resilience and throughput. The disadvantages are mostly related to complexity:

- System behavior becomes asynchronous and eventually consistent; labels may not be available immediately after the order is placed.
- Debugging and tracing flows across multiple queues and services is more complex than direct synchronous invocation.
- Operating and securing a message bus comes with its own operational challenges and complexities.

Given the scope of a bachelor thesis and the requirement for near-real-time feedback in this e-commerce workflow (e.g., changing order states instantly), a fully asynchronous service bus architecture is likely too heavy and unnecessarily complex for our purposes and is more suitable on the higher level on a general e-commerce platform architecture, such as in Baraka and Al-Ashqar's paper [12]. However, other specific parts of the system, such as background tracking updates, scheduled tasks such as pickup location synchronization, or analytics data processing, could benefit from message-oriented designs in future iterations.

2.2.4 API Gateway-Centric Multi-Carrier Architecture

The API Gateway pattern offers a pragmatic compromise between the simplicity of direct integration and the flexibility and extensibility of more advanced service-oriented approaches. It is particularly well-suited for systems that expose a unified API to clients while internally delegating work to multiple backend services. It would work as follows in our context:

- The gateway exposes a stable REST API to the frontend and other consumers.
- Each carrier integration is implemented as a separate service (Adapter) that adheres to a shared internal interface.
- The gateway routes requests to the appropriate adapter based on routing rules (e.g., selected carrier, destination).
- The gateway can also perform cross-cutting tasks such as authentication, rate limiting, logging, and caching.

This architecture combines the benefits of the Adapter pattern with the centralized control of an API Gateway. Compared to hub-and-spoke, the gateway approach more naturally aligns with microservices and modern DevOps tooling, and commercial and open-source gateway solutions (e.g., Kong, NGINX) are mature and well-supported. At the same time, it avoids some of the operational complexity of full event-driven microservice architectures.

2.2.5 Event-Driven Microservices

The most advanced architecture considered in the literature combines an API Gateway with a fully event-driven microservice back end. In such systems, individual business capabilities—Orders, Shipping, Tracking, Notifications—are each implemented as independent services that communicate via events on a message bus (e.g., Kafka). The advantages of this style include:

- High scalability and fault tolerance: each service can be scaled independently, and failures in one area (e.g., tracking updates) do not necessarily affect others.
- Fine-grained deployment and evolution: services can be updated and re-deployed independently.
- Fault tolerance and resilience: services can retry operations, handle backpressure, and recover from transient failures more gracefully.

However, systematic reviews of microservice systems also highlight the increased architectural complexity, the need for sophisticated monitoring and testing, and the challenge of managing eventual consistency across many services [13]. TODO: Check again These factors make full event-driven microservice architectures more appropriate for large organizations with substantial engineering resources (e.g., Amazon, Netflix, Uber) than for SME-oriented projects or bachelor thesis prototypes. For this thesis, event-driven microservices are therefore considered primarily as a reference point for future evolution rather than as the immediate target architecture.

2.2.6 Comparative Assessment and Architectural Choice

Synthesizing the above subsections, the architectural options can be compared along the axes that matter for this thesis: extensibility, complexity, and fitness for the SME/Shopickup context.

Architecture	Pros	Cons
Point-to-point	Simple; low upfront cost for prototypes.	High coupling, poor extensibility, duplicated logic; hard to maintain at scale.
Hub-and-spoke	Centralised transformation, consistent logging and validation.	Broker can be a single point of failure and performance bottleneck.
Message-oriented service bus	Loose coupling, resilience, natural retry/backpressure, good scalability.	Increased complexity, eventual consistency, harder debugging and ops.
API gateway-centric	Centralised cross-cutting concerns (auth, rate limiting, logging); extensible and pragmatic.	Requires gateway infrastructure and some operational overhead.
Event-driven microservices	High scalability, independent deploys, fine-grained evolution.	Highest operational and conceptual complexity; heavy monitoring/testing needs.

