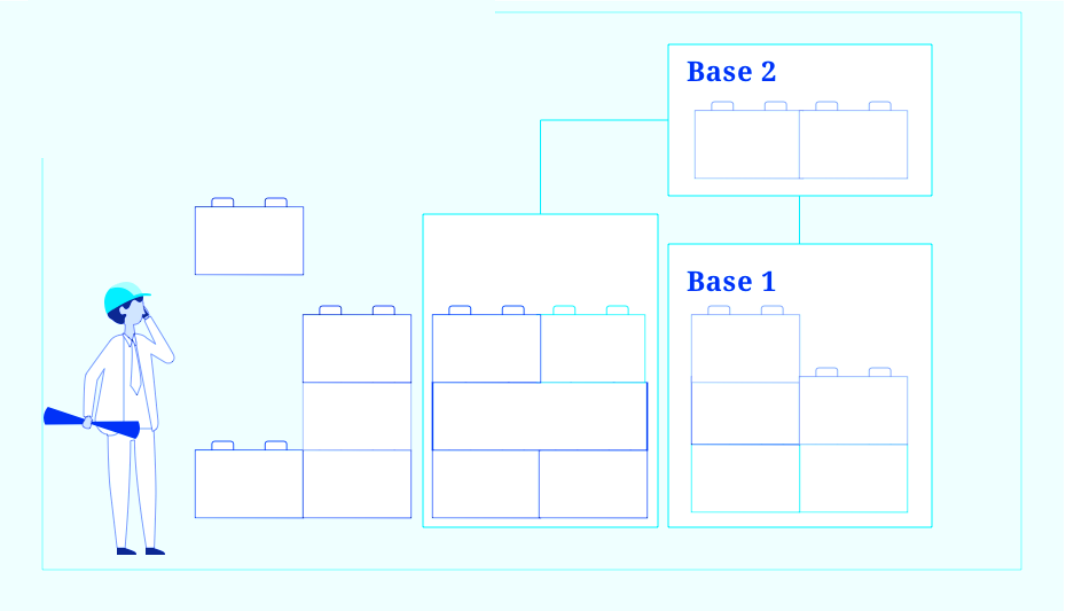
IUT Lyon 1
l'excellence technologique

Chapitre 9 – Autres patterns d'Architecture microservices et Patterns DevOps



Véronique DESLANDRES - LP DevOps, IUT Univ. Lyon1

Module Conception d'Architectures Logicielles

Sommaire de ce cours

Patterns de gestion des données

- Problématique importante pour les microservices
  - À traitement transactionnel
  - (différente pour les données du bigData : A*zure Synapse, Teradata…)*

- Avoir des services faiblement couplés, pour :
  - Facilité de déploiement, un passage à l'échelle indépendant
- Mais aussi les bases de données sont parfois répliquées et partitionnées pour optimiser certaines requêtes
- **Chaque microservice est responsable de ses données**, de leur cohérence et de leur intégrité.
  - Accès par une API à accès limité
  - Une même base de données ne doit jamais être gérée par 2 microservices différents.
- Nous allons voir les patterns suivants :
  - Pattern **CQRS**
  - Pattern **EventSourcing** : stocker les séquences d'événements
  - Pattern **SAGA** : implémenter des transactions qui impliquent plusieurs MS

