ATELIER MIDDLEWARE & ARCHITECTURE LOGICIELLE

Module : Progiciel et Système d'Intégration

RAPPORT D'ATELIER

Topologies de Middleware d'une Entreprise Engagée

Analyse comparative et cas d'usage

Préparé par

Abdellah Raissouni
Mouad Bencaid
ElKhoumsi Imane
Laraichi Yassine

Encadré par

Mme Besri Zineb

Année universitaire : 2025/2026

Remerciements

Nous tenons à exprimer notre sincère gratitude à **Mme Besri Zineb** pour son encadrement, ses conseils précieux et son accompagnement tout au long de la préparation de cet atelier. Son expertise et son soutien ont été déterminants dans la réalisation de ce travail.

Nous remercions également les auteurs et chercheurs dont les travaux ont servi de fondement à cette étude, notamment Gregor Hohpe, Bobby Woolf, Thomas Erl, et tous les contributeurs des ressources académiques et professionnelles référencées dans ce rapport.

Enfin, nous adressons nos remerciements à nos collègues et à l'ensemble des participants qui ont contribué à enrichir nos échanges et nos réflexions sur les topologies de middleware dans les entreprises engagées.

Les auteurs

Abdellah Raissouni, Mouad Bencaid, ElKhoumsi Imane, Laraichi Yassine

Année universitaire 2025/2026

Table des matières

Table des figures

Liste des tableaux

1 Introduction

1.1 Contexte et Motivation

Dans le paysage technologique actuel, les entreprises font face à un défi majeur : l'intégration de systèmes hétérogènes et distribués. Cette problématique n'est pas nouvelle, mais elle prend une dimension particulière avec l'évolution rapide des technologies et l'émergence de nouveaux paradigmes architecturaux. Le middleware, cette couche logicielle souvent invisible mais essentielle, joue un rôle crucial dans la résolution de ces défis d'intégration.

Une **entreprise engagée**, dans notre contexte, désigne une organisation qui cherche à optimiser ses processus métier en tirant parti des technologies d'intégration modernes. Cet engagement se manifeste par la volonté de connecter efficacement différents systèmes, qu'ils soient internes (Application-to-Application, A2A), externes (Business-to-Business, B2B), ou au sein d'une même application (Intra-Application Integration, IAI).

Les enjeux sont multiples : performance, scalabilité, maintenabilité, mais aussi coût et complexité. Choisir la bonne topologie de middleware n'est donc pas une décision anodine. Elle impacte directement la capacité de l'entreprise à évoluer, à s'adapter aux changements, et à innover rapidement.

1.2 Objectifs de l'Atelier

Cet atelier a pour objectif principal de fournir aux participants une compréhension approfondie des différentes topologies de middleware disponibles, leurs avantages, leurs inconvénients, et les contextes dans lesquels elles sont le plus appropriées. Plus spécifiquement, nous visons à :

— Présenter les concepts fondamentaux du middleware et son évolution historique

— Analyser en détail les principales topologies (Point-à-Point, Hub-and-Spoke, Message Bus)

— Explorer les Enterprise Integration Patterns (EIP) et leur application pratique

— Comparer les approches classiques (SOA/ESB) avec les approches modernes (Microservices, Event-Driven)

— Illustrer par des cas d'usage réels tirés de différents secteurs d'activité

— Permettre aux participants d'appliquer ces concepts à travers un exercice pratique

1.3 Structure du Document

Ce rapport est organisé en plusieurs sections qui reflètent la progression de l'atelier. Après cette introduction, nous présentons les **fondamentaux du middleware** (Section 2), incluant les définitions, l'évolution historique, et les concepts clés. La Section 3 est consacrée aux **topologies de middleware** avec une analyse comparative détaillée. La Section 4 explore les **Enterprise Integration Patterns** et leur application. La Section 5 présente des **cas d'usage réels** et des comparaisons entre différentes approches. Enfin, la Section 6 décrit la **partie pratique** de l'atelier, et la Section 7 conclut avec une synthèse et des perspectives futures.

2 Fondamentaux du Middleware et Architecture d'Intégration

Cette section introduit les concepts fondamentaux du middleware et retrace son évolution dans les architectures d'entreprise. Comprendre ces bases est essentiel pour analyser les différentes topologies présentées dans la suite du rapport.

2.1 Définition et Rôle du Middleware dans une Entreprise Engagée

Le **middleware** est une couche logicielle intermédiaire qui facilite la communication entre applications distribuées et hétérogènes [4, 14]. Concrètement, il permet à un ERP SAP (SOAP/XML) de communiquer avec un CRM Salesforce (REST/JSON) sans modifier ces systèmes.

Pour une entreprise engagée, le middleware apporte trois bénéfices clés :

— **Intégration multi-systèmes** : unification des formats et protocoles (SOAP, REST, EDI) sans réécrire les applications existantes.

— **Agilité métier** : ajout ou remplacement d'un système sans impacter l'ensemble de l'écosystème.

— **Réduction des coûts** : centralisation de la logique d'intégration évite la duplication et limite le nombre de connexions à maintenir.

Ces problématiques se déclinent dans trois grands scénarios d'intégration :

— **A2A (Application-to-Application)** : intégration entre applications internes (par exemple ERP ↔ CRM ↔ système de facturation) ;

— **B2B (Business-to-Business)** : intégration avec des partenaires externes (fournisseurs, banques, transporteurs) via des échanges sécurisés ;

— **IAI (Intra-Application Integration)** : intégration entre modules au sein d'un même progiciel (par exemple, modules d'un ERP qui partagent un bus interne).

2.2 Évolution Historique des Architectures d'Intégration

L'évolution du middleware reflète les défis croissants de complexité rencontrés par les entreprises.

Années 1990–2000 : Point-à-point. Connexions directes entre applications via RPC ou CORBA. Fonctionne pour quelques systèmes, mais la complexité explose rapidement : $n(n-1)/2$ connexions pour n applications. Ingérable au-delà de 5–10 systèmes.

Années 2000–2010 : SOA et ESB. L'**architecture orientée services** (SOA) standardise les services via SOAP/WSDL. L'**Enterprise Service Bus** (ESB) centralise les communications : transformation, routage, sécurité. Principes formalisés par Erl [2] et OASIS SOA [8] : couplage faible, réutilisabilité, composabilité.

Années 2010–2020 : Microservices. Décomposition en services fins, indépendamment déployables [6, 5]. Communication via APIs REST/gRPC. L'*API Gateway* et le *Service Mesh* remplacent partiellement l'ESB centralisé. Le middleware se spécialise et se distribue.

Topologies de Middleware d'une Entreprise Engagée

Années 2020+ : Event-Driven. L'**Event-Driven Architecture** (EDA) traite des flux massifs en temps réel [18]. Kafka [12], RabbitMQ [13], Pulsar deviennent la colonne vertébrale de l'entreprise, diffusant des événements métier (commandes, paiements, alertes).

2.3 Types de Middleware et Rôles Fonctionnels

Quatre grandes familles coexistent dans les entreprises modernes :

— **MOM (Message-Oriented Middleware)** : communication asynchrone via files/topics. Exemples : IBM MQ, RabbitMQ, Kafka. Adapté aux commandes, notifications, logs.

— **RPC** : appels synchrones distants (CORBA, gRPC). Simple à programmer mais couplage temporel fort.

— **Orienté objets** : distribution d'objets (CORBA, DCOM). Moins utilisé aujourd'hui, remplacé par SOA/microservices.

— **Transactionnel** : transactions ACID distribuées (Tuxedo, CICS). Critique pour le secteur bancaire.

Le middleware remplit quatre rôles fonctionnels :

— **Communication** : protocoles (HTTP, AMQP, JMS, gRPC), transport, chiffrement.

— **Transformation** : conversion de formats (XML/JSON/EDI), mapping de structures.

— **Routage** : orientation des messages selon règles métier (Content-Based Router, etc.).

— **Orchestration** : coordination de services pour un processus complet (ex. : validation → paiement → livraison).

2.4 Qualités Essentielles d'un Middleware

Un middleware adapté à une entreprise engagée doit satisfaire cinq qualités :

— **Fiabilité** : garantie de livraison, gestion d'erreurs, retry automatique.

— **Scalabilité** : support de volumes croissants, idéalement via scaling horizontal.

— **Sécurité** : authentification, autorisation, chiffrement, audit trail (conformité PCI-DSS, GDPR, HIPAA).

— **Observabilité** : logs structurés, métriques, traces distribuées, dashboards.

— **Performance** : latence faible, débit élevé pour respecter les SLA métiers.

2.5 Architecture d'Intégration d'Entreprise et Principes SOA

Le modèle OASIS SOA structure l'architecture en couches :

```
Applications métier
    ↓
Services (logique métier)
    ↓
Middleware (ESB / Message Bus)
    ↓
Infrastructure (réseau, sécurité)
```

Topologies de Middleware d'une Entreprise Engagée

Le middleware implémente cinq principes SOA :

— **Couplage faible** : services indépendants, contrats stables (WSDL, OpenAPI).

