Guide de sélection des patrons pour la messagerie transactionnelle et les plateformes événementielles

Résumé

Le paysage technologique des entreprises modernes est défini par une dichotomie fondamentale dans l'intégration des systèmes : l'opposition entre la messagerie orientée messages (MOM), axée sur la fiabilité transactionnelle et les commandes, et l'architecture orientée événements (EDA), conçue pour le traitement en temps réel des flux de données. Ce livre blanc aborde cette problématique en analysant en profondeur les patrons d'architecture de deux technologies représentatives de ces paradigmes : IBM MQ pour la messagerie transactionnelle et Apache Kafka, avec Confluent Cloud, pour les plateformes événementielles. La portée de ce document va au-delà d'une simple documentation technique. Sa finalité est de fournir un cadre de décision structuré, un "aiguilleur architectural", destiné aux architectes de solutions et aux décideurs techniques. Cet aiguilleur a pour objectif de les guider dans la sélection du paradigme et des patrons les plus adaptés à leurs besoins métiers et non-fonctionnels spécifiques, assurant ainsi des choix d'architecture éclairés et justifiables.

Introduction

1.1. L'enjeu de l'interopérabilité dans l'entreprise moderne

Dans le contexte actuel d'accélération de la transformation numérique, l'interopérabilité des systèmes d'information n'est plus une option, mais une nécessité stratégique pour la compétitivité et la survie des entreprises.[1] Les organisations modernes s'appuient sur un écosystème d'applications hétérogènes, allant des systèmes centraux (mainframes) aux applications infonuagiques et aux dispositifs de l'Internet des objets (IoT). Cette hétérogénéité, si elle n'est pas maîtrisée, engendre des silos de données qui freinent l'agilité, complexifient les processus et entravent la prise de décision.[2]

L'interopérabilité, définie comme la capacité de systèmes distincts à échanger et à utiliser des informations de manière transparente, est le fondement technique qui permet de surmonter ces défis.[4] Elle procure des avantages tangibles : amélioration de l'efficacité opérationnelle par l'automatisation des flux de travail et la réduction des saisies manuelles, accès en temps réel à des données fiables pour une meilleure prise de décision, et facilitation de la collaboration entre les différentes unités d'affaires.[2] En brisant les barrières entre les systèmes, l'interopérabilité permet de construire une vue unifiée des opérations, ce qui est essentiel pour répondre rapidement aux évolutions du marché.[1]

Au-delà de ces bénéfices opérationnels, l'interopérabilité est le prérequis à une architecture d'entreprise "composable". Elle transforme un système d'information monolithique et rigide en un écosystème de capacités métier modulaires et réutilisables. En établissant des interfaces standardisées et faiblement couplées entre les services, elle permet de modifier, de remplacer ou d'ajouter des composants (comme un nouveau système de paiement ou un module d'analyse) avec un impact minimal sur le reste de l'écosystème. Cette capacité à assembler et réassembler dynamiquement des services informatiques pour répondre aux opportunités métier constitue l'essence même de l'agilité d'entreprise. L'interopérabilité n'est donc pas une simple préoccupation technique, mais l'infrastructure sur laquelle repose la capacité d'adaptation stratégique de l'entreprise moderne.

1.2. La dichotomie : Messagerie vs Événements

Au cœur des architectures d'interopérabilité se trouve une distinction conceptuelle fondamentale entre deux paradigmes

de communication : la messagerie orientée messages (MOM) et l'architecture orientée événements (EDA). Bien que les deux reposent sur la communication asynchrone, leur intention, leur sémantique et leurs implications architecturales diffèrent radicalement.

La **messagerie orientée messages (MOM)** est centrée sur le concept de **commande**. Un message est une unité de données envoyée avec une intention spécifique à un destinataire connu ou à une destination logique (une file d'attente).[6] L'expéditeur a une attente, implicite ou explicite, quant à l'action qui sera entreprise par le destinataire. Par exemple, un message contenant une instruction de virement bancaire est une commande adressée au système de paiement. Les systèmes MOM sont donc optimisés pour orchestrer des flux de travail, garantir l'exécution de transactions et assurer une livraison fiable et ordonnée de directives.

L'**architecture orientée événements (EDA)**, quant à elle, est centrée sur le concept de **fait**. Un événement est un enregistrement immuable d'un fait qui s'est produit dans le système, comme "une commande a été passée" ou "un capteur a atteint une température critique".[6] Le producteur de l'événement le diffuse sans avoir connaissance des consommateurs potentiels ni de la manière dont ils réagiront.[8] L'EDA est conçue pour la notification de changements d'état et le traitement de flux de données en temps réel, permettant à de multiples systèmes de réagir de manière autonome et découplée.

Cette dichotomie reflète une différence philosophique dans la conception des systèmes. Le paradigme MOM favorise un modèle de contrôle plus centralisé, où des processus peuvent orchestrer des séquences d'actions par l'envoi de commandes ciblées. La logique est souvent concentrée dans des orchestrateurs. À l'inverse, l'EDA promeut un modèle d'autonomie décentralisée, où la logique métier est distribué parmi des consommateurs indépendants qui réagissent aux événements. La coordination est implicite et émergeante, un processus connu sous le nom de chorégraphie. Le choix entre ces deux paradigmes est une décision architecturale structurante qui influence profondément la topologie du système, le niveau de couplage entre les services et la distribution de la logique métier.

1.3. Objectif et structure du document

Ce livre blanc a pour objectif de démystifier cette dichotomie et de fournir aux architectes les outils nécessaires pour faire des choix éclairés. La première partie du document est consacrée au paradigme MOM, illustré par IBM MQ, et explore ses patrons fondamentaux de communication et de haute disponibilité. La deuxième partie se penche sur le paradigme EDA avec Apache Kafka, en détaillant les patrons d'architecture événementielle qui permettent de construire des systèmes réactifs et scalables.

La proposition de valeur principale de ce document réside dans sa troisième partie : "L'aiguilleur architectural". Cette section présente une méthodologie de sélection pragmatique et structurée, conçue pour guider les architectes à travers une série d'étapes décisionnelles, depuis la qualification des besoins métier jusqu'à la sélection du patron technique approprié. En liant systématiquement les exigences fonctionnelles et non-fonctionnelles à des solutions d'architecture concrètes, ce guide vise à rendre le processus de décision plus rigoureux, transparent et justifiable.

Partie 1 : IBM MQ - Messagerie Orientée Messages (MOM)

2.1. Principes fondateurs

Depuis des décennies, IBM MQ constitue la pierre angulaire des systèmes transactionnels critiques dans des secteurs tels que la finance, la logistique et le commerce de détail. Son succès repose sur un ensemble de garanties qui assurent une communication robuste et découplée entre des applications distribuées.[9]

Le principe central d'IBM MQ est la **fiabilité**. La livraison des messages est assurée et dissociée de l'application.[9] Ceci est réalisé grâce à deux mécanismes clés : la persistance et la gestion transactionnelle. Les messages peuvent être marqués comme persistants, ce qui signifie qu'ils sont écrits sur disque par le gestionnaire de files d'attente avant que l'accusé de réception ne soit renvoyé à l'application émettrice. En cas de défaillance du système ou du gestionnaire de files d'attente, ces messages sont préservés et seront livrés après le redémarrage.

De plus, les opérations d'envoi (MQPUT) et de réception (MQGET) de messages peuvent être intégrées dans des unités de travail transactionnelles. L'application peut regrouper plusieurs opérations de messagerie et/ou des mises à jour de base de données dans une seule transaction atomique. Ce n'est qu'au moment du commit que les messages sont réellement envoyés ou définitivement retirés de la file. En cas d'erreur, un rollback annule toutes les opérations. Cette capacité transactionnelle est la base de la garantie de livraison "une et une seule fois" (*once-and-only-once*), qui est essentielle pour les processus métier où la perte ou la duplication d'un message est inacceptable.[9]

Le deuxième principe fondateur est le **couplage faible**. Les applications communiquent de manière asynchrone par l'intermédiaire d'une entité nommée, la file d'attente (*queue*). L'application productrice n'a pas besoin de savoir où se trouve l'application consommatrice, ni même si elle est en cours d'exécution au moment de l'envoi du message. Le gestionnaire de files d'attente agit comme un intermédiaire, stockant le message jusqu'à ce que le consommateur soit prêt à le traiter.[11] Ce découplage dans le temps et l'espace offre une grande résilience et une grande flexibilité à l'architecture, permettant aux composants d'évoluer indépendamment.

2.2. Patrons de communication fondamentaux

Les principes d'IBM MQ se matérialisent à travers des patrons de communication qui modélisent différentes formes d'interactions métier. Le choix d'un patron n'est pas seulement une décision technique, mais une formalisation du contrat de communication et du niveau de couplage souhaité entre les services.