Le modèle d’une BD par microservice

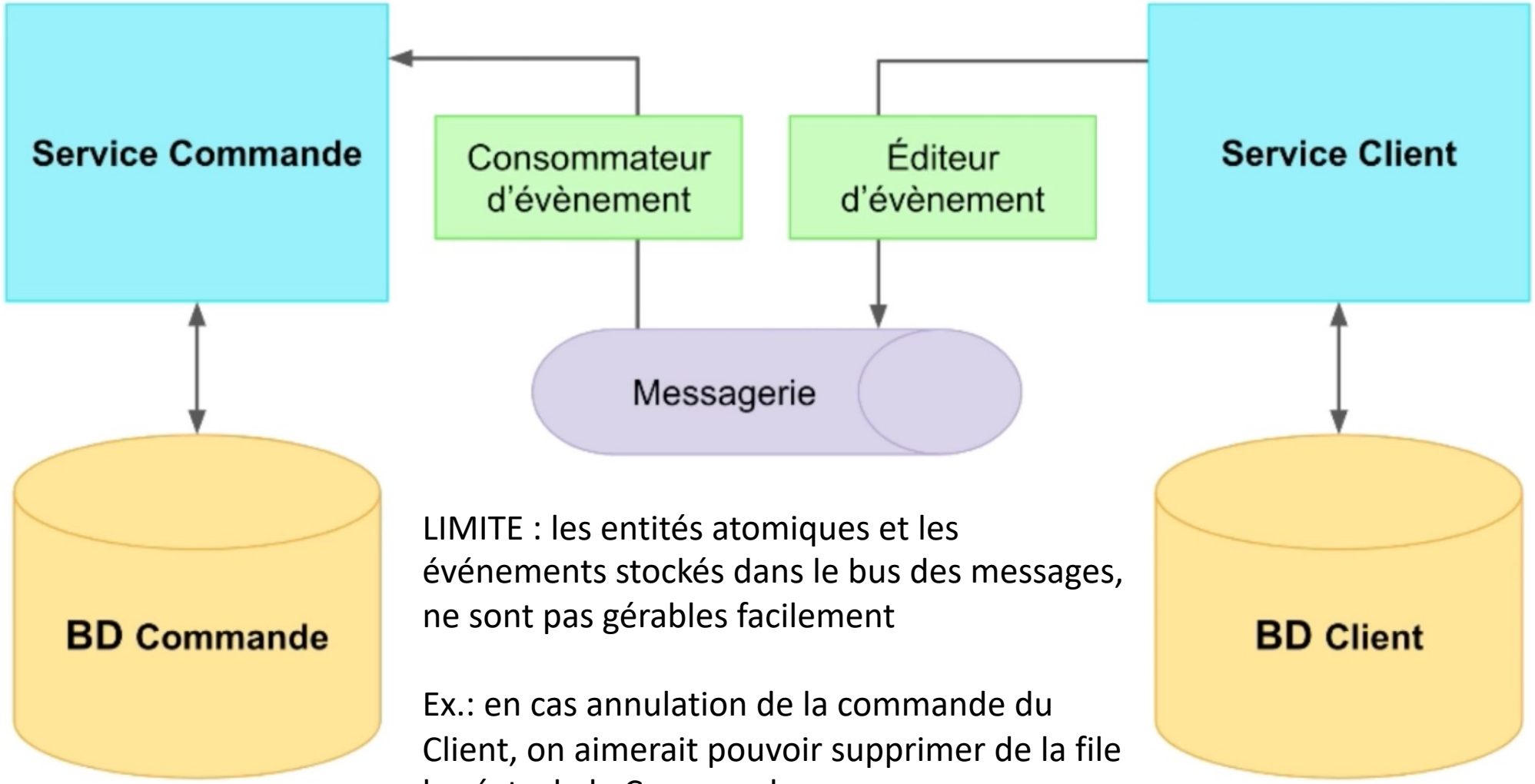
- La communication ou l'échange de données n’est possible qu’à travers un ensemble d'API bien définies.
- Exemple :
  - Une BD Commande pour le service Gestion des Commandes
  - Une BD Produit du microservice gérant le stock des produits disponibles.
- Ces 2 services sont liés :
  - Le Stock transmet à la Commande la disponibilité du produit
  - Une fois effectuée, la Commande informe le Stock de la quantité à enlever

Difficultés : la synchronisation

- En cas de dysfonctionnement de l'un ou de l'autre, il faut pouvoir interrompre la commande et garder une information fiable sur l'état du stock
  - A partir de quand / comment décide-t-on d'un dysfonctionnement ?