— **Réutilisabilité** : services partagés (ex. : authentification, facturation).

— **Composabilité** : services simples composés en processus complexes.

— **Autonomie** : chaque service contrôle ses ressources, déploiement indépendant.

— **Statelessness** : pas d'état entre appels, facilite scalabilité et résilience.

Ces principes restent valables dans les architectures modernes (microservices, événements).

3 Topologies de Middleware

Les topologies de middleware définissent comment les systèmes communiquent. Trois approches principales coexistent :

1. **Point-à-point** : connexions directes, adapté aux petits environnements.
2. **Hub-and-Spoke (ESB)** : hub centralisé gérant toutes les communications.
3. **Message Bus** : bus d'événements partagé pour communication asynchrone.

3.1 Topologie Point-à-Point

Dans une topologie point-à-point, chaque application établit des connexions directes avec toutes les autres applications dont elle dépend. Conceptuellement, pour trois applications A, B et C, on obtient :

```
Application A  Application B
Application A  Application C
Application B  Application C
```

Avantages : intuitif, pas de composant central, performances directes. Acceptable pour 2–3 systèmes, utile pour prototyper.

Inconvénients : complexité $O(n^2)$ — pour n applications, $n(n-1)/2$ connexions potentielles. Chaque changement d'interface impacte plusieurs intégrations. Logique de transformation/sécurité dupliquée. Difficilement maintenable au-delà de 5 applications.

3.2 Topologie Hub-and-Spoke (ESB)

La topologie Hub-and-Spoke remplace le maillage complet par un *hub* central : l'ESB. Toutes les applications se connectent à ce hub, qui se charge du routage, de la transformation et de la sécurité. Schématiquement :

```
Application A
Application B
Application C → ESB (Hub)
Application D
Application E
```

Topologies de Middleware d'une Entreprise Engagée

Fonctions de l'ESB : routage intelligent, transformation (SOAP $\leftrightarrow$ REST, XML $\leftrightarrow$ JSON), orchestration, sécurité centralisée, monitoring. Réduction des connexions : n au lieu de $n(n-1)/2$.

Solutions populaires : IBM Integration Bus, WSO2 Enterprise Integrator, MuleSoft Anypoint Platform, Oracle Service Bus. Adaptées aux environnements legacy avec transformation complexe et gouvernance stricte (voir cas d'usage bancaire, Section 5).

Inconvénients : point de défaillance unique, risque de goulot d'étranglement. Nécessite clustering, haute disponibilité, séparation des domaines.

3.3 Topologie Message Bus

La topologie Message Bus repose sur un bus d'événements partagé. Les applications *publient* des messages sur le bus, et d'autres applications s'y *abonnent*. Le bus implémente des canaux de type *queues* ou *topics*, comme dans Kafka ou RabbitMQ :

```
Applications productrices → Message Bus (Kafka / RabbitMQ) → Topics
Applications consommatrices →
```

Caractéristiques : asynchrone, hautement scalable. Producteur et consommateur découplés temporellement, messages persistés. Adapté aux architectures événementielles, temps réel, microservices découplés.

Avantages : performance (traitement parallèle, streaming), scalabilité horizontale, découplage fort.

Inconvénients : cohérence éventuelle[1] (mises à jour pas immédiatement visibles), complexité opérationnelle (clusters, partitions, schémas).

3.4 Topologies Modernes : API Gateway et Service Mesh

Deux composants complémentaires émergent avec les microservices :

— **API Gateway** : façade unique pour clients (web, mobile, partenaires). Gère authentification, rate limiting, agrégation, transformations.

— **Service Mesh** : communication est-ouest entre microservices. Routage, circuit breaking, retry, mTLS, observabilité.

Ces composants complètent les topologies classiques : l'API Gateway peut s'appuyer sur un ESB/Message Bus pour les systèmes internes, le Service Mesh renforce la fiabilité au sein d'un domaine microservices.

3.5 Critères de Choix d'une Topologie

Le choix dépend du contexte :

— **Nombre de systèmes** : quelques applications modernes $\rightarrow$ intégration simple ; dizaines de systèmes legacy $\rightarrow$ ESB.

1. La cohérence éventuelle (eventual consistency) signifie que les mises à jour ne sont pas immédiatement visibles dans tous les systèmes, mais finissent par se propager.

— **Temps réel** : flux synchrones (transactions bancaires) → ESB ; notifications/événements → Message Bus.

— **Gouvernance** : centralisée → Hub-and-Spoke ; équipes autonomes → topologies distribuées.

— **Maturité opérationnelle** : Kafka/microservices nécessitent compétences en observabilité, sécurité, automatisation.

4 Enterprise Integration Patterns

Les **Enterprise Integration Patterns** (EIP), introduits par Hohpe et Woolf [1], forment un vocabulaire commun pour décrire les solutions récurrentes d'intégration. Ils précisent comment les messages circulent, sont transformés et observés dans le middleware.

4.1 Messaging, Routing et Transformation

Messaging Patterns. Les patterns de *messaging* définissent la structure des canaux de communication :

— **Point-to-Point Channel** : un message est consommé par un seul destinataire, typiquement via une file (queue) ; il est utilisé pour distribuer des tâches entre plusieurs workers ;

— **Publish-Subscribe Channel** : un message est diffusé à tous les abonnés d'un canal, comme dans les topics Kafka ; il est adapté aux notifications multi-systèmes ;

— **Message Channel** : concept générique de canal de messages, qu'il soit synchrone (HTTP) ou asynchrone (queue, topic).

Routing Patterns. Les patterns de *routing* permettent d'adapter dynamiquement les trajectoires des messages :

— **Message Router** : route les messages vers différents canaux selon des règles statiques ;

— **Content-Based Router** : route en fonction du contenu du message (type de produit, montant, pays) ;

— **Dynamic Router** : permet de modifier les règles de routage à l'exécution, par exemple en fonction de la charge des services ;

— **Recipient List** : envoie une copie du message à plusieurs destinataires (inventaire, facturation, analytics, CRM dans un contexte e-commerce).

Transformation Patterns. Les patterns de *transformation* traitent des conversions de formats et d'enrichissements :

— **Message Translator** : convertit un message d'un format à un autre (XML → JSON, SOAP → REST) ;

— **Content Enricher** : ajoute des données au message (par exemple, informations client extraites d'un CRM) ;

— **Content Filter** : supprime les champs non nécessaires ou sensibles avant l'envoi à un partenaire externe ;

— **Claim Check** : stocke temporairement le contenu complet du message et ne fait circuler qu'une référence, utile pour les charges volumineuses.

4.2 Endpoints et Gestion Opérationnelle

Endpoint Patterns. Deux patterns structurent la manière dont les applications consomment les messages :

— **Polling Consumer** : l'application interroge périodiquement le canal (par exemple, toutes les cinq minutes). La mise en œuvre est simple, mais la latence et la consommation de ressources peuvent être élevées ;

— **Event-Driven Consumer** : l'application est notifiée en temps réel dès qu'un message arrive (webhook, listener). Cette approche est plus réactive mais demande une architecture plus soignée.

System Management Patterns. Pour exploiter une architecture d'intégration à l'échelle d'une entreprise engagée, la supervision devient critique :

— **Wire Tap** : crée une copie passive des messages à des fins d'audit, de logging ou d'analytics, sans perturber le flux principal ;

— **Message Store** : conserve les messages pour un traitement différé, la reprise après incident ou la traçabilité réglementaire ;

— **Detour** : permet de détourner temporairement un flux vers un environnement de test ou de maintenance, sans interrompre le fonctionnement global.

Dans les sections de cas d'usage, nous montrons comment ces patterns se combinent concrètement avec les topologies ESB et Message Bus pour construire des architectures robustes, observables et conformes aux exigences d'une entreprise engagée.

5 Cas d'Usage Réels et Comparaisons

Cette section présente des cas d'usage concrets illustrant comment les topologies de middleware sont déployées dans des entreprises engagées. Nous analysons trois exemples provenant de secteurs différents : e-commerce, banque, et santé. Chaque cas met en évidence des défis spécifiques et les solutions architecturales choisies, en mettant l'accent sur les topologies de middleware utilisées.

5.1 Cas d'Usage 1 : Plateforme E-commerce - Intégration Multi-Systèmes

5.1.1 Contexte et Défis

Considérons une plateforme e-commerce moderne qui traite **10,000 commandes par jour**, avec des pics à **500 commandes par heure**. Cette entreprise doit intégrer plusieurs systèmes hétérogènes :

— **ERP Legacy (SAP R/3)** : Gestion des stocks et commandes, utilise uniquement **SOAP/XML**

— **CRM Moderne (Salesforce)** : Gestion des clients et campagnes marketing, utilise **REST/-JSON**

— **Système de paiement (Stripe)** : Traitement des paiements en temps réel, **REST/JSON**
— **Service de livraison externe** : Calcul des frais et suivi, **REST/JSON**, latence variable
— **Système de notifications** : Emails, SMS, push notifications, **REST/JSON**

Les défis principaux sont :

— **Transformation de protocoles** : SAP utilise SOAP, tous les autres systèmes utilisent REST
— **Volume élevé** : 10,000+ commandes/jour nécessitant une scalabilité horizontale
— **Disponibilité critique** : 99.9% de disponibilité requise
— **Résilience** : Le service de livraison peut être indisponible, nécessitant des mécanismes de fallback

5.1.2 Architecture Proposée

L'architecture choisie est une **architecture hybride ESB + Message Bus**, illustrée dans la Figure 1.