Table 2.2: Comparative assessment of candidate architectures for Shopickup

Given the project scope and the desire for a balance between extensibility and operational effort, lightness, the API gateway-centric architecture combined with adapter-based carrier integrations is selected as the most suitable trade-off.

2.3 Preexisting Solutions and Papers in this Domain

As I have previously mentioned, there is a lack of academic literature directly addressing multi-carrier shipping integration at the API level. Most relevant conference paper I could find is by Lin et al. in 2010, in which they also first emphasize the gap in this research topic, then follow up by proposing a SOA guided, J2EE based framework for integrating shipping providers [14]. While their work is conceptually aligned with the principles discussed earlier, especially regarding coupling and extensibility, as also mentioned by the authors themselves, their implementation is now somewhat dated. The industry has since largely moved towards lightweight RESTful APIs and microservice architectures, not to mention the replacement of J2EE and XML-based web services with more modern frameworks. Therefore, while Lin et al.'s work provides a useful historical perspective and validation of SOA principles and wrapper patterns, even message brokers, it does not directly inform the design of a contemporary system like Shopickup.

Most other existing literature in the domain are more general and aim to theorize a broader e-commerce architecture. These can still to some extent be relevant, as they often highlight the same challenges and propose similar SOA-based solutions, or may even address logistics and shipping as part of a larger system. Good examples include the following works:

- The paper of Bhakar and Al-Ashqar, in which they propose an enterprise service bus the central integration layer for their purchase order management system, with a separate shipping service module [12].
- Xiong-yi similarly proposes SOA as the better alternative for e-commerce systems and highlights the heterogeneity of external systems, as we have also seen before in a more general context with APIs [15].
- Li's paper confirms that the adapter method is now a standard approach for encapsulating legacy service interfaces in modern SOA e-commerce platforms [16].

It is also worth mentioning that some similar commercial and even open-source solutions do exist to what we are trying to achieve, however they are often fully fledged applications with limited extensibility, such as karrio [17], or are indeed lightweight, but limited to specific vendors or use-cases and follow more of a point-to-point approach, such as ShippingProAPICollection [18].

2.4 Design, Evaluation and Analytics of Shipping Monitoring Systems

The second half of my literature review focuses on the more usability focused aspect of the application with the following two research questions in mind:

MRQ2 How can a data analytics dashboard be designed such that merchants receive actionable insights into shipping performance and customer preferences while balancing usability, scalability, and real-time needs?

SRQ2.1 How can the effectiveness of such a system in shipping operations be evaluated, which KPIs are most relevant to merchants?

Analytical dashboards and monitoring systems are widely used in various domains, including e-commerce logistics and However, it is not immediately clear how to best design and evaluate such a system, that is both efficient and provides real value to merchants. In cases like this, we can not just simply measure system performance in just technical terms, such as the Google SRE team's Four Golden Signals [19] (latency, traffic, errors, saturation), but also need to consider user-centric and domain specific metrics as well. The following subsections explore relevant literature on dashboard design architectures, KPIs and evaluation methods in the context of shipping monitoring systems.

2.4.1 On the Importance of Technical Metrics

Technical infrastructure metrics—uptime percentages, query response latencies, and system throughput—represent necessary conditions for effective monitoring systems but are not sufficient indicators of value. A shipping monitoring dashboard with near perfect uptime and zero latency means little if the presented information is not understandable or actionable. Thus, evaluation of monitoring systems must be addressed on both technical and user-centric dimensions.