2.2.1. Point à Point

Principe et Mécanisme
Le patron Point à Point est le modèle de communication le plus simple et le plus direct. Il modélise une délégation de tâche : un message est envoyé par un producteur et est destiné à être traité par un seul et unique consommateur.11 Même si plusieurs consommateurs écoutent la même destination, un seul d'entre eux recevra un message donné. Ce patron est idéal pour les scénarios de type commande, où une action spécifique doit être exécutée exactement une fois.
Implémentation avec IBM MQ
Ce patron est implémenté à l'aide de l'objet fondamental d'IBM MQ : la Queue (file d'attente). L'application productrice utilise l'appel d'API MQPUT pour déposer un message dans une file d'attente nommée. L'application consommatrice utilise l'appel MQGET pour récupérer le message de cette même file. L'opération MQGET est destructive par défaut (dans une unité de travail, le message est verrouillé puis supprimé au commit), garantissant qu'aucun autre consommateur ne pourra traiter le même message.12
Cas d'utilisation

- **Intégration de Systèmes Hétérogènes pour le Traitement des Ordres de Bourse :** Un ordre d'achat ou de vente soumis par un client est une commande critique. Il doit être traité de manière fiable et unique par le système de courtage. En plaçant l'ordre dans une file d'attente MQ, on garantit qu'il ne sera pas perdu en cas de pic de charge ou de défaillance momentanée du système de traitement. Le patron Point à Point assure qu'un seul processus de traitement prendra en charge l'ordre, évitant ainsi des exécutions multiples accidentelles.
- **Système de Paiements Bancaires Interbancaires :** Une instruction de virement de fonds entre deux banques est une

commande qui exige le plus haut niveau d'intégrité. Le message contenant les détails du virement est placé dans une file d'attente destinée à la banque réceptrice. Le patron Point à Point, combiné à la gestion transactionnelle de MQ, garantit que l'instruction de paiement sera traitée une seule et unique fois.

2.2.2. Publication/Abonnement

Principe et Mécanisme

Le patron Publication/Abonnement (Pub/Sub) modélise une diffusion d'information. Un producteur, appelé publisher, envoie un message sur un sujet logique, le Topic, sans se soucier de qui recevra l'information. Un ou plusieurs consommateurs, appelés subscribers, manifestent leur intérêt pour ce sujet en créant un abonnement (subscription). Tous les abonnés actifs à un topic reçoivent une copie de chaque message publié sur ce topic.11 Ce patron permet un découplage maximal, car le producteur et les consommateurs n'ont aucune connaissance les uns des autres.

Implémentation avec IBM MQ

IBM MQ implémente ce patron à l'aide des objets Topic et Subscription. Le moteur de publication/abonnement intégré au gestionnaire de files d'attente se charge de distribuer les messages publiés sur un topic à tous les abonnements correspondants.14 Une fonctionnalité cruciale est celle des abonnements durables (durable subscriptions). Un abonnement durable persiste même lorsque l'application abonnée est déconnectée. Les messages publiés pendant son absence sont conservés par le gestionnaire de files d'attente et lui sont livrés dès sa reconnexion, garantissant qu'aucune information n'est manquée.15

Cas d'utilisation

- **Système de Paiements Bancaires Interbancaires :** Bien que la transaction de virement elle-même utilise un patron Point à Point, la notification de son état est un cas d'usage parfait pour le Pub/Sub. Une fois le virement traité avec succès, le système de paiement central publie un message d'état (par exemple, "Virement #12345 complété") sur un topic payment.status.completed. Plusieurs systèmes internes peuvent s'abonner à ce topic : le système de reporting pour les rapports de fin de journée, le système de notification client pour envoyer un courriel de confirmation, et le système d'audit pour la traçabilité. Chacun de ces systèmes reçoit l'information et agit de manière autonome, sans que le système de paiement n'ait à gérer des intégrations point à point avec chacun d'eux.

2.2.3. Requête/Réponse

Principe et Mécanisme

Le patron Requête/Réponse modélise une conversation. Il permet à une application de simuler une interaction de type appel de procédure à distance (RPC) sur une infrastructure asynchrone. Une application "demandeuse" envoie un message de requête et attend une réponse spécifique à cette requête de la part d'une application "répondeuse".16 Ce patron introduit un couplage temporel plus fort que les autres, car le demandeur dépend d'une réponse pour poursuivre son traitement.

Implémentation avec IBM MQ

Ce patron est élégamment implémenté en utilisant les champs d'en-tête standard du message MQ. Le demandeur utilise deux files d'attente : une pour envoyer la requête et une pour recevoir la réponse.

1. Lors de l'envoi du message de requête, le demandeur peuple deux champs dans le descripteur de message (MQMD) :
   - ReplyToQ : Le nom de la file d'attente où il attend la réponse.
   - CorrelationID : Un identifiant unique qu'il génère pour cette requête spécifique.
2. Le répondeur reçoit la requête, la traite, puis prépare un message de réponse.
3. Il envoie le message de réponse à la file d'attente spécifiée dans le champ ReplyToQ de la requête.
4. Crucialement, il copie le CorrelationID du message de requête dans le champ CorrelationID du message de réponse.
5. Le demandeur effectue un MQGET sur sa file de réponse en spécifiant le CorrelationID de sa requête initiale. Le

gestionnaire de files d'attente lui retourne ainsi uniquement le message de réponse correspondant, lui permettant d'apparier sans ambiguïté les réponses à leurs requêtes respectives.[17]

Cas d'utilisation

- **Intégration de Systèmes Hétérogènes pour le Traitement des Ordres de Bourse :** Une application de trading en ligne doit fournir une confirmation quasi-instantanée à l'utilisateur lorsqu'un ordre est soumis. L'application frontale envoie l'ordre d'achat (la requête) à une file d'attente MQ. Le système de courtage, qui peut prendre quelques secondes pour valider et enregistrer l'ordre, traite la requête et renvoie un message de confirmation (la réponse) contenant un numéro de transaction. Grâce au patron Requête/Réponse, la confirmation est acheminée de manière fiable vers l'instance exacte de l'application frontale qui a initié la requête, lui permettant d'afficher le résultat à l'utilisateur.

2.3. Patrons de topologie et de haute disponibilité

Au-delà des modèles d'interaction, l'architecture d'une solution IBM MQ doit prendre en compte les exigences non-fonctionnelles telles que la scalabilité et la haute disponibilité. Le choix d'un patron de topologie est directement influencé par la stratégie d'infrastructure de l'entreprise, reflétant une évolution des modèles centrés sur le serveur vers des approches optimisées pour le cloud.

2.3.1. Cluster IBM MQ

Principe et Mécanisme
Un cluster IBM MQ est un réseau de deux ou plusieurs gestionnaires de files d'attente qui sont logiquement regroupés pour simplifier l'administration et permettre le partage de ressources.19 Au sein d'un cluster, les gestionnaires de files d'attente partagent automatiquement des informations sur les files d'attente et les canaux, ce qui réduit considérablement le nombre de définitions manuelles requises dans une topologie distribuée complexe.20
La principale valeur d'un cluster est la **répartition de charge** (*workload balancing*). Si une file d'attente est définie sur plusieurs gestionnaires de files d'attente au sein d'un cluster, une application peut envoyer un message à cette file "clusterisée" depuis n'importe quel gestionnaire de files d'attente du cluster. IBM MQ achemine alors automatiquement le message vers l'une des instances de la file, en se basant sur un algorithme de répartition de charge.[21] Il est essentiel de comprendre qu'un cluster MQ assure la scalabilité du service et la disponibilité du routage, mais pas la haute disponibilité des données elles-mêmes. Si un gestionnaire de files d'attente tombe en panne, les messages qu'il contient deviennent inaccessibles jusqu'à son redémarrage.

Cas d'utilisation

- **Intégration de Systèmes Hétérogènes pour le Traitement des Ordres de Bourse :** Une grande firme de courtage peut déployer plusieurs instances de son application de traitement des ordres sur différents serveurs pour gérer un volume élevé de transactions. En définissant la file d'attente des ordres entrants comme une file de cluster hébergée sur chaque serveur, les ordres soumis par les clients sont automatiquement distribués entre les instances d'application disponibles. Si un serveur devient surchargé ou est mis hors service pour maintenance, MQ redirige les nouveaux messages vers les autres membres actifs du cluster, assurant ainsi la continuité du service et une meilleure utilisation des ressources.