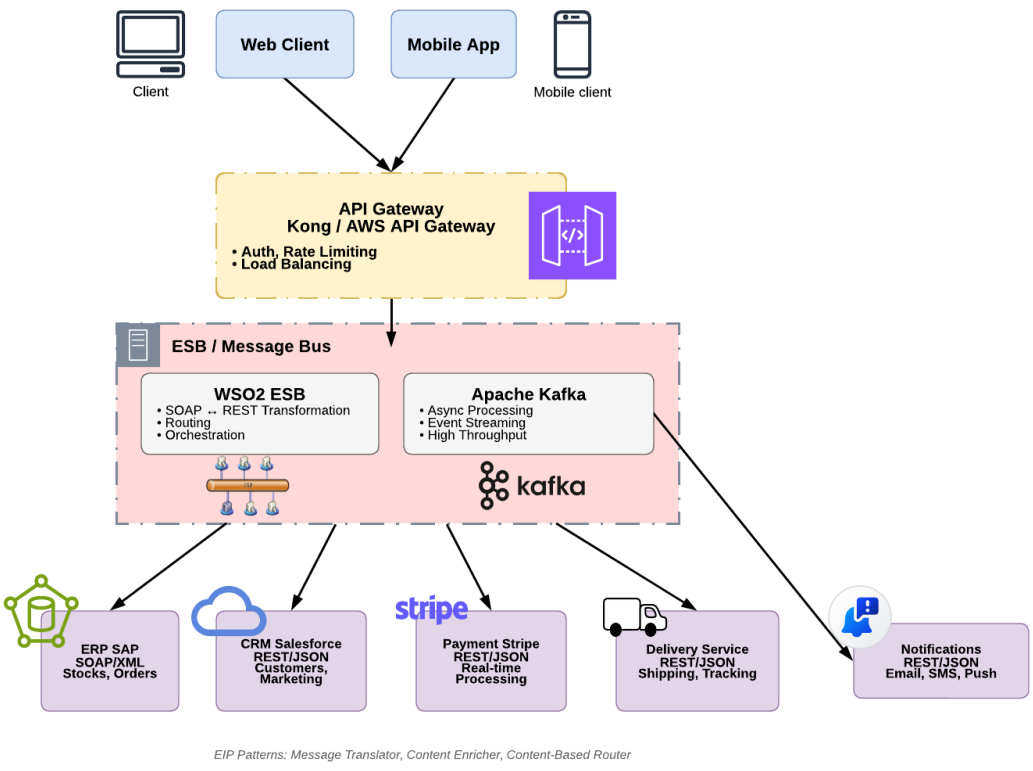

FIGURE 1 – Architecture E-commerce : Topologie hybride ESB + Message Bus

Explication de l'architecture :
L'architecture se compose de plusieurs couches :

1. **Couche Client** : Applications web et mobiles accèdent via un **API Gateway** (Kong ou AWS API Gateway) qui sert de point d'entrée unique, gérant la sécurité (OAuth 2.0), le rate limiting, et la terminaison SSL/TLS.

2. **Couche Middleware Hybride** :
   — **ESB (WSO2 Enterprise Integrator)** : Gère la transformation critique SOAP/XML REST/JSON en temps réel. L'ESB orchestre également les appels synchrones nécessaires (validation du stock, traitement du paiement).
   — **Message Bus (Apache Kafka)** : Traite les commandes de manière asynchrone, permettant de découpler le processus et d'assurer une scalabilité élevée. Kafka permet également la réplication et la persistance des messages.
3. **Couche Systèmes Métier** : Les différents systèmes (ERP SAP, CRM Salesforce, Stripe, service de livraison, notifications) sont connectés via l'ESB ou le Message Bus selon leurs besoins (synchrone vs asynchrone).

5.1.3 Patterns Enterprise Integration Patterns (EIP) Utilisés

Plusieurs patterns EIP sont appliqués dans cette architecture :

— **Message Translator** : Transformation SOAP/XML REST/JSON pour l'intégration avec SAP
— **Content Enricher** : Enrichissement de la commande avec les données client depuis le CRM Salesforce
— **Content-Based Router** : Routage des commandes selon le type de produit (électronique, vêtements, etc.) vers les bons canaux de traitement
— **Publish-Subscribe** : Notification simultanée de plusieurs services (inventaire, facturation, analytics, CRM) lors de la création d'une commande
— **Circuit Breaker** : Protection contre les pannes du service de livraison externe, avec fallback vers des données en cache

5.1.4 Justification des Choix Architecturaux

Le choix d'une architecture hybride ESB + Message Bus est justifié par :

— **ESB pour la transformation** : La transformation SOAP/REST est complexe et nécessite une logique centralisée. L'ESB excelle dans ce type de transformation.
— **Message Bus pour la scalabilité** : Le traitement asynchrone des commandes via Kafka permet de gérer les pics de charge sans bloquer les autres opérations.
— **Séparation des responsabilités** : L'ESB gère les opérations synchrones critiques (paiement), tandis que Kafka gère les opérations asynchrones (notifications, analytics).

5.2 Cas d'Usage 2 : Banque - Intégration Systèmes Legacy avec APIs Modernes

5.2.1 Contexte et Défis

Considérons une grande banque qui doit exposer des APIs modernes pour les applications mobiles et les partenaires, tout en préservant ses systèmes legacy critiques. Cette banque possède :

— **Systèmes legacy** :
  — IBM Mainframe (Z/OS) avec COBOL et CICS
  — AS/400 avec RPG et CICS
  — Base de données Core Banking (DB2, IMS)
— **Applications modernes** : Mobile banking, web banking, APIs pour partenaires
— **Exigences réglementaires** : Conformité PCI-DSS (paiements), GDPR (données personnelles)

Les défis principaux sont :

— **Protocoles propriétaires** : Les systèmes legacy utilisent COBOL, CICS, et le protocole 3270, incompatibles avec REST/JSON moderne
— **Performance** : Les systèmes legacy peuvent être lents (5-10 secondes de réponse)
— **Fiabilité** : Les systèmes legacy doivent rester disponibles, mais peuvent subir des pannes
— **Sécurité** : Conformité réglementaire stricte nécessitant un audit trail complet

5.2.2 Architecture Proposée

L'architecture choisie est une **topologie Hub-and-Spoke avec ESB**, illustrée dans la Figure 2.