- Si les contextes des MS sont mal délimités, et c'est souvent le cas en général, les microservices auront besoin de données des uns les autres pour implémenter les traitements, ce qui va conduire à des interactions de type *spaghetti* entre les services de l'application
  - Découplage plus difficile qd on migre une grosse application monolithique vers une archi MS que pour une NOUVELLE application

Une solution : l'architecture orientée événement



LIMITE : les entités atomiques et les événements stockés dans le bus des messages, ne sont pas gérables facilement

Ex.: en cas annulation de la commande du Client, on aimerait pouvoir supprimer de la file les évts de la Commande…

Image : JC Simo, Udemy

Modèle d'une BD partagée

- Par 2 ou 3 microservices (pas plus), pour permettre les évolutions
- Approche moins radicale
- Cas à préférer lors d'une migration d'appli monolithique
- Aide à implémenter les transactions ACID des archi MS
  - Atomique, Cohérente, Isolée, Durable (cf SAGA)
- Limites :
  - Supprime beaucoup des avantages des microservices : *par ex., les modifications des schémas des tables doivent être faits de façon coordonnée entre les différentes équipes de développeurs des MS*
  - Conflit d'exécution aux mêmes données
  - Parfois considéré comme un anti-pattern

Pattern CQRS

Command Query Responsibility Segregation

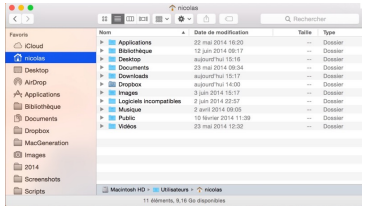
CQRS : Vue d'ensemble

- **Sépération des Commandes (CRUD) des Queries (lecture) sur les données**
- Dans une architecture microservices, les requêtes ne sont pas aussi simple qu'en *n-tier*, puisque les données sont réparties sur plusieurs services :
  - La lecture agrège souvent plusieurs entités issues de plusieurs BD
  - L'écriture met à jour des entités

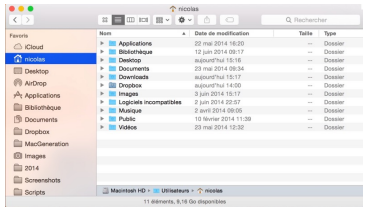
- CQRS vise simplement à séparer :
  - les flux d'écriture (les **commandes** : opérations, requêtes ayant un effet de bord sur les données, par ex. un *post)*
  - des flux de **lecture** des données (la *query*)