2.3.2. Gestionnaires de files d'attente multi-instances

Principe et Mécanisme
Ce patron fournit une solution de haute disponibilité (HA) de type actif/passif pour un gestionnaire de files d'attente

unique.23 L'architecture repose sur deux serveurs (ou machines virtuelles) :

1. Une instance **active** du gestionnaire de files d'attente s'exécute sur le premier serveur et traite toutes les opérations de messagerie.
2. Une instance passive (en attente) s'exécute sur le second serveur. Les deux instances partagent les mêmes données de files d'attente et les mêmes journaux de transactions, qui doivent être hébergés sur un stockage réseau partagé (par exemple, un système de fichiers NFS) accessible par les deux serveurs.24

L'instance active détient un verrou sur les fichiers de données. L'instance passive surveille continuellement ce verrou. Si le serveur actif ou l'instance active tombe en panne, le verrou est libéré. L'instance passive le détecte, acquiert le verrou, rejoue les journaux de transactions pour assurer la cohérence des données, et devient la nouvelle instance active. Les applications clientes peuvent alors se reconnecter et reprendre leurs opérations avec une interruption minimale. Ce patron assure la disponibilité des données du gestionnaire de files d'attente, mais introduit une dépendance critique sur la résilience du stockage réseau partagé.[25]

Cas d'utilisation

- **Intégration de Systèmes Hétérogènes pour le Traitement des Ordres de Bourse et Système de Paiements Bancaires Interbancaires :** Pour ces systèmes où toute interruption de service peut avoir des conséquences financières importantes, le patron multi-instances est une solution éprouvée. Il garantit qu'en cas de défaillance matérielle, de panne du système d'exploitation ou de maintenance planifiée du serveur actif, un basculement quasi-automatique vers le serveur de secours a lieu, préservant l'accès aux messages en transit et minimisant le temps d'indisponibilité (RTO/RPO).

2.3.3. Native HA pour les environnements conteneurisés

Principe et Mécanisme
Native HA est un patron de haute disponibilité moderne, spécifiquement conçu pour les déploiements d'IBM MQ sur des plateformes de conteneurs comme Kubernetes ou Red Hat OpenShift.26 Contrairement au modèle multi-instances, il ne repose pas sur un stockage réseau partagé, une approche souvent considérée comme un anti-patron dans les environnements cloud-natifs.
L'architecture Native HA utilise un groupe de trois pods Kubernetes, chacun exécutant une instance du même gestionnaire de files d'attente et disposant de son propre volume de stockage persistant (généralement du stockage par blocs).[27]

- Une instance est **active** : elle traite les connexions clientes et les messages.
- Les deux autres instances sont des **répliques** : elles reçoivent une copie synchrone de toutes les écritures du journal de transactions de l'instance active.

Si le pod actif échoue, un quorum entre les deux répliques restantes permet d'élire une nouvelle instance active. Comme cette nouvelle instance possède une copie à jour et cohérente des données grâce à la réplication synchrone, elle peut prendre le relais très rapidement sans perte de données.[25] Ce modèle de réplication au niveau de l'application est plus résilient et mieux aligné avec les principes de l'infrastructure immuable et distribuée des conteneurs.

Cas d'utilisation

- **Système de Paiements Bancaires Interbancaires :** Si une institution financière modernise son infrastructure en migrant ses applications critiques vers une plateforme conteneurisée, Native HA devient le patron de haute disponibilité de choix. En répartissant les trois pods sur différentes zones de disponibilité au sein d'une région cloud, l'architecture peut tolérer la panne d'un pod, d'un nœud de calcul, et même d'un centre de données entier, offrant

le niveau de résilience requis pour un service de paiement qui doit fonctionner 24/7.

Partie 2 : EDA - Architecture orientée événements avec Kafka

3.1. Principes fondateurs

Apache Kafka, et par extension les offres gérées comme Confluent Cloud, représente un changement de paradigme par rapport aux intergiciels de messagerie traditionnelle. Il ne doit pas être considéré comme une simple file d'attente de messages améliorée, mais comme un **journal de transactions distribué et immuable** (*distributed commit log*).[29] Cette distinction est fondamentale pour comprendre son architecture et ses cas d'usage.

Les concepts clés de Kafka sont les suivants :

- **Topic :** Un topic est un flux nommé d'événements. Il peut être vu comme une catégorie ou un fil d'actualité auquel les producteurs écrivent des événements et auquel les consommateurs s'abonnent.[30]
- **Partition :** Pour des raisons de scalabilité et de parallélisme, un topic est divisé en une ou plusieurs partitions. Chaque partition est un journal d'événements strictement ordonné et immuable. Les événements sont ajoutés de manière séquentielle à la fin du journal et ne peuvent pas être modifiés. La garantie d'ordre des messages n'est assurée qu'au sein d'une même partition.[30]
- **Offset :** Chaque événement au sein d'une partition se voit attribuer un identifiant séquentiel unique appelé offset. Cet offset agit comme un curseur, permettant aux consommateurs de savoir précisément où ils se trouvent dans le flux et de relire les événements depuis n'importe quel point dans le temps, tant que les données sont conservées.[30]
- **Rétention :** Contrairement à une file d'attente traditionnelle où les messages sont supprimés après consommation, les événements dans Kafka sont conservés pendant une période configurable (par exemple, 7 jours) ou jusqu'à une certaine taille. Cela permet à de multiples consommateurs de lire et relire le même flux d'événements indépendamment les uns des autres.
- **Traitement de flux (Stream Processing) :** Kafka n'est pas seulement un système de stockage et de transport. Son écosystème, notamment avec des bibliothèques comme Kafka Streams et des outils comme ksqlDB, est conçu pour permettre le traitement continu et en temps réel des événements à mesure qu'ils arrivent, permettant des transformations, des agrégations et des jointures de flux de données.[29]

3.2. Patrons d'architecture événementielle

Les principes de Kafka en font la plateforme idéale pour implémenter des patrons d'architecture avancés qui vont bien au-delà de la simple communication. Ces patrons, souvent utilisés conjointement, forment le cœur des systèmes réactifs et distribués modernes.

3.2.1. Approvisionnement en événements (Event Sourcing)

Principe et Mécanisme
L'approvisionnement en événements est un patron de persistance qui renverse l'approche traditionnelle de la gestion des données. Au lieu de stocker uniquement l'état actuel d'une entité dans une base de données (par exemple, le solde d'un compte), on stocke la séquence complète et chronologique de tous les événements qui ont affecté cette entité.31 L'état actuel est alors considéré comme une projection dérivée, obtenue en rejouant tous les événements depuis le début.32 Chaque changement d'état est capturé comme un événement immuable, créant une source de vérité auditable et complète.
Implémentation avec Kafka
Kafka est une plateforme naturelle pour l'implémentation de l'Event Sourcing. Un topic Kafka peut servir de magasin

d'événements (event store) pour un type d'entité donné (par exemple, un topic order-events pour les commandes).33 L'immuabilité des partitions garantit que l'historique ne peut pas être altéré. L'ordonnancement au sein d'une partition assure que les événements pour une entité spécifique (par exemple, une commande identifiée par son orderId utilisé comme clé de partitionnement) sont stockés et peuvent être relus dans l'ordre exact où ils se sont produits. La politique de rétention configurable de Kafka permet de conserver cet historique aussi longtemps que nécessaire.

Cas d'utilisation

- **Système de Suivi des Commandes en Temps Réel et Analyse des Tendances E-commerce :** Plutôt que d'avoir une table Commandes avec une colonne statut qui est constamment mise à jour, on enregistre une série d'événements : CommandeCréée, PaiementAccepté, ArticleExpédié, CommandeLivrée, ArticleRetourné. Cette approche offre un audit complet de la vie de la commande, permet de reconstruire son état à n'importe quel moment pour le support client, et fournit un flux de données riche pour analyser le parcours client, les délais de traitement ou les taux de retour.
- **Plateforme IoT pour la Surveillance de Capteurs Industriels :** Chaque mesure d'un capteur (température, pression) est enregistrée comme un événement immuable. Le magasin d'événements conserve l'historique complet des lectures. Cela permet non seulement de connaître l'état actuel de l'équipement, mais aussi d'analyser les tendances historiques, de détecter des anomalies en comparant les modèles actuels aux données passées, et de rejouer des séquences d'événements pour simuler et déboguer des pannes.

3.2.2. CQRS (Command Query Responsibility Segregation)

Principe et Mécanisme

Le patron CQRS préconise la séparation des modèles utilisés pour modifier l'état du système (les Commandes, le côté écriture) de ceux utilisés pour lire l'état du système (les Requêtes, le côté lecture).35 Souvent, dans les applications traditionnelles, un seul modèle de données (par exemple, une structure de tables relationnelles normalisées) est utilisé pour les deux opérations, ce qui conduit à des compromis. Le CQRS reconnaît que les exigences de l'écriture (cohérence, validation) et de la lecture (performance, agrégation de données pour l'affichage) sont fondamentalement différentes et bénéficient de modèles optimisés séparément.