Infrastructure metrics often simply serve as enablers of business value rather than measures of value themselves. Google's Four Golden Signals framework [19] provides useful infrastructure monitoring patterns, but requires contextualization for shipping domains:

- **Latency** for example in case of our application is not as crucial as in some other real-time systems, such as financial trading platforms for example. The main importance of latency in our case is more about user experience when merchants interact with the dashboard, rather than strict real-time constraints.
- **Traffic** and throughput indicate system load but reveal nothing about whether the dashboard drives better merchant decisions. It is, however an important factor when considering scalability, as in our case we are very often also constrained by external API rate limits we must respect, such as carrier APIs or Shopify's Admin API.
- **Error rates** measure system reliability, and are critical for both trustworthiness and usability. It must be monitored closely with each added integration and feature.
- **Saturation** indicates when scaling is needed but not whether the system generates merchant value. It is also not much of a concern in our case, as there are not many computationally intensive operations, and due to the web technologies used, scaling is often handled automatically by the cloud provider.

Evidently, in our case, while technical metrics are important for ensuring reliable operation, what we are really looking for are user-centric KPIs that reflect the dashboard's effectiveness in driving merchant decisions.

2.4.2 Real-Time vs. Batch Analytics

Now let us address the main topic of MRQ2, which is the architectural and design choice of our proposed dashboard system. A fundamental architectural decision in shipping monitoring systems is the choice between real-time and batch analytics approaches. This decision is not primarily technical but rather reflects what decision-making problems, time horizons and type of data analysis the system aims to support.

Operational Intelligence vs. Business Intelligence and Their Different Needs

To understand the distinction, we first need to clarify the difference between operational intelligence and business intelligence. For this, I turn to Sandu's definitions [20]:

> "BI enables business users to report on, analyze and optimize business operations to reduce costs and increase revenues. Organizations use BI for strategic and tactical decision making where the decision-making cycle may span a time period of several weeks or months. ... there is now a need to use BI to help drive and optimize business operations on a daily basis ... usually called operational business intelligence and real-time business intelligence." [20]

Batch analytics usually processes accumulated data in scheduled bulk operations, enabling comprehensive analysis of patterns over longer time horizons and greater amounts of data. A batch job that runs weekly might analyze all shipments of the given week to give greater insight on carrier performance, cost trends, or customer preferences.

Operational intelligence focuses on current situations, and thus they require low-latency, real time data processing. Real-time analytics processes data continuously as events occur, enabling decisions in motion while situations are still unfolding. This rapid response window—often measured in minutes rather than hours—enables operational intelligence, acting on immediate situations, such as shipment delays or inventory shortages.

These two intelligence types serve different decision problems and cannot easily substitute for each other. The architectural implications are profound as well. On one hand, real-time systems prioritize latency by accepting operational complexity. When discussing big data scale, real-time analytics are especially hard, as evidenced by the research of Zheng et al. [21] in which they find that real-time big data systems must simultaneously satisfy strict timeliness, high stability and availability on different levels of scale which leads to substantially higher system complexity compared to traditional batch-oriented data warehouses. On the other hand, batch systems prioritize analytical comprehensiveness by accepting latency. Data is extracted from operational systems, transformed through staged pipelines, and in case of great amounts of data loaded into dimensional data warehouses where complex SQL queries execute efficiently. Historical data is preserved for deep analysis. However, data arrives with significant latency and usually based on scheduled intervals, limiting responsiveness to immediate disruptions. A batch system discovering that a carrier had a 20% delay rate last week due to issues specific to it is useful for strategic planning, but too late for immediate action on current shipments such as switching on turning off the carrier.

It is apparent, that most business use cases require both types of intelligence and thus the corresponding analytical methods and architectures. A good way to decide between real-time and batch analytics for given problems is by considering the decision-window requirements for them.

Here are two illustrative use cases for shipping monitoring systems:

- **Real-time use case (Operational Intelligence)**: Shipment Service Level Agreement (SLA) breach alerts. Shipping carriers often offer promised delivery windows, such as "delivery within 2 business days", and merchants need to be able to monitor these SLAs, both to proactively manage customer expectations and to take corrective actions against carriers when breaches occur.
- **Batch use case (Business Intelligence)**: Weekly carrier performance reports. Merchants need to analyze carrier performance over longer periods to inform strategic decisions about carrier selection and customer preferences, contract negotiations, and logistics planning.