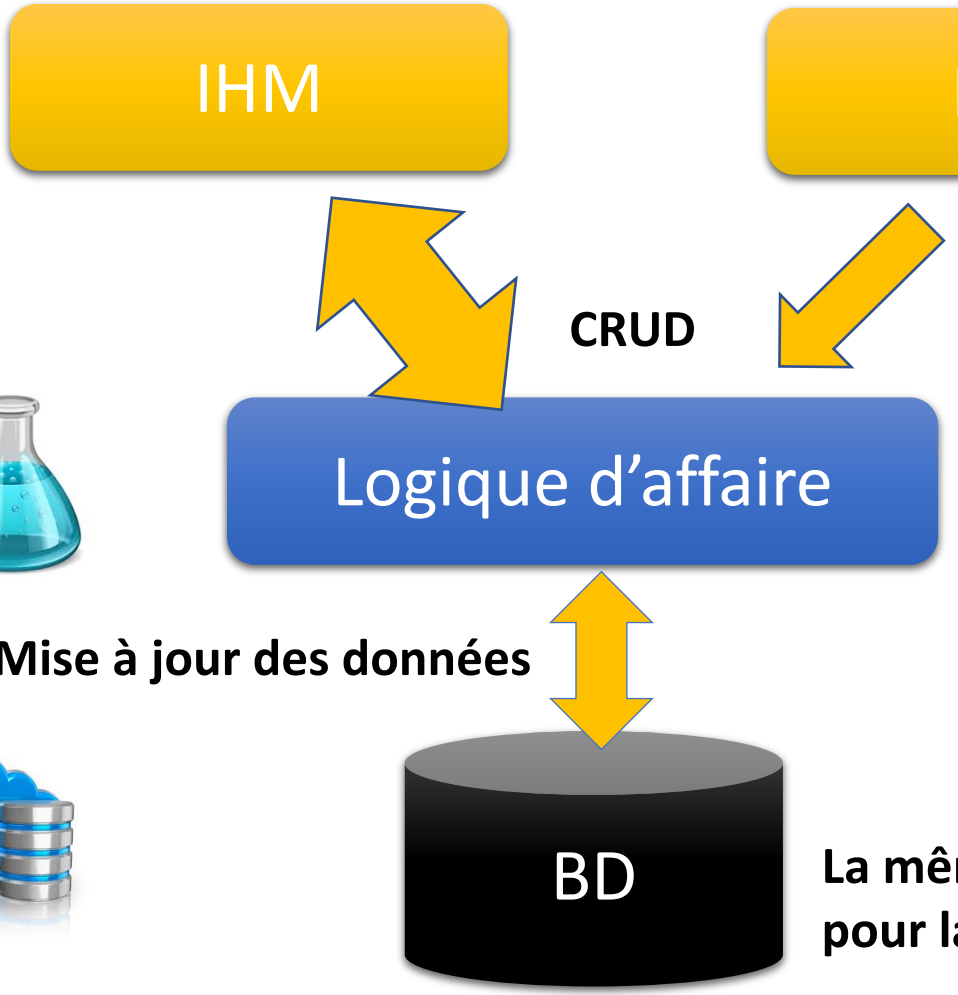
Système classique

Client A



Client B

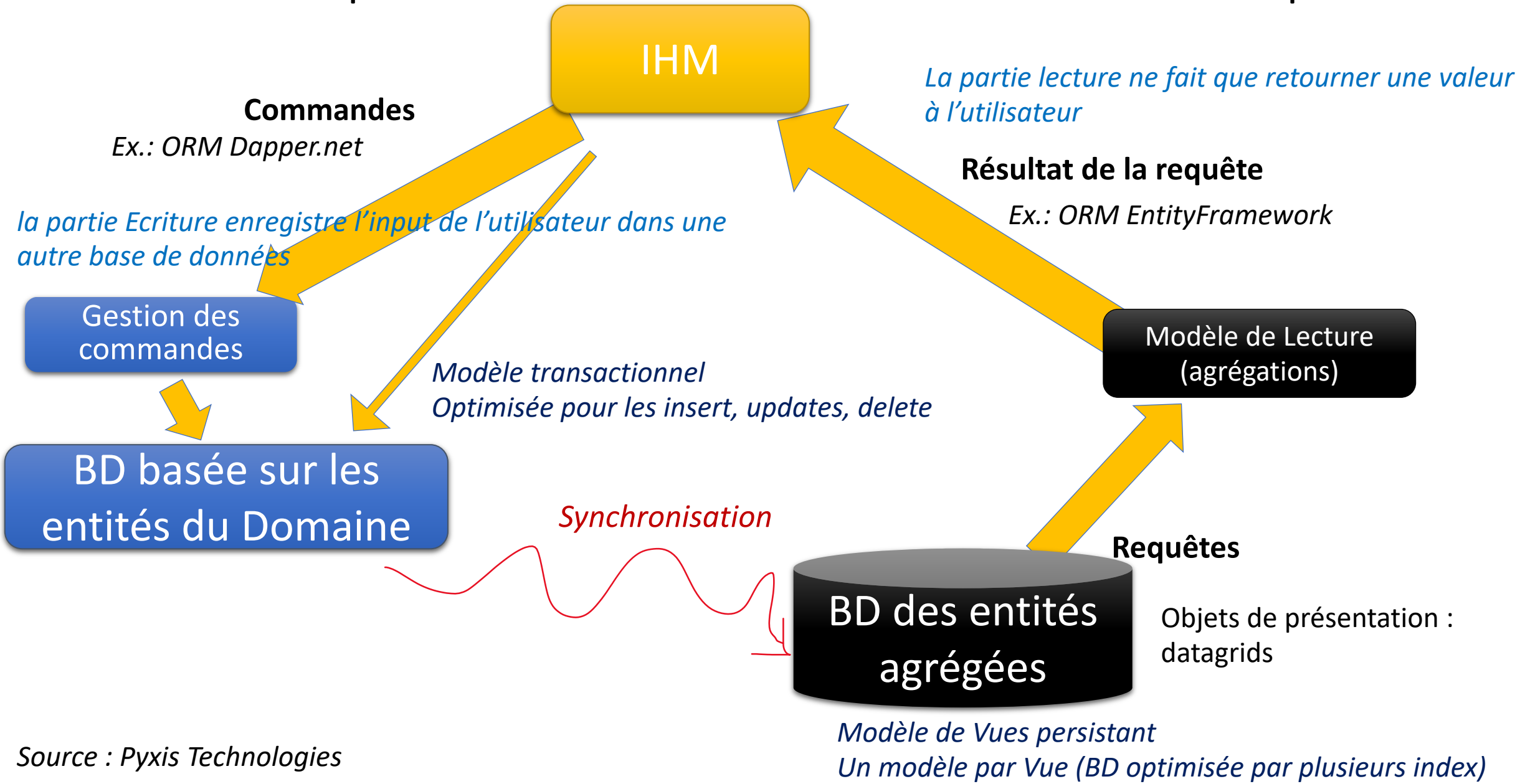




- Accès concurrent à gérer