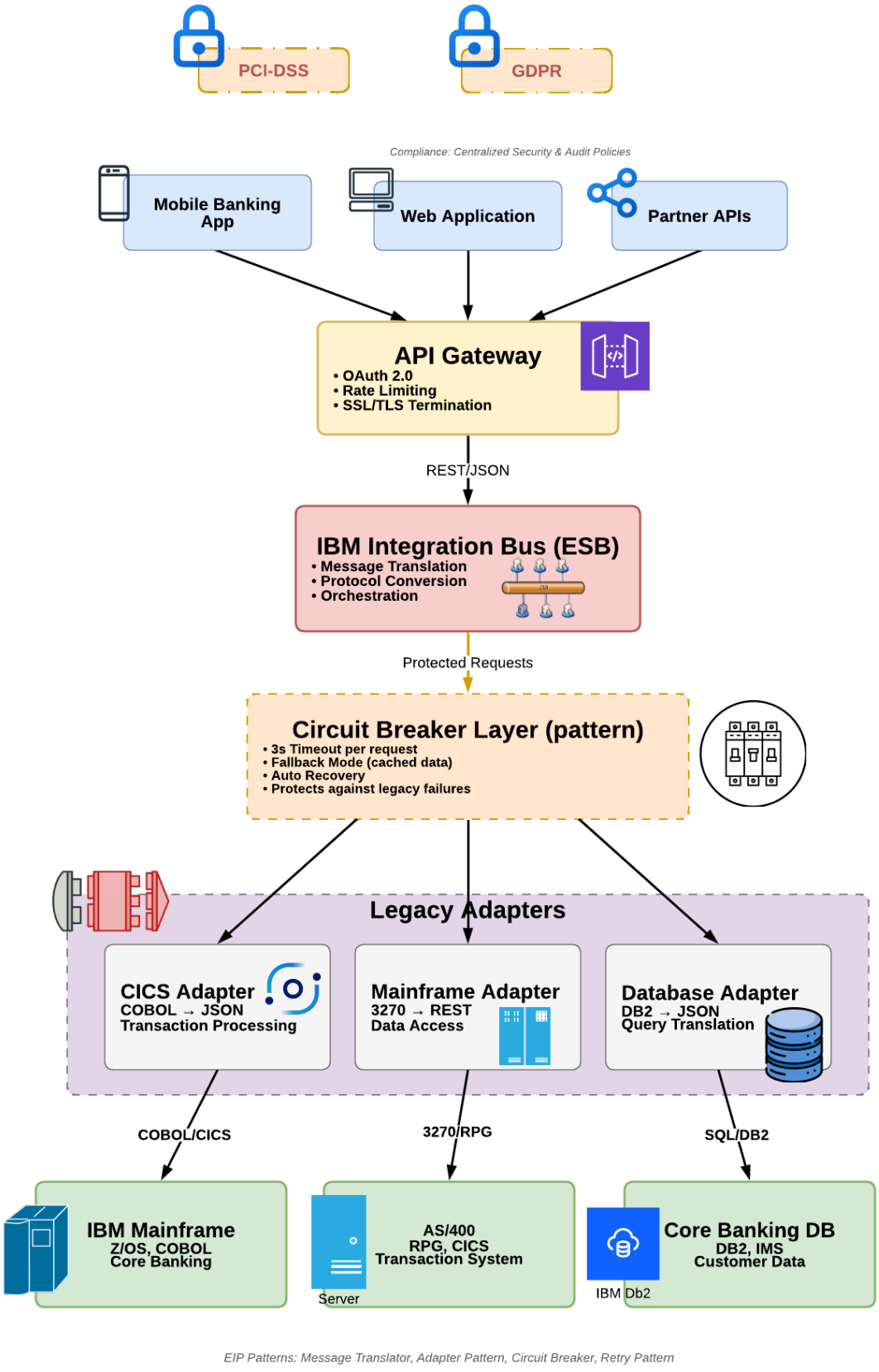

FIGURE 2 – Architecture Banque : Topologie Hub-and-Spoke avec ESB et Circuit Breaker

Explication de l'architecture :
L'architecture se compose de plusieurs couches, de haut en bas :

1. **Couche Conformité (Métadonnées)** : En haut du diagramme, les badges **PCI-DSS** et **GDPR** indiquent la conformité de l'architecture entière. Ces badges sont des métadonnées visuelles, non des composants connectés, représentant la gouvernance centralisée de la sécurité et de l'audit.
2. **Couche Client** : Trois types de clients accèdent au système :
   — Applications mobiles
   — Applications web
   — APIs pour partenaires externes
3. **API Gateway** : Point d'entrée unique gérant :
   — Authentification OAuth 2.0
   — Rate limiting pour protéger contre les abus
   — Terminaison SSL/TLS
4. **ESB (IBM Integration Bus)** : Cœur de l'architecture, le hub central qui :
   — Centralise toutes les intégrations (topologie Hub-and-Spoke)
   — Effectue la transformation de protocoles (COBOL JSON, 3270 REST)
   — Gère l'orchestration des appels synchrones
   — Centralise la gouvernance et les politiques de sécurité
5. **Couche Circuit Breaker** : **Critique** - Positionnée entre l'ESB et les adapters, cette couche :
   — Surveille chaque requête vers les systèmes legacy
   — Si un mainframe ne répond pas dans les 3 secondes, le circuit s'ouvre
   — Retourne immédiatement des données en cache ou un message d'erreur gracieux
   — Protège les applications modernes contre les pannes des systèmes legacy
   — Permet la récupération automatique une fois le système legacy disponible
6. **Couche Adapters** : Adapters spécialisés pour chaque système legacy :
   — **CICS Adapter** : Traduit COBOL/CICS vers JSON
   — **Mainframe Adapter** : Convertit le protocole 3270 vers REST
   — **Database Adapter** : Transforme les requêtes DB2/IMS vers JSON
7. **Systèmes Legacy** : Les systèmes existants (Mainframe, AS/400, Core Banking DB) restent inchangés, préservant les investissements existants.

5.2.3 Patterns EIP Utilisés

— **Message Translator** : Transformation COBOL JSON, 3270 REST, DB2 JSON

— **Adapter Pattern** : Adapters spécialisés pour chaque système legacy, masquant leur complexité

— **Circuit Breaker** : Protection contre les pannes des systèmes legacy avec fallback et récupération automatique

— **Retry Pattern** : Retry automatique en cas d'échec temporaire, avec backoff exponentiel

5.2.4 Justification de la Topologie Hub-and-Spoke

Pourquoi Hub-and-Spoke et pas Message Bus ici ?

— **Systèmes legacy synchrones** : Les transactions bancaires doivent être validées immédiatement. Un Message Bus asynchrone ne conviendrait pas pour ces opérations critiques.

— **Transformation complexe** : L'ESB centralise la transformation de multiples protocoles propriétaires, évitant la duplication de logique.

— **Gouvernance centralisée** : Les exigences réglementaires (PCI-DSS, GDPR) nécessitent un contrôle centralisé de la sécurité et de l'audit, facilité par l'ESB.

— **Point de défaillance acceptable** : Dans le secteur bancaire, la centralisation de la gouvernance est préférée à la complexité distribuée, même si cela crée un point de défaillance unique (mitigé par le clustering de l'ESB).

5.3 Cas d'Usage 3 : Hôpital - Interopérabilité entre Systèmes Hospitaliers

5.3.1 Contexte et Défis

Considérons un hôpital moderne qui doit intégrer plusieurs systèmes hospitaliers utilisant différents standards médicaux :

— **Système d'Information Hospitalier (SIH)** : Dossier patient électronique

— **Laboratoires** : Résultats d'analyses médicales

— **Pharmacie** : Gestion des médicaments et interactions

— **Imagerie médicale** : PACS (Picture Archiving and Communication System)

— **Partenaire externe** : Assurance maladie pour le remboursement

Les défis principaux sont :

— **Multiples standards** : HL7 v2 (ancien), HL7 v3, FHIR (moderne), DICOM (imagerie)

— **Interopérabilité** : Transformation entre différents standards de données médicales

— **Sécurité** : Données sensibles nécessitant conformité HIPAA (USA) et GDPR (Europe)

— **Temps réel** : Certaines données critiques (urgences) nécessitent un traitement immédiat

— **Notifications multi-systèmes** : Un résultat de laboratoire doit notifier simultanément le SIH, la pharmacie, et l'assurance

5.3.2 Architecture Proposée

L'architecture choisie est une **topologie Message Bus**, illustrée dans la Figure 3.

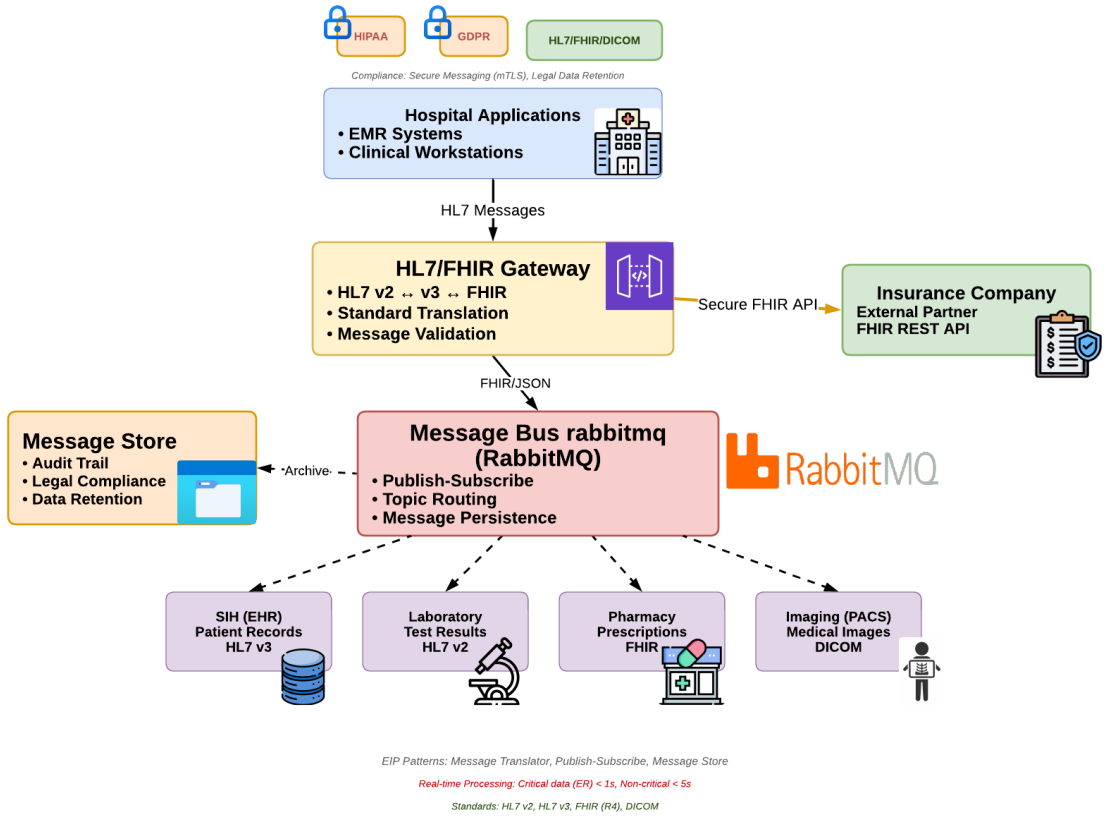

FIGURE 3 – Architecture Santé : Topologie Message Bus avec transformation HL7/FHIR