Implémentation avec Kafka

Le CQRS est le complément naturel de l'Event Sourcing. Dans une telle architecture, le magasin d'événements dans Kafka constitue le modèle d'écriture, la source de vérité ultime. Cependant, interroger ce journal d'événements pour obtenir l'état actuel d'une entité peut être inefficace. Pour résoudre ce problème, on utilise des processeurs de flux comme Kafka Streams ou ksqlDB pour consommer le flux d'événements en temps réel et construire des modèles de lecture, aussi appelés vues matérialisées.36 Ces vues sont des représentations de l'état actuel, dénormalisées et optimisées pour des requêtes rapides. Elles peuvent être stockées dans divers systèmes (bases de données NoSQL, caches, ou même des KTables Kafka).

Cas d'utilisation

- **Système de Suivi des Commandes en Temps Réel et Analyse des Tendances E-commerce :** Le flux d'événements de commande dans Kafka est le modèle d'écriture. Un premier processus Kafka Streams consomme ce flux pour mettre à jour une base de données de recherche (comme Elasticsearch) utilisée par le service client pour retrouver rapidement des commandes. Un second processus, utilisant ksqlDB, agrège les données en temps réel pour alimenter un tableau de bord analytique affichant les ventes par région. Un troisième processus met à jour la vue "Mes Commandes" pour l'utilisateur dans une base de données relationnelle. Chaque modèle de lecture est ainsi parfaitement adapté à son cas d'usage, tout en étant dérivé de la même source de vérité.

3.2.3. Saga

Principe et Mécanisme

Le patron Saga est une approche pour gérer la cohérence des données dans une architecture de microservices sans recourir à des transactions distribuées (qui sont souvent complexes et peu performantes). Une saga est une séquence de transactions locales, où chaque transaction met à jour la base de données d'un seul service. Si une transaction locale échoue, la saga exécute une série de transactions de compensation pour annuler les opérations des transactions précédentes qui ont déjà réussi, garantissant ainsi une cohérence éventuelle.38

Implémentation avec Kafka

Kafka est souvent utilisé comme le bus de communication pour coordonner les étapes d'une saga. Il existe deux approches principales :

- **Chorégraphie :** Il n'y a pas de coordinateur central. Chaque service, après avoir complété sa transaction locale, publie un événement sur un topic Kafka. Les autres services s'abonnent aux événements qui les concernent et déclenchent leur propre transaction locale en réponse. La logique est décentralisée.[40]
- **Orchestration :** Un service orchestrateur central gère le flux de la saga. Il envoie des commandes (qui peuvent être des messages sur des topics Kafka) aux services participants et attend des événements de réponse. En fonction des réponses, il décide de la prochaine étape ou déclenche des transactions de compensation en cas d'erreur.[39]

Cas d'utilisation

- **Système de Suivi des Commandes en Temps Réel et Analyse des Tendances E-commerce :** Le processus de création d'une commande implique le service Commandes, le service Paiement et le service Inventaire. Une saga chorégraphiée se déroulerait comme suit :
  1. Le service Commandes reçoit une requête, effectue sa transaction locale (créer la commande avec le statut "en attente") et publie un événement CommandeCréée.
  2. Le service Paiement consomme l'événement CommandeCréée, traite le paiement et publie un événement PaiementEffectué.
  3. Le service Inventaire consomme l'événement PaiementEffectué, réserve le stock et publie StockRéservé. Si le paiement échoue, le service Paiement publie PaiementÉchoué. Le service Commandes consomme cet événement et exécute sa transaction de compensation (passer le statut de la commande à "annulée").

3.3. Patrons de diffusion et de résilience

3.3.1. Publication/Abonnement en éventail (Fan-out)

Principe et Mécanisme

Le patron Fan-out décrit le scénario où un seul message ou événement est diffusé et consommé indépendamment par plusieurs applications ou services distincts. Chaque consommateur reçoit sa propre copie du message et la traite selon sa propre logique, sans affecter les autres.

Implémentation avec Kafka

Ce patron est une caractéristique native et fondamentale du modèle de consommation de Kafka, géré par le concept de groupe de consommateurs (consumer group). Lorsqu'un producteur publie un événement sur un topic, Kafka le stocke. Pour que plusieurs applications reçoivent toutes une copie de ce flux d'événements, chacune doit s'abonner au topic en utilisant un group.id unique. Kafka garantit qu'une copie de chaque message du topic sera livrée à un membre de chaque groupe de consommateurs.42 Au sein d'un même groupe de consommateurs, les partitions du topic sont réparties entre les instances de l'application pour permettre la scalabilité du traitement, mais le groupe dans son ensemble ne voit chaque message qu'une seule fois.

Cas d'utilisation

- **Système de Suivi des Commandes en Temps Réel et Analyse des Tendances E-commerce :** L'événement CommandeCréée est publié une seule fois sur le topic orders. Le service de paiement (s'abonnant avec group.id=payments), le service de notification client (group.id=notifications) et le service d'analyse des fraudes (group.id=fraud-detection) consomment tous le même flux d'événements en parallèle, chacun pour son propre besoin.
- **Plateforme IoT pour la Surveillance de Capteurs Industriels :** Les données brutes des capteurs sont publiées sur un topic sensor-readings. Un service de surveillance en temps réel (group.id=real-time-monitoring), un service d'archivage à long terme qui écrit dans un data lake (group.id=archiving), et un service d'entraînement de modèles de maintenance prédictive (group.id=ml-training) consomment tous le même flux de données simultanément.

3.3.2. File d'attente de messages non traitables (Dead Letter Queue)

Principe et Mécanisme

La Dead Letter Queue (DLQ) est un patron de résilience essentiel pour la construction de pipelines de données robustes. Lorsqu'un message ne peut être traité avec succès par une application consommatrice, même après plusieurs tentatives (par exemple, en raison d'un format de données invalide, d'une violation d'une règle métier non récupérable, ou d'une erreur de désérialisation), il est déplacé vers un topic de destination spécial, la DLQ.43 Cela empêche le message "empoisonné" de bloquer le traitement des messages valides qui le suivent dans la partition, tout en préservant le message erroné pour une analyse ultérieure, un traitement manuel ou une réinjection automatisée après correction.45

Implémentation avec Kafka

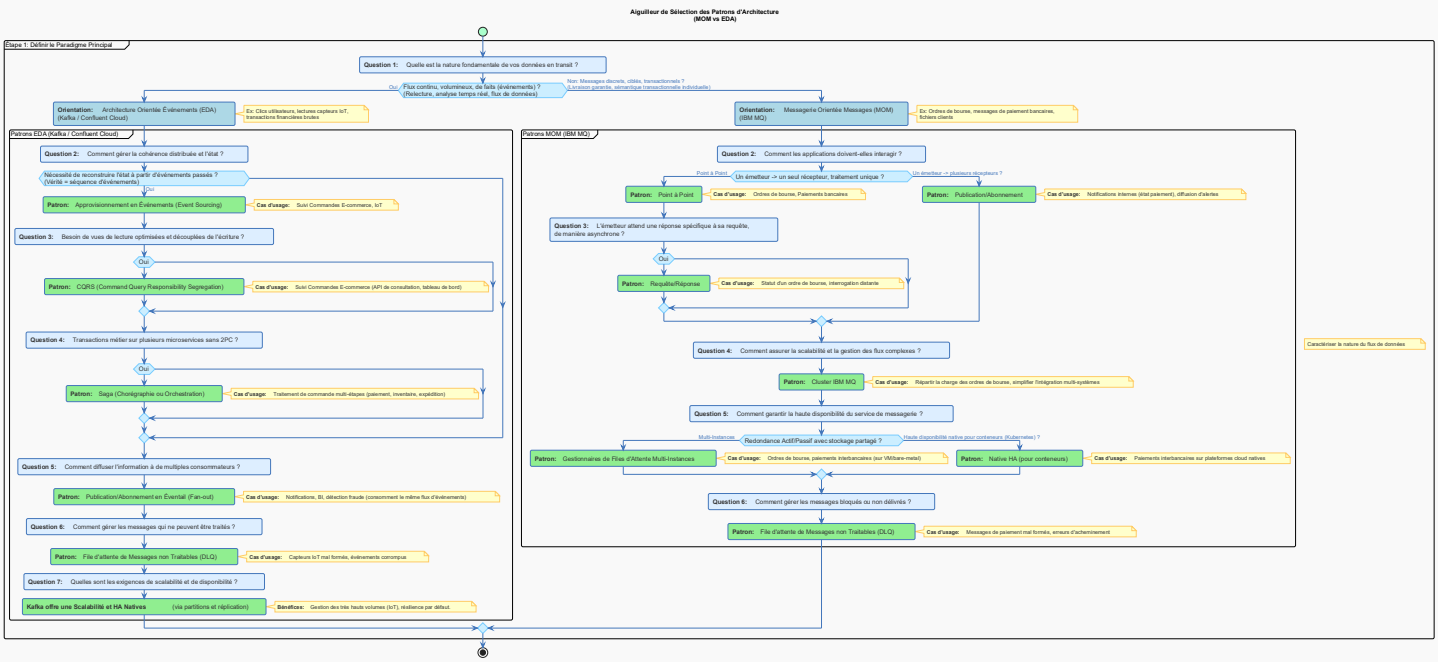
La logique de DLQ n'est pas une fonctionnalité native du broker Kafka, mais elle est implémentée au niveau de l'écosystème client.