- Impossibilité d'optimiser Lecture et Ecriture en même temps
  - ex.: augmentation des index pour la L
  - Consensus : aucune réelle satisfaction, ni en L ni en E

La même base de données est utilisée pour la lecture et l'écriture de données

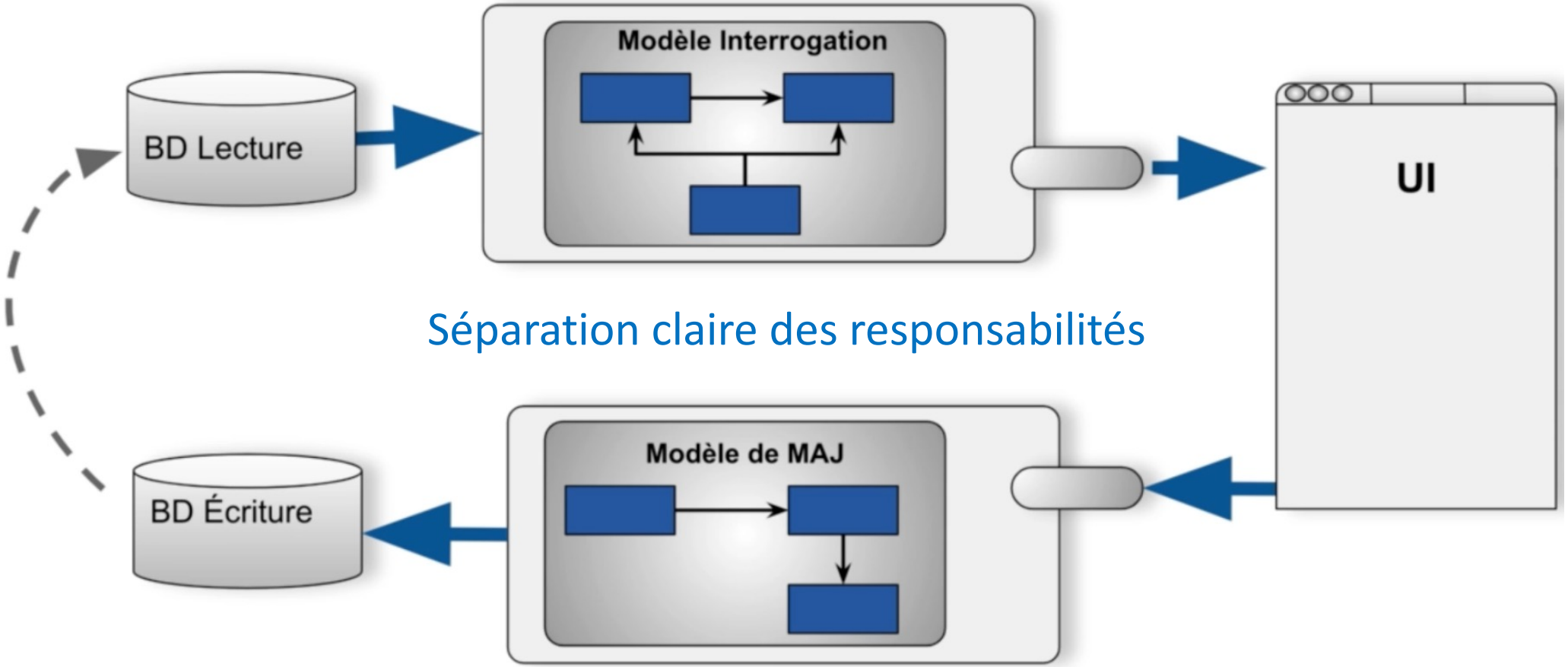
CQRS : Séparation des commandes et des requêtes