Explication de l'architecture :
L'architecture se compose de plusieurs couches :

1. **Couche Conformité (Métadonnées)** : En haut, les badges **HIPAA**, **GDPR**, et les standards **HL7/FHIR/DICOM** indiquent la conformité de l'architecture. Ces badges sont des métadonnées visuelles, non des composants connectés.
2. **Couche Applications Hospitalières** : Les systèmes EMR (Electronic Medical Records) et postes cliniques publient des messages au format HL7.
3. **HL7/FHIR Gateway** : Transforme les différents standards :
   — HL7 v2 → HL7 v3
   — HL7 v3 → FHIR (Fast Healthcare Interoperability Resources)
   — FHIR est le standard moderne basé sur REST/JSON
4. **Message Bus (RabbitMQ)** : Cœur de l'architecture, le bus d'événements qui :
   — Permet la communication asynchrone et découplée
   — Supporte le pattern **Publish-Subscribe** : un message peut être consommé par plusieurs systèmes simultanément
   — Persiste les messages pour la récupération en cas de panne
5. **Message Store** : Archive tous les messages pour :

— Conformité légale (archivage des données médicales)

— Traçabilité complète (audit trail)

— Récupération en cas de perte

6. **Systèmes Hospitaliers** : Les différents systèmes (SIH, Laboratoires, Pharmacie, Imagerie, Assurance) s'abonnent aux messages pertinents via le Message Bus.

5.3.3 Patterns EIP Utilisés

— **Message Translator** : Transformation HL7 v2 HL7 v3 FHIR pour l'interopérabilité entre systèmes

— **Publish-Subscribe** : Notification simultanée de plusieurs systèmes lorsqu'un laboratoire publie de nouveaux résultats. Le SIH, la pharmacie (pour vérifier les interactions médicamenteuses), et l'assurance maladie (pour le remboursement) sont tous notifiés automatiquement.

— **Message Store** : Archivage des données médicales pour la conformité légale et la traçabilité

5.3.4 Justification de la Topologie Message Bus

Pourquoi Message Bus et pas ESB ici ?

— **Communication asynchrone** : Les notifications médicales n'ont pas besoin d'être traitées immédiatement. Un résultat de laboratoire peut être traité quelques secondes ou minutes plus tard sans impact critique (sauf urgences).

— **Multi-cast** : Le pattern Publish-Subscribe est essentiel. Un message (nouveau résultat) doit être diffusé à plusieurs systèmes simultanément. L'ESB nécessiterait une orchestration séquentielle, tandis que le Message Bus permet la notification simultanée et découplée.

— **Découplage** : Les systèmes hospitaliers sont indépendants. Le laboratoire ne connaît pas quels systèmes vont recevoir le message - c'est le Message Bus qui route selon les abonnements.

— **Scalabilité** : Facile d'ajouter de nouveaux systèmes (nouveaux laboratoires, nouvelles pharmacies) sans modifier les systèmes existants.

5.4 Comparaison SOA vs Microservices

Cette section compare deux paradigmes architecturaux majeurs : **SOA avec ESB** versus **Microservices** [17]. C'est un choix stratégique qui impacte toute l'architecture d'une entreprise engagée.

5.4.1 Architecture Visuelle

La Figure 4 illustre visuellement les différences architecturales entre SOA/ESB et Microservices.

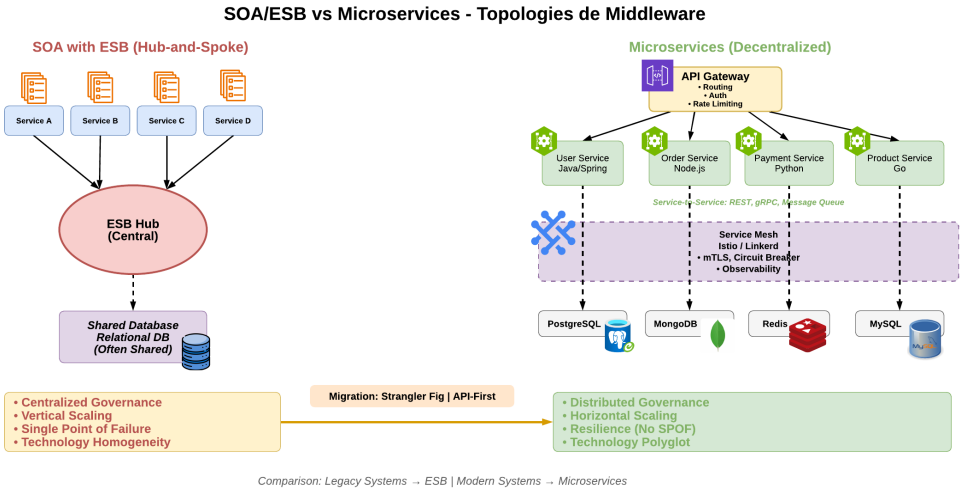

FIGURE 4 – Comparaison Architecturale : SOA/ESB (Hub-and-Spoke) vs Microservices (Décentralisé)

Explication de la comparaison :

— **Côté SOA/ESB (gauche)** : Architecture centralisée où tous les services communiquent via le bus d'entreprise. Les services partagent souvent une base de données commune. La gouvernance est centralisée dans l'ESB.

— **Côté Microservices (droite)** : Architecture décentralisée où chaque microservice est indépendant, avec sa propre base de données. La communication se fait via REST/gRPC, avec un API Gateway en point d'entrée et un Service Mesh pour la gestion du trafic inter-services.

— **Flèche de migration** : Indique les stratégies de migration possibles (Strangler Fig Pattern, API-First Approach, Hybrid Approach).

5.4.2 Tableau Comparatif Détaillé

Le Tableau 1 présente une comparaison systématique des deux approches.

TABLE 1 – Comparaison Détaillée : SOA/ESB vs Microservices

Critère	SOA (ESB)	Microservices
Architecture	Centralisée (Hub-and-Spoke)	Décentralisée (Service Mesh)
Taille des services	Services moyens/grands	Services très petits (single responsibility)
Communication	SOAP, REST, Messages via ESB	REST, gRPC, Messages (direct)
Gouvernance	Centralisée (ESB)	Distribuée (API Gateway + Service Mesh)
Déploiement	Monolithique ou modulaire	Indépendant par service
Technologie	Souvent homogène	Hétérogène (polyglot)
Base de données	Souvent partagée	Base de données par service
Complexité	Complexité centralisée	Complexité distribuée
Scalabilité	Scalabilité verticale	Scalabilité horizontale
Résilience	Point de défaillance unique (ESB)	Résilience distribuée

Note : Les cellules colorées en rouge clair indiquent les caractéristiques de SOA/ESB, tandis que les cellules en vert clair indiquent celles des Microservices.

5.4.3 Quand Utiliser SOA/ESB ?

Avantages de SOA/ESB :

— **Intégration de systèmes legacy** : Excellente pour intégrer de nombreux systèmes legacy avec des protocoles propriétaires

— **Transformation complexe** : Facilite la transformation entre différents protocoles (SOAP REST, EDI XML)

— **Gouvernance centralisée** : Contrôle centralisé des politiques de sécurité, routage, et transformation

— **Équipe centralisée** : Adapté aux organisations avec une équipe centralisée dédiée à l'intégration

Cas d'usage appropriés :

— Entreprises avec beaucoup de systèmes legacy (mainframe, AS/400)

— Intégration B2B complexe avec de nombreux partenaires

— Environnements avec exigences de conformité strictes (banque, assurance, santé)

— Organisations préférant une gouvernance centralisée

Inconvénients :

— Point de défaillance unique (l'ESB)

— Goulot d'étranglement potentiel

— Vendor lock-in possible

— Scalabilité limitée (souvent verticale)

5.4.4 Quand Utiliser Microservices ?

Avantages de Microservices :

— **Scalabilité indépendante** : Chaque service peut être scalé indépendamment selon ses besoins

— **Déploiement indépendant** : Permet des releases fréquentes et rapides

— **Polyglot** : Chaque service peut utiliser la technologie optimale pour son domaine

— **Résilience** : Pas de point de défaillance unique, résilience distribuée

Cas d'usage appropriés :

— Applications cloud-native

— Besoin de scalabilité élevée (millions d'utilisateurs)

— Équipes autonomes (DevOps, équipes par domaine)

— Innovation rapide requise

— Systèmes modernes sans legacy

Inconvénients :

— Complexité opérationnelle (orchestration, monitoring, tracing)

— Gestion de la cohérence distribuée (Saga pattern, Event Sourcing)

— Network latency entre services

— Debugging difficile (tracing distribué nécessaire)

5.4.5 Stratégies de Migration

Pour les entreprises avec des systèmes legacy souhaitant migrer vers les microservices, plusieurs stratégies existent :

1. **Strangler Fig Pattern** : Envelopper progressivement le monolithe avec des microservices, migrer fonctionnalité par fonctionnalité. Exemple : migrer d'abord le service de paiement, puis le service de commande.
2. **API-First Approach** : Exposer d'abord des APIs pour les fonctionnalités, puis migrer progressivement vers des microservices. Permet une transition en douceur.
3. **Hybrid Approach** : Garder l'ESB pour les systèmes legacy, utiliser des microservices pour les nouvelles fonctionnalités, avec un API Gateway pour unifier l'accès. C'est souvent la meilleure approche pour les entreprises engagées.