Lambda Architecture as a Hybrid Reference Model

To formalize the hybrid approach described above, I turn to the Lambda Architecture, a data-processing design pattern introduced by Nathan Marz [22] to handle massive quantities of data by taking advantage of both batch and stream-processing methods. While originally conceived to address the complexities of “Big Data” and the CAP theorem in distributed systems, the core principles of the Lambda Architecture provide a robust reference model for this project’s monitoring system, even at a smaller scale. Marz argues that traditional databases conflate two distinct problems: storage of immutable facts and the computation of views (queries) on those facts [22]. In purely real-time or mutable systems, ensuring consistency while maintaining availability (as per the CAP theorem) introduces significant complexity, such as read-repair logic, vector clocks, and the risk of permanent data corruption from buggy writes. To “beat” this complexity, the Lambda Architecture proposes separating the system into three distinct layers, a structure that aligns directly with the hybrid requirements identified in the previous section:

1. **Batch Layer (Master Dataset):** Marz defines the “master dataset” as an immutable, append-only set of raw data [22]. Because data is never overwritten, human errors or buggy code cannot permanently corrupt the system; views can simply be recomputed. In the context of shipping monitoring, this corresponds to the nightly storage of all raw shipment events and carrier updates. This layer precomputes complex, heavy views (e.g., weekly carrier performance reports) that don’t need to be up-to-the-second but must be accurate and comprehensive.

2. **Speed Layer (Real-Time):** Since batch views are inherently out of date by the duration of the batch interval, a separate "Speed Layer" compensates for this "data lag" [22]. This layer processes only the most recent data (the delta) to provide low-latency views. Marz notes that while this layer requires complex incremental algorithms and eventual consistency, any errors here are transient—they are eventually overwritten by the correct Batch Layer views.

3. **Serving Layer:** "The serving layer pulls together the results from both the speed and batch layers, which makes an easy single view for the applications and users, which consume the data." as defined by Lekkala [23].

While Marz's original proposal suggests heavy industrial technologies like Apache Hadoop for the speed layer [22], the pattern is technology-agnostic, which is made apparent by how different papers implement it or suggest implementing it via different technologies [23], [24]. For a focused application like Shopickup, strictly adhering to these "Big Data" technologies would introduce unnecessary operational overhead, which is a common issue noted by critics, such as Krep who points out that maintaining two parallel codebases (one for batch, one for streaming) for the same purposes violates the "Don't Repeat Yourself" (DRY) principle and increases maintenance costs [25]. However, the functional separation remains valid and valuable. Instead of Hadoop and Storm, Shopickup can implement a lightweight approach that takes inspiration from the Lambda architecture:

- The **Batch Layer** can be implemented effectively using standard relational database aggregations or scheduled cron jobs that run during off-peak hours to generate static reporting tables, which later could be replaced by a more robust ETL pipeline if needed.

- The **Speed Layer** does not require a complex event-streaming engine (like Kafka) if the data volume does not demand it. Instead, frequent polling of carrier APIs (e.g., every 5 minutes) or lightweight webhook handlers if available can serve the same functional purpose: capturing the latest state to alert merchants of immediate issues.

- The **Serving Layer** can be a simple API that merges results from both layers, serving real-time alerts from the Speed Layer and historical reports from the Batch Layer to the dashboard.

Based on the literature research this hybrid approach seems to be a suitable candidate for our project, which I will further explore in the design chapter. Were the scope of the project larger, Lambda's alternative, the Kappa architecture proposed by Kreps [25] or even a CQRS + Event Sourcing approach [26] might be considered.