- **Kafka Connect :** Le framework Kafka Connect dispose d'un support intégré pour les DLQ. Les connecteurs peuvent être configurés pour router automatiquement les enregistrements qui échouent lors de la conversion ou du traitement vers un topic DLQ spécifié.[46]
- **Bibliothèques clientes :** Des frameworks comme Spring for Kafka offrent des mécanismes de gestion d'erreurs qui, après un nombre configurable de tentatives de retraitement, peuvent publier le message échoué dans une DLQ.[48]
- **Implémentation manuelle :** Une application consommatrice peut implémenter cette logique en utilisant un bloc try-catch. En cas d'exception persistante, l'application utilise un KafkaProducer pour envoyer le message original, souvent enrichi de métadonnées sur l'erreur (comme la trace de la pile d'exécution et l'horodatage), vers le topic DLQ avant de valider l'offset du message original.[49]

Cas d'utilisation

- **Plateforme IoT pour la Surveillance de Capteurs Industriels :** Un capteur défectueux commence à envoyer des charges utiles de données corrompues (par exemple, du texte au lieu de JSON). Le service de traitement, qui s'attend à un format JSON valide, échoue systématiquement à désérialiser ces messages. Après trois tentatives infructueuses, le gestionnaire d'erreurs du consommateur publie le message binaire original dans le topic iot-raw-data-dlq, en ajoutant des en-têtes Kafka pour indiquer la source de l'erreur. Cela permet au traitement des données des autres capteurs de continuer sans interruption, tandis qu'une équipe d'ingénieurs peut examiner la DLQ pour diagnostiquer et corriger le problème sur le capteur défectueux.

Partie 3 : Guide de sélection d'un patron



4.1. Introduction à la méthodologie de sélection

Cette section constitue l'outil central de ce livre blanc. Elle propose une démarche structurée et pragmatique pour guider les architectes et les décideurs techniques dans le processus de sélection du patron d'architecture et de la technologie d'intégration les plus appropriés. L'objectif est de traduire les exigences métier et non-fonctionnelles en décisions d'architecture justifiées, en naviguant de manière systématique des questions de haut niveau vers des choix de patrons spécifiques. La méthodologie se décompose en quatre étapes séquentielles.

4.2. Étape 1 : Définir le paradigme principal (MOM vs EDA)

La première étape, et la plus fondamentale, consiste à déterminer la nature intrinsèque du flux de communication. La question clé à se poser est la suivante :

La communication représente-t-elle une *commande* ou un *fait*?

- **Si la communication est une COMMANDE :** L'intention est de déclencher une action spécifique, un traitement transactionnel, chez un destinataire ou un type de destinataire bien défini. L'expéditeur a une attente quant au résultat de cette action. Dans ce cas, le paradigme **MOM (Messagerie Orientée Messages)**, incarné par **IBM MQ**, est le point de départ naturel.
  - **Exemple illustratif :** Le cas du **Système de Paiements Bancaires Interbancaires** est emblématique. Une instruction de virement est une commande explicite : "Transférer X montant du compte A au compte B". Le destinataire est le système de la banque cible, et l'action attendue est précise et doit être exécutée de manière transactionnelle et unique.
- **Si la communication est un FAIT :** L'intention est de notifier qu'un événement, un changement d'état, s'est produit dans le système. Le producteur de l'événement n'a pas connaissance des consommateurs et ne leur dicte aucune action. Il ne fait que diffuser une information. Dans ce cas, le paradigme **EDA (Architecture Orientée Événements)**, incarné par **Apache Kafka**, est le plus approprié.
  - **Exemple illustratif :** Le cas du **Système de Suivi des Commandes en Temps Réel E-commerce** illustre

parfaitement ce paradigme. La création d'une commande est un fait. Le service qui publie l'événement CommandeCréée ne sait pas et ne se soucie pas de savoir si le service de paiement, le système de notification, ou un module d'analyse des fraudes vont réagir. Chacun de ces systèmes s'abonne au fait et agit de manière autonome.

4.3. Étape 2 : Qualifier le modèle d'interaction

Une fois le paradigme principal identifié, cette étape affine le choix en se concentrant sur la nature précise de l'interaction entre les composants. Le tableau suivant sert de grille de décision pour associer une intention métier à un patron de communication spécifique. **Tableau 4.3 : Grille de décision des modèles d'interaction**

Intention Métier	Question Clé	Patron Recommandé	Technologie	Cas d'Usage Illustratif
Déléguer une tâche unique	Une seule entité doit-elle traiter cette commande de manière fiable?	Point à Point	IBM MQ (Queue)	Traitement d'un ordre de bourse ; Instruction de virement bancaire.
Diffuser une information	Plusieurs systèmes autonomes doivent-ils être informés de ce fait ou de ce changement d'état?	Publication/Abonnement (Fan-out)	IBM MQ (Topic) / Kafka (Topic avec plusieurs groupes de consommateurs)	Notification d'état de paiement à des systèmes internes ; Diffusion des données de capteurs IoT à des applications d'analyse et d'archivage.
Engager une conversation	Le système initiateur a-t-il besoin d'une réponse directement corrélée à sa requête initiale pour poursuivre son traitement?	Requête/Réponse	IBM MQ (en-têtes ReplyToQ & CorrelationID)	Demande de confirmation d'un ordre de bourse avec un numéro de transaction en retour.

4.4. Étape 3 : Évaluer les exigences de cohérence et de gestion de l'état

Cette étape aborde les scénarios plus complexes impliquant la gestion de l'état et des transactions distribuées, qui sont souvent des facteurs déterminants dans le choix des patrons avancés, principalement dans le monde EDA.

- **Question 1 : La transaction métier s'étend-elle sur plusieurs microservices et nécessite-t-elle une logique d'annulation coordonnée en cas d'échec?**
  - Si la réponse est oui, le patron **Saga** est requis pour maintenir la cohérence des données à travers les services. Le choix se portera alors sur une implémentation par chorégraphie ou orchestration utilisant Kafka comme bus d'événements.
  - **Exemple :** Le processus de commande e-commerce, impliquant les services de commande, de paiement et d'inventaire, ne peut être réalisé en une seule transaction atomique et doit donc être modélisé comme une Saga.
- **Question 2 : Est-il nécessaire de conserver un historique complet, immuable et auditable de tous les changements apportés à une entité métier?**
  - Si la réponse est oui, notamment pour des raisons de conformité, d'audit, d'analyse historique ou de débogage avancé, le patron **Event Sourcing** est la solution la plus indiquée.
  - **Exemple :** Pour le suivi des commandes e-commerce, l'Event Sourcing permet de répondre à des questions comme "Quel était le statut de cette commande il y a trois jours à 15h00?" ou d'analyser le temps moyen entre l'acceptation du paiement et l'expédition, des capacités difficiles à obtenir avec un modèle de données qui ne stocke que l'état actuel.
- **Question 3 : Les modèles de données requis pour l'écriture (validation, cohérence) et la lecture (affichage, recherche) ont-ils des structures et des exigences de performance radicalement différentes?**
  - Si la réponse est oui, le patron **CQRS** doit être envisagé. Il est presque toujours utilisé en conjonction avec l'Event Sourcing pour créer des vues de lecture optimisées (vues matérialisées) à partir du flux d'événements.
  - **Exemple :** Le système e-commerce a besoin d'un modèle d'écriture transactionnel (le journal d'événements) mais aussi de multiples modèles de lecture : une structure dénormalisée pour la recherche rapide par le service client, des données agrégées pour les tableaux de bord, etc.

4.5. Étape 4 : Définir la topologie et les exigences non-fonctionnelles

La dernière étape se concentre sur les aspects non-fonctionnels critiques comme la haute disponibilité (HA), la scalabilité et la résilience, qui dictent le choix des patrons de topologie et de gestion des erreurs.