5.5 Middleware dans l'Économie des APIs (API Economy)

5.5.1 Le Nouveau Paradigme

Les APIs ne sont plus seulement des interfaces techniques. Elles sont devenues des **produits à part entière** qui génèrent de la valeur - c'est l'**API Economy**. C'est un changement de paradigme fondamental : l'intégration n'est plus un coût, c'est un **actif business**.

Exemples concrets :

— **Stripe** : API de paiement générant des milliards de dollars de revenus

— **Twilio** : API de communication (SMS, voix) utilisée par des centaines de milliers d'entreprises

— **AWS** : Tous leurs services cloud exposés via des APIs

— **Google Maps API** : Utilisé par des millions d'applications

Ces entreprises ont transformé leurs services techniques en **produits monétisables**.

5.5.2 Rôle du Middleware dans l'API Economy

Le middleware joue un rôle crucial de **catalyseur** dans cette transformation :

1. **Standardisation de l'Exposition** :
   — Transformation de services hétérogènes (legacy COBOL, microservices, bases de données) en APIs uniformes et consommables
   — Le middleware masque la complexité technique interne

— Permet d'exposer des services legacy comme des APIs modernes

2. **Masquage de la Complexité Technique** :
   — Transformation de protocoles, sécurité, monitoring
   — Permet aux développeurs externes de consommer les APIs facilement sans connaître l'architecture interne
   — Gère la versioning, la compatibilité, et l'évolution des APIs
3. **Mesure et Monétisation** :
   — Analytics d'utilisation, facturation par appel, gestion des quotas
   — Essentiel pour transformer une API technique en produit business
   — Permet la création de modèles de revenus récurrents

5.5.3 Impact pour l'Entreprise Engagée

L'API Economy ouvre de nouveaux modèles économiques :

— **Partenariats via APIs** : Facilite les intégrations avec des partenaires, créant des écosystèmes
— **Écosystèmes de développeurs** : Attire des développeurs externes qui créent de la valeur avec vos APIs
— **Revenus récurrents** : Création de revenus récurrents à partir de services techniques existants
— **Innovation accélérée** : Les APIs permettent une innovation plus rapide en permettant la réutilisation de services

Le middleware devient ainsi un **catalyseur de transformation business**, pas seulement technique. Il permet de transformer l'infrastructure technique en actifs business monétisables.

6 Partie Pratique

6.1 Objectifs pédagogiques et positionnement dans l'atelier

La partie pratique de l'atelier a pour vocation de **mettre en situation concrète** les concepts théoriques présentés dans les sections précédentes (topologies de middleware, patterns d'intégration, API Gateway, etc.). Elle s'appuie sur deux supports distribués séparément au professeur :

— un **Guide des Prérequis** détaillant la préparation de l'environnement (installation de Docker, Git, clonage des dépôts, construction des images) ;
— un **Guide de Travaux Pratiques** décrivant pas à pas la démonstration sur les architectures microservices avec et sans middleware.

L'objectif principal de cette partie est double :

— illustrer, sur un cas d'étude e-commerce réaliste, **l'impact des choix de topologie de middleware** sur la scalabilité, la résilience et la maintenabilité ;

— permettre aux participants de **relier les notions abstraites** (ESB, Message Bus, API Gateway, patterns EIP) à une expérience pratique observable en temps réel.

Cette partie pratique s'inscrit ainsi comme un **complément expérimental** au rapport de recherche sur l'exposé, en vérifiant empiriquement certains résultats théoriques discutés dans les sections 2 à 5.

6.2 Description du cas d'étude et environnement expérimental

6.2.1 Contexte de l'entreprise simulée

Le cas d'étude met en scène une plateforme d'e-commerce, notée **TechStore**, qui vend des produits électroniques en ligne. L'entreprise est passée d'une application monolithique à une architecture microservices afin de mieux absorber la croissance de la charge et d'améliorer sa tolérance aux pannes. La Figure 5 présente l'architecture monolithique de départ utilisée comme point de comparaison.

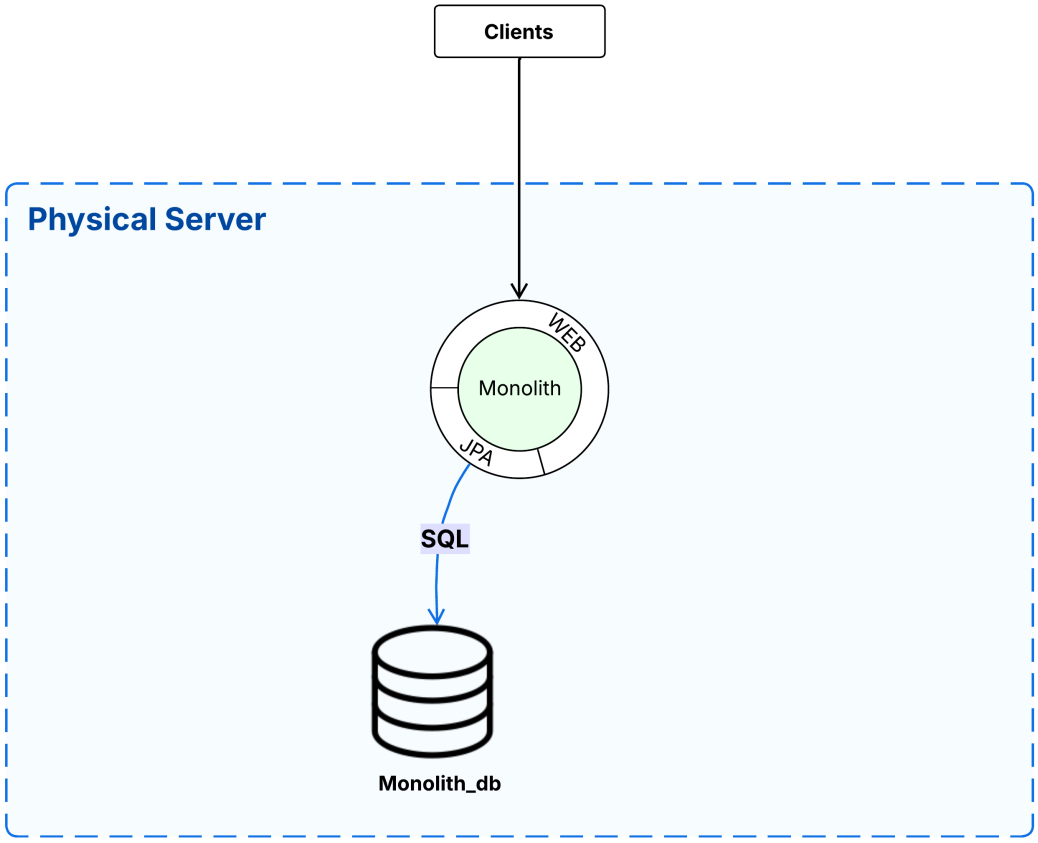

FIGURE 5 – Architecture monolithique de référence utilisée dans la partie pratique

Dans la version microservices, le système est décomposé en plusieurs services métier (gestion des produits, gestion des commandes, etc.) qui communiquent entre eux via des APIs REST. Les caractéristiques de charge (volume de commandes, disponibilité attendue, contraintes de latence) sont paramétrées pour refléter le contexte d'une **entreprise engagée** telle que décrite en introduction du rapport.

6.2.2 Environnement technique

L'environnement expérimental repose sur :

— l'orchestration de conteneurs Docker pour simuler plusieurs instances de services ;

— deux projets distincts : une architecture **sans API Gateway** et une architecture **avec API Gateway** jouant le rôle de middleware d'entrée ;

— un jeu de commandes standardisé (Docker Compose) permettant de démarrer, scaler et arrêter les différents scénarios pendant la séance.

Le **Guide des Prérequis** structure la préparation en plusieurs étapes (installation, vérification de Docker et Git, clonage des dépôts, construction des images). Il garantit que le temps de la séance est consacré à l'analyse architecturale plutôt qu'à la résolution de problèmes d'environnement.

6.3 Déroulement de la démonstration pratique

La démonstration est construite en deux temps, afin de **contraster de manière contrôlée** une architecture microservices non régulée et une architecture régulée par un middleware de type API Gateway.

6.3.1 Phase 1 : microservices sans middleware d'entrée

Dans un premier temps, les participants déploient une architecture microservices où les services (*Products*, *Orders*, etc.) sont directement exposés au client. La Figure 6 illustre la topologie étudiée.