2.4.3 Domain-Specific KPIs for Shipping Operations

With understanding of the real-time/batch distinction established, we can now continue with the SRQ2.1 and define domain-specific KPIs appropriate for merchant evaluation and decision-making. KPIs of organizations are, as Piela summarized usually organized by factors most relevant to the company, such as whether they are financial or non-financial, primary or secondary metrics, etc. [27]. We however, categorize them into tiers reflecting the different decision horizons they support, as described earlier. This serves both to clarify their purpose and to guide the underlying data architecture needed to support them. We will be in a later chapter attempt to implement these KPIs into our own project's dashboard according to SMART criteria [28], then evaluate their effectiveness based on merchant feedback and the IS system success model by DeLone and McLean [29].

Operational KPIs: Real-Time Monitoring for Immediate Action

Operational KPIs are designed to support immediate decision-making, often in response to unfolding events. These KPIs are typically displayed on real-time dashboards and trigger automated alerts when thresholds are breached. The following table summarizes domain-specific operational KPIs relevant to shipping monitoring systems and their sources: (Note that KPI research is still ongoing, so this table and the references are not yet complete)

KPI	Description	Source
On Time Delivery Rate (OTDR)	What percentage of shipments are delivered on time according to the promised delivery date.	Piela [27]
OTDR SLA	Here I believe it is important to differentiate for merchants the measure of OTDR as suggested by the merchant for customers and the OTDR in regard to broken SLAs	None

Table 2.3: Operational KPIs for shipping monitoring systems

Strategic KPIs: Batch Monitoring for Tactical Planning

Strategic KPIs are designed to inform long-term planning and strategic decision-making, based on comprehensive historical data.

KPI	**Description**	**Source**
Accuracy of Forecasting	Measures how accurately the system predicts the delivery times and volumes.	
Carrier Cost per Shipment	Average cost incurred per shipment by each carrier. Helps in negotiating contracts and selecting carriers.	
Damage Rate	Percentage of shipments that are reported as damaged upon delivery. Indicates carrier handling quality.	Piela [30]
Return Rate	Percentage of shipments that are returned by customers. High return rates may indicate issues with product quality or shipping accuracy.	

Table 2.4: Strategic KPIs for shipping monitoring systems

Chapter 3
METHODOLOGY

This chapter first describes, with an emphasis of technical details, the commercial implementation of my application known as Shopickup. This serves as context for the two other main parts of this chapter, which describe how based on the literature review, what design choices were chosen for the carrier integration architecture and the monitoring dashboard respectively. For each of the latter two sections, I first describe and argue for the chosen design, then follow up with strategies for evaluation.

3.1 Shopickup Application Overview

Shopickup is a multi-carrier shipping integration and monitoring platform designed for e-commerce merchants. It allows merchants to extend their online stores with a unified interface for managing shipments across multiple carriers, print labels, track packages, and gain insights into shipping performance through a dedicated dashboard. Customers of these stores also have the benefit of selecting from multiple shipping options, including pickup points through an interactive map interface.

3.1.1 User Flows

The best way to understand the user flows of Shopickup is through the following diagram, which will illustrate the main interactions both for merchants and customers here.

Then we will describe the labeled user flows, both for merchants and customers, giving an idea of how the application is used in practice.

3.1.2 Architectural Overview and Tech Stack

Here will be an overall high-level overview of the system architecture, describing the main components and the technology stack used for each of them. For this purpose, a detailed and labeled UML diagram of the system will be provided here...

...which will then be followed by a description of each component and the technologies used.

3.1.3 Limitations and Scope

It is also important to address some of the shortcomings of the practically implemented system:

- **Platform agnosticism:** While the backend is built in a way where most of it is platform-agnostic and reusable, at the time of writing the frontend is primarily designed to integrate with Shopify's admin dashboard via an embedded app.
- **International carriers:** Although the carrier integration system verifiably works with multiple national carriers (such as MPL for Hungary and Österreichische Post for Austria), the practical application was only released for the Hungarian market.

3.2 Carrier Integration Architecture Design

As described in the literature review, after considering various architectural options, an API gateway-centric architecture combined with adapter-based carrier integrations was chosen as the most suitable trade-off for the carrier integration system of Shopickup. This section will describe the specifics of how this architecture was implemented, what technologies were used, and the challenges faced during development.