- **Pour les systèmes basés sur IBM MQ :**
  - **Haute Disponibilité :** La question clé est : "Quel est l'environnement de déploiement cible?"
    - Pour des déploiements sur des serveurs traditionnels ou des machines virtuelles, le patron **Gestionnaires de files d'attente multi-instances** est la solution standard pour la HA actif/passif.
    - Pour des déploiements sur des plateformes de conteneurs (Kubernetes, OpenShift), le patron **Native HA** est la solution moderne et idiomatique, offrant une résilience supérieure sans dépendre d'un stockage partagé.
    - **Exemple :** Les cas d'usage bancaires, avec leurs exigences de temps de fonctionnement de 99,999%, imposent l'un de ces deux patrons de HA robustes.
  - **Scalabilité :** La question clé est : "Est-il nécessaire de répartir la charge de traitement des messages sur plusieurs nœuds pour augmenter le débit?"
    - Si oui, le patron **Cluster IBM MQ** permet une répartition de charge transparente pour les applications.
- **Pour les systèmes basés sur Kafka :**
  - **Résilience :** La question clé est : "Comment le système doit-il réagir face à des messages qui échouent de manière

persistante au traitement?"

- Pour éviter de bloquer le traitement et de perdre des données, le patron **Dead Letter Queue (DLQ)** est indispensable.
- **Exemple :** La nature des données IoT, potentiellement variables et parfois malformées, rend l'implémentation d'une DLQ quasi-obligatoire pour garantir la robustesse du pipeline de données. Un capteur défectueux ne doit pas paralyser l'ingestion des données de milliers d'autres capteurs fonctionnels.

4.6. Matrice de synthèse des patrons et cas d'usage

Le tableau suivant récapitule les patrons discutés, leurs caractéristiques clés et les cas d'usage pertinents pour servir de référence rapide.**Tableau 4.6 : Matrice de synthèse des patrons, technologies et cas d'usage**

Patron	Paradigme	Principe Clé	Garantie de Livraison	Couplage	Cas d'Usage Pertinent
Point à Point	MOM	Un message, un consommateur.	At-least-once, Exactly-once	Faible (destination)	Ordre de Bourse, Virement Bancaire
Publication/Abonnement	MOM / EDA	Un message, plusieurs consommateurs.	At-least-once	Très faible (sujet)	Notification d'état de paiement
Requête/Réponse	MOM	Conversation synchrone sur infrastructure asynchrone.	At-least-once, Exactly-once	Fort (conversationnel)	Confirmation d'ordre de bourse
Cluster IBM MQ	MOM	Répartition de charge et administration simplifiée.	N/A (topologie)	N/A	Scalabilité du traitement des ordres
Multi-instances MQ	MOM	HA actif/passif avec stockage partagé.	N/A (HA)	N/A	Systèmes bancaires critiques sur VM
Native HA MQ	MOM	HA	N/A (HA)	N/A	Systèmes

		actif/répliques sans stockage partagé.			bancaires sur Kubernetes
Event Sourcing	EDA	Persistance de la séquence d'événements comme source de vérité.	At-least-once	Très faible	Suivi de commande E-commerce, IoT
CQRS	EDA	Séparation des modèles de lecture et d'écriture.	N/A (persistance)	N/A	Tableaux de bord E-commerce
Saga	EDA	Gestion de transactions distribuées via des compensations.	At-least-once	Faible (événementiel)	Processus de commande E-commerce
Fan-out (Kafka)	EDA	Diffusion d'un flux à plusieurs groupes de consommateurs.	At-least-once	Très faible (sujet)	Plateforme IoT, Analyse E-commerce
Dead Letter Queue	EDA	Isolation des messages non traitables pour la résilience.	N/A (résilience)	N/A	Traitement des données IoT

Conclusion

5.1. Synthèse : Choisir le bon outil pour le bon travail

Ce livre blanc a exploré les deux paradigmes dominants de l'interopérabilité des systèmes d'entreprise : la messagerie orientée messages (MOM) et l'architecture orientée événements (EDA). À travers l'analyse d'IBM MQ et d'Apache Kafka, il est devenu clair qu'il n'existe pas de solution universelle. La philosophie directrice qui doit guider toute décision architecturale est celle de "choisir le bon outil pour le bon travail".

IBM MQ excelle dans les scénarios qui exigent une fiabilité transactionnelle absolue, des garanties de livraison "une et une seule fois" et une communication ciblée de type commande. Il est le choix privilégié pour le cœur transactionnel des entreprises, là où chaque message représente une opération métier critique qui ne peut tolérer ni perte ni duplication.[10]

Apache Kafka, de son côté, est conçu pour la gestion et le traitement à grande échelle de flux continus d'événements. Sa force réside dans sa capacité à servir de journal immuable pour des architectures comme l'Event Sourcing, à découpler radicalement les producteurs des consommateurs, et à permettre l'analyse en temps réel. Il est l'outil de choix pour les pipelines de données, les systèmes réactifs et les cas d'usage où le volume, la vitesse et la capacité de rejouer les données sont primordiaux.[10]

Le rôle de l'architecte est d'analyser rigoureusement la nature du problème métier à résoudre—s'agit-il d'une commande ou d'un fait?—et de sélectionner le paradigme et les patrons qui y répondent le mieux.

5.2. Vers des architectures hybrides

Les mondes de la messagerie transactionnelle et des plateformes événementielles ne sont pas mutuellement exclusifs ; au contraire, ils sont de plus en plus complémentaires. Les entreprises les plus performantes reconnaissent que la force réside souvent dans la combinaison des deux paradigmes au sein d'une architecture hybride cohérente.[51]

Un patron hybride courant consiste à utiliser Kafka pour ce qu'il fait de mieux : l'ingestion et le traitement en temps réel de volumes massifs de données à faible valeur unitaire. Par exemple, Kafka peut agréger des flux d'événements provenant de clics sur un site web, de logs d'application ou de capteurs IoT. Des processeurs de flux peuvent analyser ces données pour en extraire des informations à haute valeur ajoutée ou des événements métier significatifs (par exemple, "Détection d'une transaction potentiellement frauduleuse" ou "Client VIP ajoutant un article coûteux à son panier").

Ces événements métier critiques, qui nécessitent une action garantie et transactionnelle, sont ensuite transmis à IBM MQ. IBM MQ prend alors le relais pour orchestrer de manière fiable les transactions dans les systèmes centraux, comme le blocage d'une carte de crédit ou le déclenchement d'une offre promotionnelle personnalisée.[53]

Le pont technologique standard pour réaliser cette intégration est **Kafka Connect**. IBM fournit des connecteurs source et sink pour IBM MQ, qui permettent de transférer de manière fiable des messages d'une file d'attente MQ vers un topic Kafka (source) ou des événements d'un topic Kafka vers une file d'attente MQ (sink).[54] Cette approche permet de tirer parti de la scalabilité de Kafka en amont et de la robustesse transactionnelle de MQ en aval, créant ainsi une architecture globale plus puissante et résiliente.

5.3. Perspectives

L'évolution des architectures d'intégration ne s'arrête pas. Plusieurs tendances émergentes continuent de redéfinir la manière dont les systèmes communiquent, en déplaçant la complexité de l'intergiciel vers des couches d'infrastructure plus abstraites et intelligentes. Une tendance majeure est l'adoption du **maillage de services (Service Mesh)**, particulièrement dans les écosystèmes de microservices conteneurisés. Un maillage de services comme Istio ou Linkerd externalise la logique de communication inter-services (routage avancé, sécurité avec mTLS, résilience via des disjoncteurs, observabilité) dans un proxy "sidecar" qui s'exécute aux côtés de chaque service.[57] En gérant la communication au niveau de l'infrastructure plutôt qu'au niveau de l'application ou du broker, le maillage de services pourrait à terme réduire le périmètre fonctionnel des intergiciels traditionnels, qui se concentreraient alors davantage sur le stockage durable et la sémantique de la communication (commande vs événement) que sur les aspects réseau.[59]

Une autre tendance structurante est l'essor des **architectures événementielles sans serveur (Serverless EDA)**. L'intégration native de plateformes comme Kafka avec des services de calcul sans serveur, tels qu'AWS Lambda ou Azure Functions, permet de construire des systèmes hautement réactifs et élastiques avec une charge opérationnelle minimale.[60] Dans ce modèle, les développeurs se concentrent uniquement sur l'écriture de fonctions qui réagissent à des événements, tandis que la plateforme infonuagique gère automatiquement la mise à l'échelle, le provisionnement et la disponibilité de l'infrastructure d'exécution en fonction de la charge.[63]

La convergence de ces tendances suggère un avenir où l'intégration système devient de plus en plus déclarative et automatisée. Le rôle de l'architecte évolue : il passe de la configuration manuelle de "tuyaux" et de brokers à la définition de "politiques" de communication, de sécurité et de résilience au sein d'un écosystème de plus en plus intelligent et autonome. La maîtrise des principes fondamentaux de la messagerie et des événements, telle que présentée dans ce document, restera cependant la compétence essentielle pour concevoir et gouverner ces architectures du futur.