Source : Pyxis Technologies

CQRS : 2 microservices indépendants

Service de consultation : fluide, rapide, importants accès concurrents possibles, scalabilité horizontale en cas de pic de consultations



CQRS : pro / cons

Avantages :

- Supporte de multiple vues dénormalisées scalables et performantes
- Améliore l'objectif des MS de séparation des intentions (*segregation of concerns*) = la lecture et l'écriture
- Nécessaire dans une architecture dirigée par les événements *(event driven architecture)*

Inconvénients :

- Augmente la complexité
- Duplication de code potentielle
- Délai de réplication/visualisations cohérentes

Pattern Event Sourcing

Nécessairement associé à CQRS

Stocker les évènements

- L'*event sourcing* consiste à stocker tout ce qu'il s'est passé dans un système **sous forme d'événements** dans le but de conserver tout changement du modèle
  - Séquences d'événements
- Dans un exemple de site *e-commerce*, on créerait des évènements 'Panier Créé', 'Panier Modifié', 'AdresseClient Modifiée', 'Commande Effectuée'
- Ces événements seraient placés dans un *EventStore*, tel un *gestionnaire de versions* des événements.
- On pourrait alors facilement **revenir à un état antérieur** des données.

EventSourcing : pro/cons

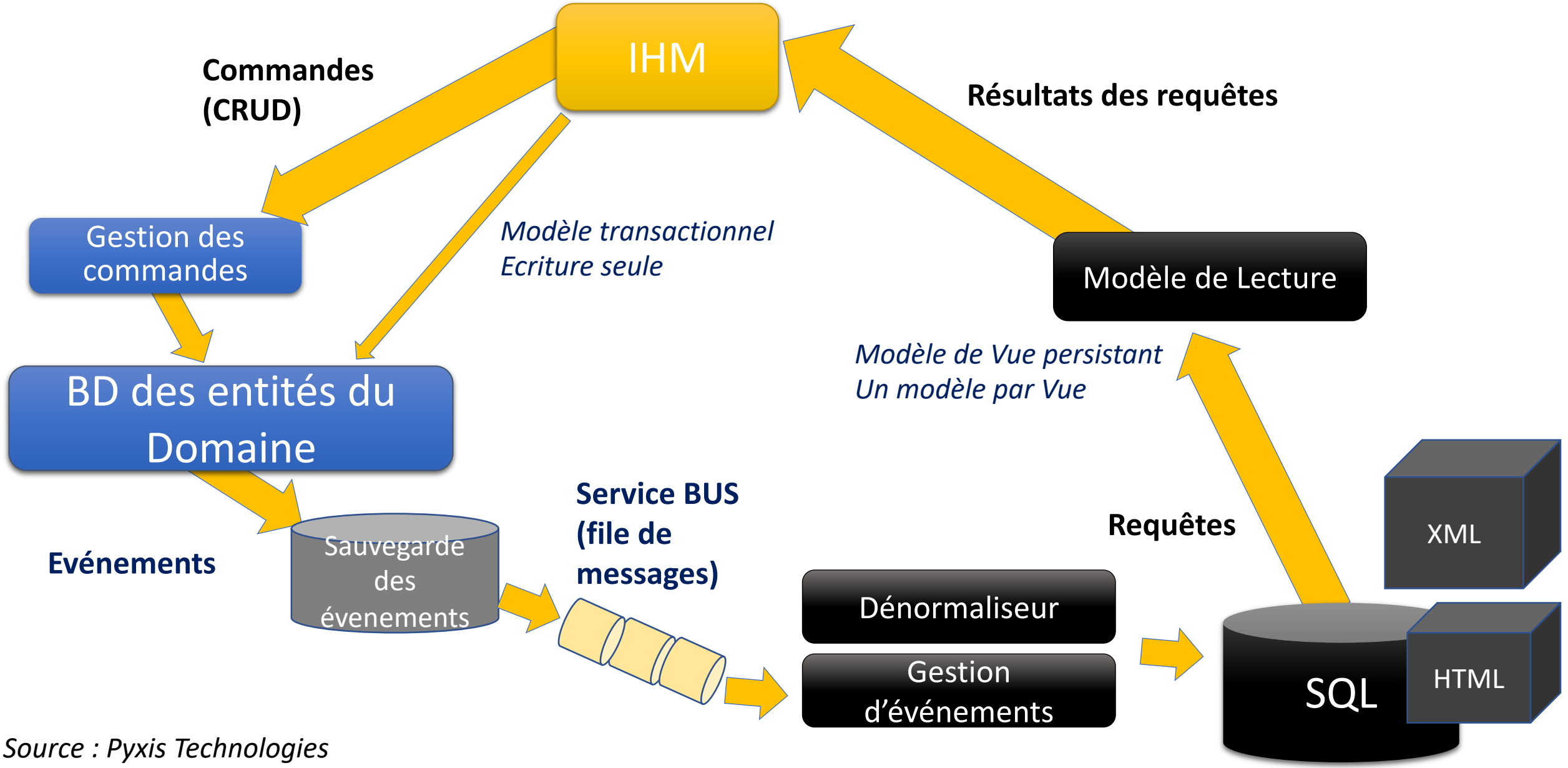
- Avantages :
  - Permet de publier des événements **de manière fiable** à chq changement de l'état de l'objet cible
  - On dispose d'un **journal d'audit fiable** des modifications apportées aux données
  - Stocke des séquences d'évts et pas les objets : évite en partie **les incohérences des données** dans les tables relationnelles
  - Rend possible des requêtes temporelles :
    - Déterminer un état d'une entité à un instant donné
  - Couplage faible entre les entités
    - Elles ne communiquent que par des événements

- Inconvénients :
  - Nécessite une courbe d'apprentissage ! Change la façon de gérer les données
  - Capacité de stockage des événements : suffisante
  - L'entrepôt des événements est un peu difficile à requêter pour reconstruire les états des entités, mais CRQS permet de faciliter cette tâche