At the time of writing, this part of the system has not been developed yet, so a technology-agnostic design description will be provided instead:

Component	Description
API Gateway (edge)	Single ingress for clients; handles auth, tenant routing, rate limiting, request validation, caching, and routing to internal services. Routes requests to: Adapter Orchestrator, Tracking Service, or Analytics/Reporting endpoints.
Adapter Orchestrator / Broker	Small coordinator that selects the correct carrier adapter based on routing rules (carrier ID, destination, merchant settings). Implements retries, backoff, circuit breakers, and synchronous vs asynchronous dispatch choices.
Carrier Adapters (one per carrier)	Implement a fixed internal interface (e.g., createShipment, getLabel, track, cancel). Responsible for mapping canonical domain model -> carrier request, calling carrier API (including auth), interpreting responses and errors, and returning normalized output. Packaged as independently deployable modules/plugins.
Canonical Data Model / Normalizer	Shared domain schema used inside the platform; adapters translate between canonical and carrier models.
Tracking Service / State Reconciler	Periodic polling or webhook consumer that ingests carrier state changes, normalizes them, and updates shipment state; emits events for alerts/analytics.

After a more detailed and technical description of the implementation comes a section describing the typical request flows through the system for common operations such as creating shipments, printing labels, and tracking packages.

In the final part of this section, I will also describe it in more detail, regarding our most important criteria, which is extensibility and maintainability.

3.3 Monitoring Dashboard Design

Based on the literature review, a hybrid Lambda Architecture-inspired design was chosen for the monitoring dashboard of Shopickup. This section will describe the specifics of how this architecture was implemented, what technologies were used, and the challenges faced during development.

3.4 Evaluation Methodology

Describe the planned evaluation methods of the dashboard system and used KPIs. Both qualitative interviews and quantitative perhaps according to the DeLone and McLean IS success model.

The plan is to conduct user interviews with merchants using the dashboard to gather qualitative feedback on usability, usefulness, and overall satisfaction. Additionally, system usage analytics will be collected to measure engagement metrics such as session duration, feature usage frequency, and task completion rates. These quantitative metrics in conjunction with survey data will help assess the effectiveness of the dashboard in improving shipping operations.

A combination of these methods will provide a comprehensive evaluation of the dashboard's impact on merchant decision-making and operational efficiency. One planned way to get merchants on board for this evaluation is to offer them heavily discounted or favorable pricing plans in the initial launch phase in exchange for their participation in the study.

Chapter 4
SUMMARY

This exposé summarizes the current state of the thesis. The research phase is essentially complete: the literature review has established the architectural foundations for multi-carrier integration and the dashboard layer, with an API gateway-centric adapter architecture identified as the most appropriate trade-off for extensibility and operational effort, and a hybrid real-time/batch analytics model selected for KPI-driven monitoring, with defined operational and strategic KPIs tailored to shipping contexts.

Based on these findings, the methodology has been defined to guide the subsequent implementation and evaluation phases. The next steps are to operationalize this methodology by detailing the concrete system design, implementing the prototype components, and executing the planned evaluation through qualitative interviews and quantitative measurements. This work will commence immediately following this exposé's approval.

Bibliography

[1] Capital One. “eCommerce Statistics (2025): Sales & User Growth Trends,” Capital One Shopping, Accessed: Jan. 9, 2026. [Online]. Available: https://capitaloneshopping.com/research/ecommerce-statistics/.

[2] Eurostat. “E-commerce statistics,” Accessed: Jan. 9, 2026. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=E-commerce_statistics.

[3] Eurostat. “E-commerce statistics for individuals,” Accessed: Jan. 9, 2026. [Online]. Available: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=E-commerce_statistics_for_individuals.