Ouvrages cités

1. Pourquoi l'interopérabilité est la nouvelle norme de l'entreprise du futur. Gabriel S. Serapiao Leal, Meritis - InformatiqueNews.fr, dernier accès : octobre 15, 2025, https://www.informatiquenews.fr/pourquoi-linteroperabilite-est-la-nouvelle-norme-de-lentreprise-du-futur-gabriel-s-serapiao-leal-meritis-77065
2. Qu'est-ce que l'interopérabilité ? | IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/fr-fr/think/topics/interoperability
3. Qu'est-ce que l'interopérabilité - AWS, dernier accès : octobre 15, 2025, https://aws.amazon.com/fr/what-is/interoperability/
4. Introduction à l'interopérabilité des systèmes - Meritis, dernier accès : octobre 15, 2025, https://meritis.fr/introduction-a-linteroperabilite-des-systemes/
5. Interopérabilité : Le cœur battant de la collaboration technologique - Enreach, dernier accès : octobre 15, 2025, https://www.enreach.com/fr/actualite-ressources/blog-thematique/interoperabilite-le-coeur-battant-de-la-collaboration-technologique
6. Scaling Microservices: Message-Driven vs Event-Driven Architecture | by Leela Kumili, dernier accès : octobre 15, 2025, https://medium.com/@leela.kumili/scaling-microservices-message-driven-vs-event-driven-architecture-16fe79402e40
7. Message Driven vs Event Driven :: Akka Guide, dernier accès : octobre 15, 2025, https://doc.akka.io/libraries/guide/concepts/message-driven-event-driven.html
8. The Complete Guide to Event-Driven Architecture | by Seetharamugn | Medium, dernier accès : octobre 15, 2025, https://medium.com/@seetharamugn/the-complete-guide-to-event-driven-architecture-b25226594227
9. Présentation d'IBM MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/SSFKSJ_9.2.0/com.ibm.mq.pro.doc/q001020_.htm
10. Kafka vs. IBM MQ: Key Differences and Comparative Analysis - GitHub, dernier accès : octobre 15, 2025, https://github.com/AutoMQ/automq/wiki/Kafka-vs.-IBM-MQ:-Key-Differences-and-Comparative-Analysis
11. About IBM MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-mq/9.2.x?topic=mq-about
12. IBM MQ queue points and mediation points, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/was/9.0.5?topic=server-mq-queue-points-mediation-points
13. MQGET (Get message) on IBM i, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/SSFKSJ_9.2.0/com.ibm.mq.ref.dev.doc/q106150_.htm

14. Subscriptions - IBM® MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/SSFKSJ_9.2.0/com.ibm.mq.explorer.doc/e_subscriptions.html
15. Subscriptions - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-mq/9.2?topic=concepts-subscriptions
16. Receive an MQ request message, pass the message to MQ, and then pass the response from MQ to the requesting client - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/app-connect/11.0.0?topic=wm-receive-mq-request-message-pass-message-mq-then-pass-response-from-mq-requesting-client
17. Retrieving responses by message ID or by correlation ID - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/datapower-gateway/10.6.0?topic=mha-retrieving-responses-by-message-id-by-correlation-id
18. MQ and MQ JMS features - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/bpm/8.5.7?topic=jms-mq-mq-features
19. Creating and configuring a queue manager cluster - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/SSFKSJ_9.2.0/com.ibm.mq.explorer.doc/e_cluster_configuring.html
20. Example clusters - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/SSFKSJ_9.2.0/com.ibm.mq.pla.doc/q002755_.htm
21. IBM MQ Clustering Made Simple - Medium, dernier accès : octobre 15, 2025, https://medium.com/@arvindtechweb/ibm-mq-clustering-made-simple-167e7b6b6bc0
22. IBM MQ cluster commands and attributes, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-mq/9.4.x?topic=reference-mq-cluster-commands-attributes
23. Gestionnaires de files d'attente multi-instance sur IBM i, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.2?topic=i-multi-instance-queue-managers
24. Récupération de données et haute disponibilité - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.2.x?topic=managers-data-recovery-high-availability
25. Haute disponibilité pour IBM MQ dans des conteneurs, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.3?topic=containers-high-availability-mq-in
26. Configurations à haute disponibilité - IBM MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.3.x?topic=restart-high-availability-configurations
27. Native HA - IBM® MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.2.x?topic=operator-native-ha
28. Sélection de l'architecture à haute disponibilité cible pour IBM MQ s'exécutant dans des conteneurs, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/fr/ibm-mq/9.4.0?topic=mmo-selecting-target-ha-architecture-mq-running-in-containers
29. Apache Kafka : Le guide complet des concepts et des fonctionnalités - Ambient IT, dernier accès : octobre 15, 2025, https://www.ambient-it.net/apache-kafka-guide-complet/
30. Learning Apache Kafka concepts - IBM Cloud Docs, dernier accès : octobre 15, 2025, https://cloud.ibm.com/docs/EventStreams?topic=EventStreams-apache_kafka
31. Event sourcing with Kafka: A practical example - Tinybird, dernier accès : octobre 15, 2025, https://www.tinybird.co/blog-posts/event-sourcing-with-kafka
32. Event Sourcing pattern - Azure Architecture Center | Microsoft Learn, dernier accès : octobre 15, 2025, https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
33. Event Sourcing: An Introduction - Confluent, dernier accès : octobre 15, 2025, https://www.confluent.io/learn/event-sourcing/
34. Event Sourcing with Kafka: Architecture Patterns You Should Know - AI Academy, dernier accès : octobre 15, 2025, https://ai-academy.training/2025/05/24/event-sourcing-with-kafka-architecture-patterns-you-should-know/

35. CQRS Pattern - Azure Architecture Center | Microsoft Learn, dernier accès : octobre 15, 2025, https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs
36. What is CQRS in Event Sourcing Patterns? - Confluent Developer, dernier accès : octobre 15, 2025, https://developer.confluent.io/courses/event-sourcing/cqrs/
37. CQRS Pattern with Kafka Streams — Part 1 | by Saeed Aghaee - Medium, dernier accès : octobre 15, 2025, https://mrwersa.medium.com/cqrs-pattern-with-kafka-streams-part-1-112f381e9b98
38. Implementing the Saga Pattern with Choreography and Orchestration | by Dinesh Arney, dernier accès : octobre 15, 2025, https://medium.com/@dinesharney/implementing-the-saga-pattern-using-choreography-and-orchestration-53e66cbd520e
39. Saga Design Pattern - Azure Architecture Center | Microsoft Learn, dernier accès : octobre 15, 2025, https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
40. Saga pattern: Choreography and Orchestration | by Blogs4devs - Medium, dernier accès : octobre 15, 2025, https://medium.com/@blogs4devs/saga-pattern-choreography-and-orchestration-1758b61e1cfa
41. Saga Pattern in Microservices: Orchestration vs Choreography - DEV Community, dernier accès : octobre 15, 2025, https://dev.to/rock_win_c053fa5fb2399067/saga-pattern-in-microservices-orchestration-vs-choreography-mml
42. How to make fanout in Apache Kafka? - Codemia, dernier accès : octobre 15, 2025, https://codemia.io/knowledge-hub/path/how_to_make_fanout_in_apache_kafka
43. Apache Kafka Dead Letter Queue: A Comprehensive Guide - Confluent, dernier accès : octobre 15, 2025, https://www.confluent.io/learn/kafka-dead-letter-queue/
44. Building Reliable Reprocessing and Dead Letter Queues with Kafka | Uber Blog, dernier accès : octobre 15, 2025, https://www.uber.com/blog/reliable-reprocessing/
45. Dead Letter Queue - Karafka framework documentation, dernier accès : octobre 15, 2025, https://karafka.io/docs/Dead-Letter-Queue/
46. Error Handling via Dead Letter Queue in Apache Kafka - Kai Waehner, dernier accès : octobre 15, 2025, https://www.kai-waehner.de/blog/2022/05/30/error-handling-via-dead-letter-queue-in-apache-kafka/
47. Kafka Connect Deep Dive – Error Handling and Dead Letter Queues | Confluent, dernier accès : octobre 15, 2025, https://www.confluent.io/blog/kafka-connect-deep-dive-error-handling-dead-letter-queues/
48. Dead-Letter Topic Processing :: Spring Cloud Stream, dernier accès : octobre 15, 2025, https://docs.spring.io/spring-cloud-stream/reference/kafka/kafka-binder/dlq.html
49. What is the best practice to retry messages from Dead letter Queue for Kafka, dernier accès : octobre 15, 2025, https://stackoverflow.com/questions/65747292/what-is-the-best-practice-to-retry-messages-from-dead-letter-queue-for-kafka
50. IBM MQ vs Apache Kafka: Choosing the Best for Message Queuing and Event Streaming, dernier accès : octobre 15, 2025, https://risingwave.com/blog/ibm-mq-vs-apache-kafka-choosing-the-best-for-message-queuing-and-event-streaming/
51. MQ vs Kafka Explained: Differences, Similarities & Combined Power - Avada Software, dernier accès : octobre 15, 2025, https://avadasoftware.com/mq-vs-kafka/
52. The winning combination for real-time insights: Messaging and event-driven architecture, dernier accès : octobre 15, 2025, https://www.ibm.com/new/product-blog/the-winning-combination-for-real-time-insights-messaging-and-event-driven-architecture
53. Bridging the Gap: Event-Driven Architectures and Mainframes | by Madhu | Medium, dernier accès : octobre 15, 2025, https://medium.com/@madhuba07/bridging-the-gap-event-driven-architectures-and-mainframes-917c8d2e82d0
54. Kafka Connect scenarios - IBM MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-mq/9.3.x?topic=scenarios-kafka-connect
55. Kafka Connect scenarios - IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-