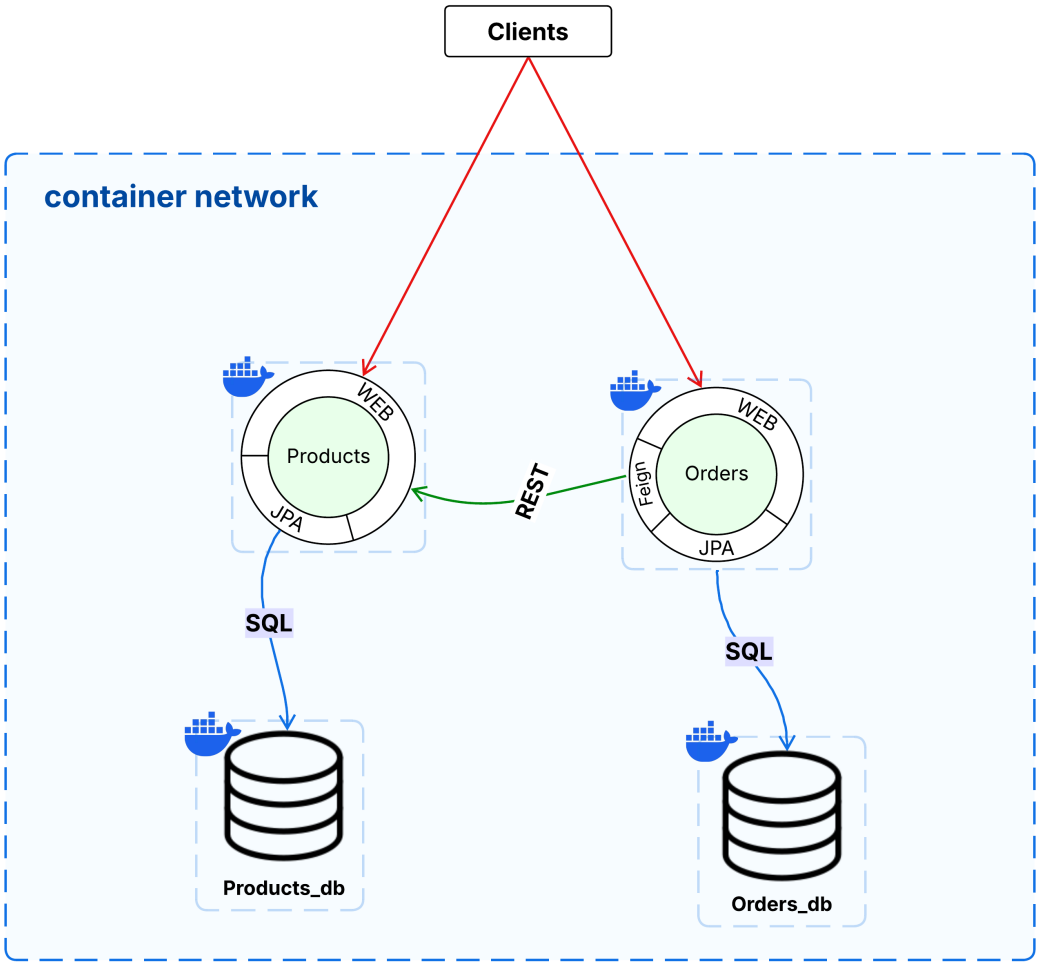

FIGURE 6 – Architecture microservices sans middleware (API Gateway) étudiée dans la phase 1

À partir de ce scénario, plusieurs limitations typiques d'une architecture distribuée non régulée sont mises en évidence :

— **dépendance forte du client aux adresses des services** (ports, IPs, topologie réseau) ;

— **absence de mécanisme centralisé de répartition de charge**, rendant inefficace le *scaling* horizontal ;

— **instabilité des adresses** (IPs éphémères des conteneurs) rendant fragile le couplage inter-services ;

— **duplication des préoccupations transversales** (authentification, logging, etc.) dans chaque microservice.

Ces observations permettent de matérialiser, sur un cas concret, plusieurs risques déjà discutés théoriquement dans la comparaison SOA / microservices et dans la section consacrée à l'économie des APIs.

6.3.2 Phase 2 : microservices régulés par une API Gateway

Dans un second temps, les mêmes services métier sont déployés derrière une **API Gateway** qui joue le rôle de middleware d'entrée unique. La Figure 7 présente la nouvelle topologie.

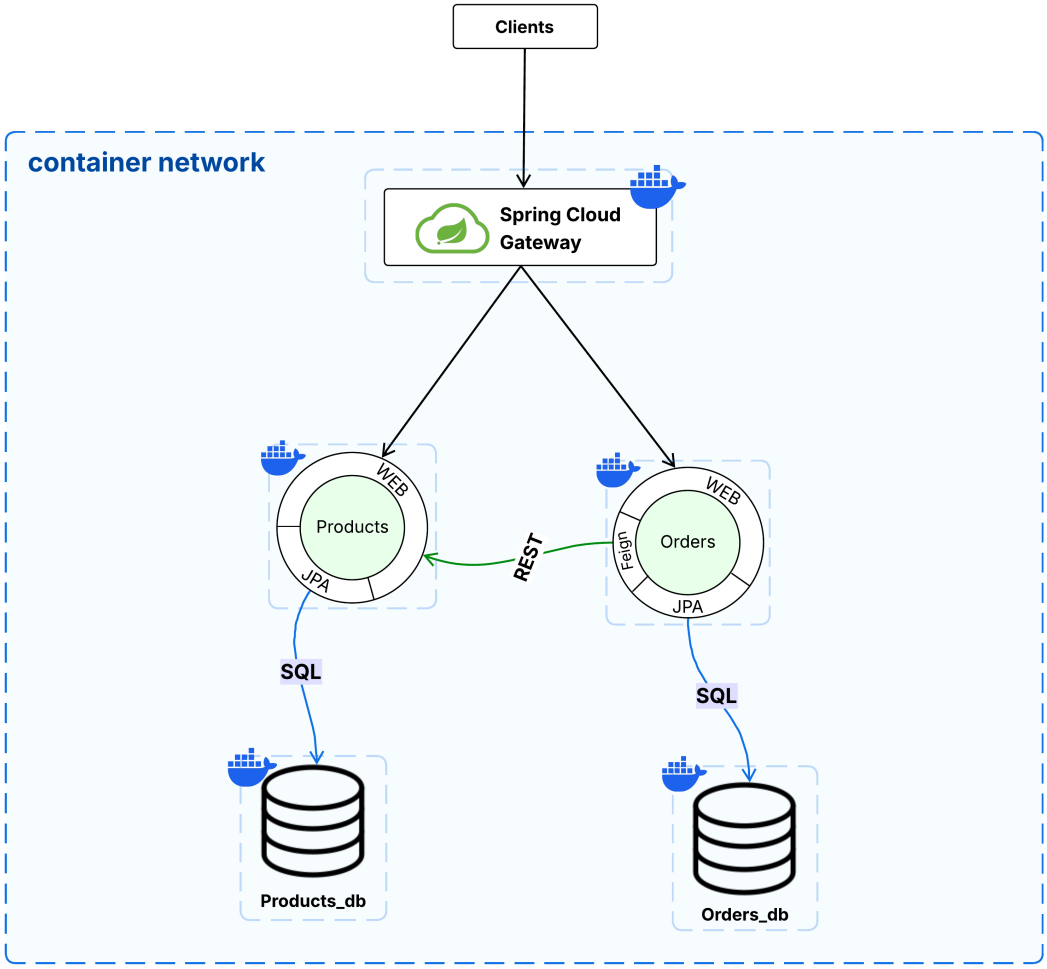

FIGURE 7 – Architecture microservices régulée par un middleware de type API Gateway

Cette deuxième configuration met en évidence plusieurs apports du middleware :

— **point d'entrée unique** pour tous les clients, ce qui casse le couplage direct aux microservices ;

— **répartition de charge transparente** vers plusieurs instances d'un même service (scalabilité horizontale effective) ;

— **centralisation des préoccupations transversales** (authentification, contrôle de flux, journalisation) au niveau de la Gateway ;

— **amélioration de la posture de sécurité** en isolant les services métier dans un réseau interne non exposé.

D'un point de vue patterns, cette phase illustre concrètement l'utilisation d'une façade d'entrée (*API Gateway pattern*) combinée à des capacités de routage, de *load balancing* et de gestion des *cross-cutting concerns*.

6.4 Synthèse des apprentissages de la partie pratique

La partie pratique permet de valider expérimentalement plusieurs conclusions du rapport :

— dans une architecture microservices réelle, **l'absence de middleware** conduit rapidement à une complexité opérationnelle excessive (gestion des adresses, scalabilité manuelle, duplication des mécanismes de sécurité) ;

— l'introduction d'un **middleware d'entrée** (ici une API Gateway) améliore significativement le **découplage**, la **résilience** et la **gouvernance** du système, au prix d'un composant supplémentaire à administrer ;

— le choix d'une topologie de middleware doit être pensé **en cohérence avec les objectifs métiers** (expérience client, agilité, contraintes réglementaires) et non uniquement sous un angle technique.