[4] Capital One. “How Many Online Stores Are There? | 2025 Statistics & Analysis,” Capital One Shopping, Accessed: Jan. 9, 2026. [Online]. Available: https://capitaloneshopping.com/research/number-of-online-stores/.

[5] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, and A. R. B. C. Hussin, “Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation,” *Information Systems*, vol. 91, p. 101 491, Jul. 1, 2020, ISSN: 0306-4379. DOI: 10.1016/j.is.2020.101491. Accessed: Jan. 11, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437920300028.

[6] M. P. Papazoglou and W.-J. van den Heuvel, “Service oriented architectures: Approaches, technologies and research issues,” *The VLDB Journal*, vol. 16, no. 3, pp. 389–415, Jul. 1, 2007, ISSN: 0949-877X. DOI: 10.1007/s00778-007-0044-3. Accessed: Jan. 11, 2026. [Online]. Available: https://doi.org/10.1007/s00778-007-0044-3.

[7] A. Neumann, N. Laranjeiro, and J. Bernardino, “An Analysis of Public REST Web Service APIs,” *IEEE Transactions on Services Computing*, vol. 14, no. 4, pp. 957–970, Jul. 2021, ISSN: 1939-1374. DOI: 10.1109/TSC.2018.2847344. Accessed: Jan. 11, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/8385157.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, Oct. 31, 1994, 496 pp., ISBN: 978-0-321-70069-8. Google Books: 6oHuKQe3TjQC.

[9] N. A. Ochuba, D. Kisina, S. Owoade, A. C. Uzoka, T. P. Gbenle, and O. S. Adanigbo, “Systematic Review of API Gateway Patterns for Scalable and Secure Application Architecture,” *Journal of Frontiers in Multidisciplinary Research*, vol. 2, no. 1, pp. 94–100, 2021, ISSN: 30509726. DOI: 10.54660/.IJFMR.2021.2.1.94-100. Accessed:

Jan. 11, 2026. [Online]. Available: https://www.multidisciplinaryfrontiers.com/search?q=FMR-2025-1-099&search=search.

[10] T. Coulin, M. Detante, W. Mouchère, and F. Petrillo. "Software Architecture Metrics: A literature review." arXiv: 1901.09050 [cs], Accessed: Jan. 11, 2026. [Online]. Available: http://arxiv.org/abs/1901.09050, pre-published.

[11] T. Gulledge, "What is integration?" *Industrial Management & Data Systems*, vol. 106, no. 1, pp. 5–20, Jan. 1, 2006, ISSN: 0263-5577. DOI: 10.1108/02635570610640979. Accessed: Jan. 11, 2026. [Online]. Available: https://doi.org/10.1108/02635570610640979.

[12] R. S. Baraka and Y. M. Al-Ashqar, "Building a SOA-Based Model for Purchase Order Management in E-Commerce Systems," in *2013 Palestinian International Conference on Information and Communication Technology*, Apr. 2013, pp. 107–114. DOI: 10.1109/PICICT.2013.27. Accessed: Jan. 12, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6545945.

[13] V. Benavente, L. Yantas, I. Moscol, C. Rodriguez, R. Inquilla, and Y. Pomachagua, "Comparative Analysis of Microservices and Monolithic Architecture," in *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, Dec. 2022, pp. 177–184. DOI: 10.1109/CICN56167.2022.10008275. Accessed: Jan. 11, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/10008275.

[14] C. Lin, J. Yao, H. Zhang, D. Ji, and G. Ji, "Research on Shipping E-Commerce Platform based on J2EE and SOA development technique," in *2010 7th International Conference on Service Systems and Service Management*, Jun. 2010, pp. 1–4. DOI: 10.1109/ICSSSM.2010.5530260. Accessed: Jan. 11, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5530260.

[15] L. Xiong-yi, "Research and Application of SOA in B2B Electronic Commerce," in *2009 International Conference on Computer Technology and Development*, vol. 1, Nov. 2009, pp. 649–653. DOI: 10.1109/ICCTD.2009.256. Accessed: Jan. 12, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5359755.