mq/9.4.x?topic=scenarios-kafka-connect
56. Kafka Connect common topologies - IBM® MQ, dernier accès : octobre 15, 2025, https://www.ibm.com/docs/en/ibm-mq/9.3.x?topic=scenarios-kafka-connect-common-topologies
57. Un Service Mesh, c'est quoi ? - Red Hat, dernier accès : octobre 15, 2025, https://www.redhat.com/fr/topics/microservices/what-is-a-service-mesh
58. What is Service Mesh? - AWS, dernier accès : octobre 15, 2025, https://aws.amazon.com/what-is/service-mesh/
59. What is a Service Mesh? | IBM, dernier accès : octobre 15, 2025, https://www.ibm.com/think/topics/service-mesh
60. Building serverless event streaming applications with Amazon MSK and AWS Lambda, dernier accès : octobre 15, 2025, https://aws.amazon.com/blogs/big-data/building-serverless-event-streaming-applications-with-amazon-msk-and-aws-lambda/
61. How does serverless architecture handle event-driven workflows? - Milvus, dernier accès : octobre 15, 2025, https://milvus.io/ai-quick-reference/how-does-serverless-architecture-handle-eventdriven-workflows
62. Event-Driven Architecture with Serverless Functions — Part 1 | by Superstream | Medium, dernier accès : octobre 15, 2025, https://superstreamai1.medium.com/event-driven-architecture-with-serverless-functions-part-1-d2b54babc193
63. Serverless EDA with AWS Lambda & Kafka - Aiven, dernier accès : octobre 15, 2025, https://aiven.io/developer/serverless-eda-aws-lambda-apache-kafka

Annexe A – PlantUML

```
@startuml
!theme mars
' Configuration globale du style
skinparam activity {
  BackgroundColor #DDEEFF
  BorderColor #4477BB
  FontColor #333333
  FontSize 14
  FontName "Arial"

  StartColor #AAFFCC
  EndColor #FF9999

  DiamondBackgroundColor #CCEEFF
  DiamondBorderColor #77CCFF

  ArrowColor #4477BB
  ArrowFontColor #4477BB
  ArrowFontSize 12
}

skinparam note {
  BackgroundColor #FFFFCC
  BorderColor #FFCC66
  FontColor #333333
  FontSize 12
  FontName "Arial"
```

```
}

title "Aiguilleur de Sélection des Patrons d'Architecture\n(MOM vs EDA)"

start

partition "Étape 1: Définir le Paradigme Principal" {
  note right: Caractériser la nature du flux de données

  :**Question 1:** Quelle est la nature fondamentale de vos données en transit ?;

  if (Flux continu, volumineux, de faits (événements) ? \n(Relecture, analyse temps réel, flux de données)) then
(Oui)
    #lightblue:**Orientation:** Architecture Orientée Événements (EDA)\n(Kafka / Confluent Cloud);
    note right: Ex: Clics utilisateurs, lectures capteurs IoT,\ntransactions financières brutes

    partition "Patrons EDA (Kafka / Confluent Cloud)" {
      :**Question 2:** Comment gérer la cohérence distribuée et l'état ?;
      if (Nécessité de reconstruire l'état à partir d'événements passés ? \n(Vérité = séquence d'événements))
then (Oui)
        #lightgreen:**Patron:** Approvisionnement en Événements (Event Sourcing);
        note right: **Cas d'usage:** Suivi Commandes E-commerce, IoT

        :**Question 3:** Besoin de vues de lecture optimisées et découplées de l'écriture ?;
        if (Oui) then ( )
          #lightgreen:**Patron:** CQRS (Command Query Responsibility Segregation);
          note right: **Cas d'usage:** Suivi Commandes E-commerce (API de consultation, tableau de bord)
        endif

        :**Question 4:** Transactions métier sur plusieurs microservices sans 2PC ?;
        if (Oui) then ( )
          #lightgreen:**Patron:** Saga (Chorégraphie ou Orchestration);
          note right: **Cas d'usage:** Traitement de commande multi-étapes (paiement, inventaire, expédition)
        endif
      endif

      :**Question 5:** Comment diffuser l'information à de multiples consommateurs ?;
      #lightgreen:**Patron:** Publication/Abonnement en Éventail (Fan-out);
      note right: **Cas d'usage:** Notifications, BI, détection fraude (consomment le même flux d'événements)

      :**Question 6:** Comment gérer les messages qui ne peuvent être traités ?;
      #lightgreen:**Patron:** File d'attente de Messages non Traitables (DLQ);
      note right: **Cas d'usage:** Capteurs IoT mal formés, événements corrompus

      :**Question 7:** Quelles sont les exigences de scalabilité et de disponibilité ?;
      #lightgreen:**Kafka offre une Scalabilité et HA Natives** (via partitions et réplication);
      note right: **Bénéfices:** Gestion des très hauts volumes (IoT), résilience par défaut.
    }

  else (Non: Messages discrets, ciblés, transactionnels ? \n(Livraison garantie, sémantique transactionnelle
individuelle))
    #lightblue:**Orientation:** Messagerie Orientée Messages (MOM)\n(IBM MQ);
    note right: Ex: Ordres de bourse, messages de paiement bancaires,\nfichiers clients
```

```
    partition "Patrons MOM (IBM MQ)" {
      :**Question 2:** Comment les applications doivent-elles interagir ?;
      if (Un émetteur -> un seul récepteur, traitement unique ?) then (Point à Point)
        #lightgreen:**Patron:** Point à Point;
        note right: **Cas d'usage:** Ordres de bourse, Paiements bancaires

        :**Question 3:** L'émetteur attend une réponse spécifique à sa requête, \nde manière asynchrone ?;
        if (Oui) then ( )
          #lightgreen:**Patron:** Requête/Réponse;
          note right: **Cas d'usage:** Statut d'un ordre de bourse, interrogation distante
        endif
      else (Un émetteur -> plusieurs récepteurs ?)
        #lightgreen:**Patron:** Publication/Abonnement;
        note right: **Cas d'usage:** Notifications internes (état paiement), diffusion d'alertes
      endif

      :**Question 4:** Comment assurer la scalabilité et la gestion des flux complexes ?;
      #lightgreen:**Patron:** Cluster IBM MQ;
      note right: **Cas d'usage:** Répartir la charge des ordres de bourse, simplifier l'intégration multi-systèmes

      :**Question 5:** Comment garantir la haute disponibilité du service de messagerie ?;
      if (Redondance Actif/Passif avec stockage partagé ?) then (Multi-Instances)
        #lightgreen:**Patron:** Gestionnaires de Files d'Attente Multi-Instances;
        note right: **Cas d'usage:** Ordres de bourse, paiements interbancaires (sur VM/bare-metal)
      else (Haute disponibilité native pour conteneurs (Kubernetes) ?)
        #lightgreen:**Patron:** Native HA (pour conteneurs);
        note right: **Cas d'usage:** Paiements interbancaires sur plateformes cloud natives
      endif

      :**Question 6:** Comment gérer les messages bloqués ou non délivrés ?;
      #lightgreen:**Patron:** File d'attente de Messages non Traitables (DLQ);
      note right: **Cas d'usage:** Messages de paiement mal formés, erreurs d'acheminement
    }
  endif
}

stop
@enduml
```