Les deux documents annexes (*Guide des Prérequis* et *Guide de Travaux Pratiques*) complètent ce rapport en fournissant, respectivement, les détails d'implémentation et le déroulé pédagogique complet de la séance. Ils constituent ainsi le **support opérationnel** de la partie pratique, tandis que le présent rapport en propose une **lecture académique et synthétique**.

7 Conclusion

Cet atelier a exploré les différentes topologies de middleware et leur application dans les entreprises engagées. À travers l'analyse théorique des fondamentaux, des topologies et des patterns d'intégration, complétée par trois cas d'usage concrets (e-commerce, banque, santé), plusieurs enseignements clés émergent.

7.1 Synthèse des Apprentissages

Il n'existe pas de solution universelle. Le choix d'une topologie dépend du contexte spécifique : nombre et nature des systèmes (legacy vs modernes), volume de données, contraintes de temps réel, niveau de gouvernance, et maturité opérationnelle. Comme l'illustrent nos cas d'usage, une banque avec des systèmes legacy choisira un ESB centralisé, tandis qu'un hôpital privilégiera un Message Bus pour les notifications asynchrones.

Les topologies classiques restent pertinentes. Le Point-à-Point convient aux petits environnements (2–3 systèmes). L'ESB excelle pour l'intégration de systèmes legacy avec transformation complexe et gouvernance centralisée. Ces approches ne sont pas obsolètes—elles répondent à des besoins spécifiques que les architectures modernes ne couvrent pas toujours.

Les approches modernes offrent de nouvelles capacités. Le Message Bus (Kafka, RabbitMQ) apporte scalabilité horizontale et découplage temporel, essentiels pour les architectures événementielles. Le Service Mesh renforce la fiabilité des microservices. Cependant, cette puissance s'accompagne d'une complexité opérationnelle accrue nécessitant des compétences spécialisées.

Les Enterprise Integration Patterns constituent un langage commun. Les patterns de Hohpe et Woolf [1] fournissent un vocabulaire standardisé pour décrire les solutions récurrentes. Message Translator, Content Enricher, Circuit Breaker—ces patterns se combinent avec les topologies pour construire des architectures robustes et observables.

L'architecture hybride est souvent optimale. Comme le montre le cas e-commerce, combiner ESB (pour transformation synchrone) et Message Bus (pour traitement asynchrone) permet de bénéficier des avantages de chaque approche. Cette flexibilité est cruciale pour les entreprises avec des besoins hétérogènes.

7.2 Critères de Choix d'une Topologie

Le choix d'une topologie doit s'appuyer sur une analyse systématique de plusieurs critères :

— **Nombre de systèmes** : moins de 5 systèmes → Point-à-Point acceptable ; plus de 10 systèmes → ESB ou Message Bus nécessaire pour éviter la complexité $O(n^2)$.

— **Types de systèmes** : systèmes legacy nombreux (mainframe, AS/400) → ESB pour transformation ; systèmes modernes cloud-native → Message Bus ou Service Mesh.

— **Volume et performance** : faible volume → ESB suffit ; très haut volume (millions de messages/jour) → Message Bus (Kafka) pour scalabilité horizontale.

— **Contraintes temporelles** : flux synchrones (transactions bancaires) → ESB ; notifications et événements → Message Bus asynchrone.

— **Gouvernance** : gouvernance centralisée → Hub-and-Spoke (ESB) ; équipes autonomes → topologies distribuées (Microservices, Service Mesh).

— **Budget et compétences** : solutions open source (Kafka, RabbitMQ, WSO2) vs solutions enterprise (IBM, MuleSoft) ; compétences opérationnelles requises pour exploiter les clusters distribués.

Ces critères ne sont pas indépendants—une décision architecturale résulte d'un compromis entre ces différents facteurs, comme l'illustrent nos cas d'usage.

7.3 Perspectives Futures

Le domaine du middleware évolue rapidement, porté par l'essor du cloud, de l'IoT, et de l'intelligence artificielle. Plusieurs tendances émergent qui transformeront les architectures d'intégration :

7.3.1 Serverless Integration

Les Functions as a Service (AWS Lambda, Azure Functions, Google Cloud Functions) permettent de créer des intégrations sans gérer d'infrastructure. Cette approche « pay-per-use » est particulièrement adaptée aux intégrations avec volume variable ou aux besoins ponctuels. Le middleware devient alors une collection de fonctions déclenchées par événements, réduisant les coûts d'infrastructure.

7.3.2 Edge Computing

Le traitement à la périphérie du réseau réduit la latence et permet des intégrations distribuées. Cette approche est cruciale pour l'IoT, les applications temps réel, et les cas d'usage nécessitant une réactivité locale (ex. : contrôle industriel, véhicules autonomes). Le middleware doit alors gérer la synchronisation entre edge et cloud.

7.3.3 AI/ML dans l'Intégration

L'intelligence artificielle commence à être intégrée dans les plateformes de middleware pour :

— Routage intelligent basé sur l'apprentissage automatique (optimisation dynamique des chemins)

— Détection d'anomalies automatique (identification proactive des problèmes)

— Optimisation automatique des performances (ajustement dynamique des paramètres)

— Prédiction des pannes (maintenance prédictive)

Ces capacités permettront aux systèmes de s'auto-optimiser et de s'adapter automatiquement aux changements de charge ou de comportement.

7.3.4 Low-Code/No-Code Integration

Les plateformes visuelles (MuleSoft Composer, Zapier, Microsoft Power Automate) permettent de créer des intégrations sans écrire de code, rendant l'intégration accessible aux équipes métier. Cette démocratisation accélère la création d'intégrations tout en réduisant la dépendance aux équipes de développement.

7.4 Message Final

Le choix d'une topologie de middleware n'est pas une décision technique isolée, mais une décision stratégique qui impacte l'agilité, la scalabilité, les coûts, et la capacité d'innovation de l'entreprise. Comme nous l'avons illustré à travers les cas d'usage, il n'existe pas de solution universelle—la meilleure architecture est celle qui correspond aux besoins spécifiques de l'organisation, tout en tenant compte des contraintes techniques, organisationnelles, réglementaires, et budgétaires.

L'évolution continue des technologies et des pratiques garantit que ce domaine restera passionnant et en constante évolution. Les architectes d'intégration doivent rester à jour avec les dernières tendances (serverless, edge computing, AI/ML) tout en maîtrisant les fondamentaux (topologies classiques, patterns EIP) qui restent pertinents et qui constituent le socle sur lequel reposent les innovations futures.

Enfin, cet atelier a montré l'importance d'une approche pragmatique : plutôt que de suivre aveuglément les tendances, il faut analyser le contexte, comprendre les contraintes, et choisir la solution qui apporte le plus de valeur à l'entreprise engagée.

Références

[1] Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.

[2] Erl, T. (2005). *Service-Oriented Architecture : Concepts, Technology, and Design*. Prentice Hall.

[3] Word, J. (2009). *Systems Integration : A Practical Approach to Enterprise Integration*. Wiley.

[4] Linthicum, D. S. (2000). *Enterprise Application Integration*. Addison-Wesley Professional.

[5] Richardson, C. (2018). *Microservices Patterns : With Examples in Java*. Manning Publications.

[6] Newman, S. (2015). *Building Microservices : Designing Fine-Grained Systems*. O'Reilly Media.

[7] Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture : An Engineering Approach*. O'Reilly Media.

[8] OASIS. (2006). *SOA Reference Model*. OASIS Standard. https://docs.oasis-open.org/soa-rm

[9] WSO2. (2023). *WSO2 Enterprise Integrator Documentation (ESB, API Manager)*. https://wso2.com/integration

[10] IBM. (2023). *IBM MQ & IBM Integration Bus Documentation*. https://www.ibm.com/products/mq

[11] Hohpe, G. (n.d.). *Enterprise Integration Patterns - Site officiel*. https://www.enterpriseintegrationpatterns.com

[12] Apache Software Foundation. (2023). *Apache Kafka Documentation (event streaming & middleware distribué)*. https://kafka.apache.org/documentation

[13] RabbitMQ. (2023). *RabbitMQ Patterns & Topologies*. https://www.rabbitmq.com/getstarted.html

[14] IEEE. (2018). *A Survey on Middleware Architectures for Distributed Systems*. IEEE Transactions on Software Engineering.

[15] ACM. (2015). *Enterprise Application Integration : An Overview*. ACM Computing Surveys, 47(3).

[16] Elsevier. (2017). *Middleware Technologies for Distributed Systems : An Overview*. Journal of Systems and Software.

[17] IEEE. (2019). *SOA vs. Microservices : A Comparative Study*. IEEE Software.

[18] Gartner. (2021). *Event-Driven Architecture in Modern Enterprises*. Gartner Research Report.

[19] MIT. (2015). *Middleware for Distributed Enterprise Systems*. MIT Open Access Theses. https://dspace.mit.edu

[20] University of Colorado. (2017). *Service-Oriented Middleware for Enterprise Integration*. University of Colorado Boulder Theses and Dissertations.

[21] INRIA. (n.d.). *Publications sur l'intergiciel (middleware) & systèmes distribués*. https://hal.inria.fr