[16] J. Li, "Design of B2B E-commerce Platform Based on SOA Architecture," *IOP Conference Series: Materials Science and Engineering*, vol. 569, no. 3, p. 032 051, Jul. 2019, ISSN: 1757-899X. DOI: 10.1088/1757-899X/569/3/032051. Accessed: Jan. 12, 2026. [Online]. Available: https://doi.org/10.1088/1757-899X/569/3/032051.

[17] *Karrioapi/karrio*, Karrio Shipping, Jan. 9, 2026. Accessed: Jan. 10, 2026. [Online]. Available: https://github.com/karrioapi/karrio.

[18] kevinvenclovas, *Kevinvenclovas/ShippingProAPICollection*, Dec. 10, 2025. Accessed: Jan. 10, 2026. [Online]. Available: https://github.com/kevinvenclovas/ShippingProAPICollection.

[19] "Google SRE monitoring ditributed system - sre golden signals," Accessed: Jan. 12, 2026. [Online]. Available: https://sre.google/sre-book/monitoring-distributed-systems/.

[20] D. SANDU, "Operational and real-time Business Intelligence," *Informatica Economica Journal*, vol. XII, Jan. 1, 2008.

[21] Z. Zheng, J. Liu, and S. Sun, "Real-time big data processing framework: Challenges and solutions," *ResearchGate*, Aug. 8, 2025. DOI: 10.12785/amis/090646. Accessed: Jan. 13, 2026. [Online]. Available: https://www.researchgate.net/publication/282688181_Real-time_big_data_processing_framework_Challenges_and_solutions.

[22] N. Marz. "How to beat the CAP theorem," thoughts from the red planet, Accessed: Jan. 13, 2026. [Online]. Available: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[23] C. Lekkala, "Leveraging Lambda Architecture for Efficient Real-Time Big Data Analytics," vol. 7, pp. 59–64, Feb. 29, 2020. DOI: 10.5281/zenodo.11100539.

[24] Z. Hasani, M. Kon-Popovska, and G. Velinov, "Lambda Architecture for Real Time Big Data Analytic," *ICT Innovations*, 2014.

[25] J. Kreps. "Questioning the Lambda Architecture," O'Reilly Media, Accessed: Jan. 13, 2026. [Online]. Available: https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[26] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Packt Publishing Ltd, Jan. 31, 2018, 357 pp., ISBN: 978-1-78847-120-6. Google Books: oyZKDwAAQBAJ.

[27] J. Piela, "Key performance indicator analysis and dashboard visualization in a logistics company," *Avainmittareiden analysointi ja dashboard visualisointi logistiikan yrityksessä*, 2017. Accessed: Jan. 13, 2026. [Online]. Available: https://lutpub.lut.fi/handle/10024/147689.

[28] Z. Ishak, S. L. Fong, and S. C. Shin, "SMART KPI Management System Framework," in *2019 IEEE 9th International Conference on System Engineering and Technology (ICSET)*, Oct. 2019, pp. 172–177. DOI: 10.1109/ICSEngT.2019.8906478. Accessed: Jan. 14, 2026. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8906478.

[29] K. Savolainen, "Performance dashboard effectiveness improvement to support service delivery within global IT solution provider," M.S. thesis, South-Eastern Finland University of Applied Sciences, 2023. Accessed: Jan. 14, 2026. [Online]. Available: http://www.theseus.fi/handle/10024/810243.

[30] J. Cai, X. Liu, Z. Xiao, and J. Liu, "Improving supply chain performance management: A systematic approach to analyzing iterative KPI accomplishment," *Decision Support Systems*, vol. 46, no. 2, pp. 512–521, Jan. 1, 2009, ISSN: 0167-9236. DOI: 10.1016/j.dss.2008.09.004. Accessed: Jan. 14, 2026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923608001693.