Entrepôt des événements

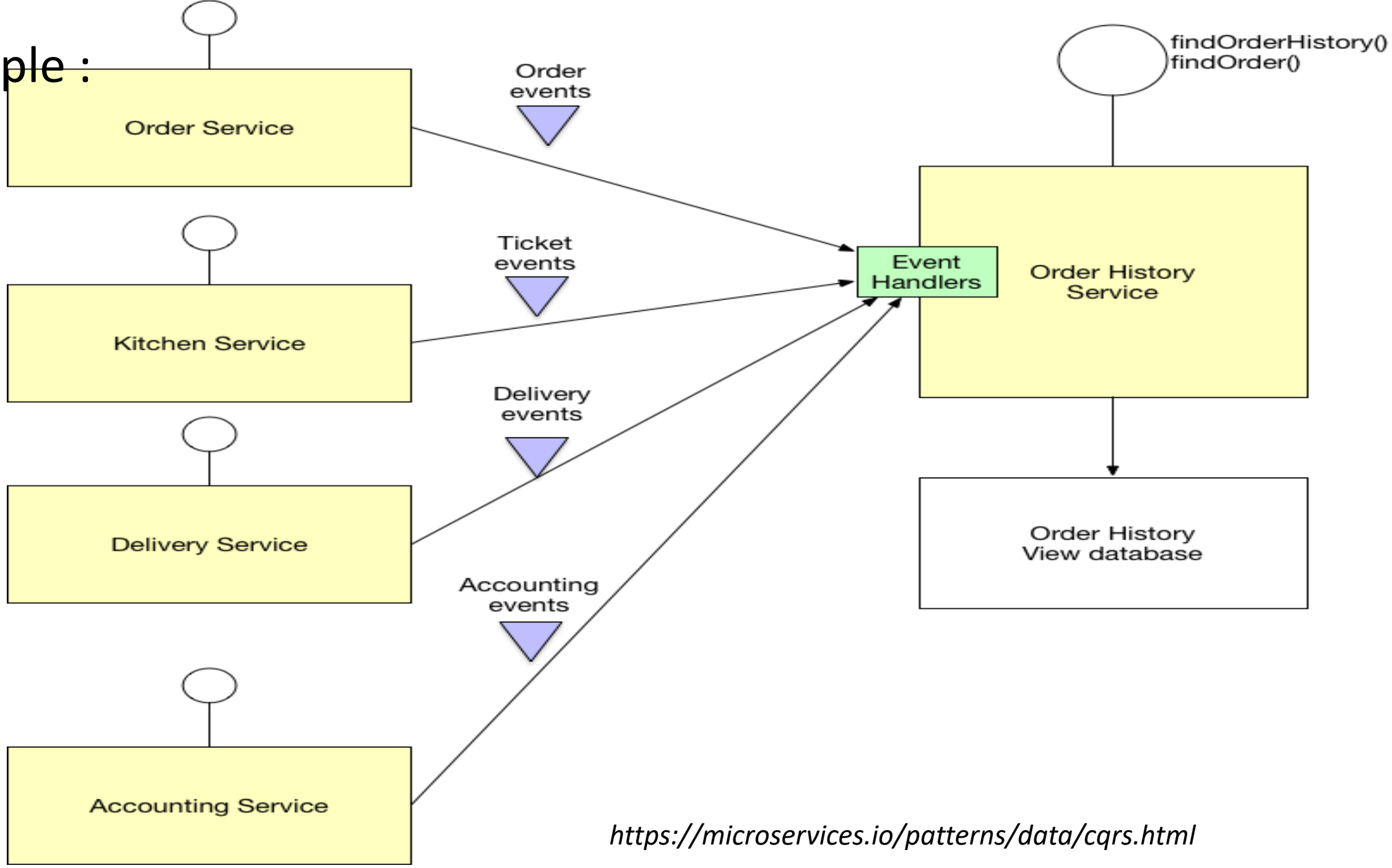
- C’est un microservices avec une API permettant aux services de s’abonner aux événements

CQRS + EventSourcing



Source : Pyxis Technologies

Exemple :



https://microservices.io/patterns/data/cqrs.html

Pattern SAGA

Gestion des transactions

Pattern SAGA : gestion des transactions

- Gérer la cohérence des données en cas de **transactions** par plusieurs microservices
- Une transaction = regroupement logique de *n* opérations
- Un événement = un changement d'état produit sur une entité
- Une commande contient toutes les informations nécessaires pour effectuer une action ou gérer un événement
- Architecture microservices : **transactions ACID**
  - Atomique, Cohérente, Isolée, Durable (cf SAGA)

Les transactions au sein d'un même service sont ACID
(car toutes sur la même BD)

Transaction ACID

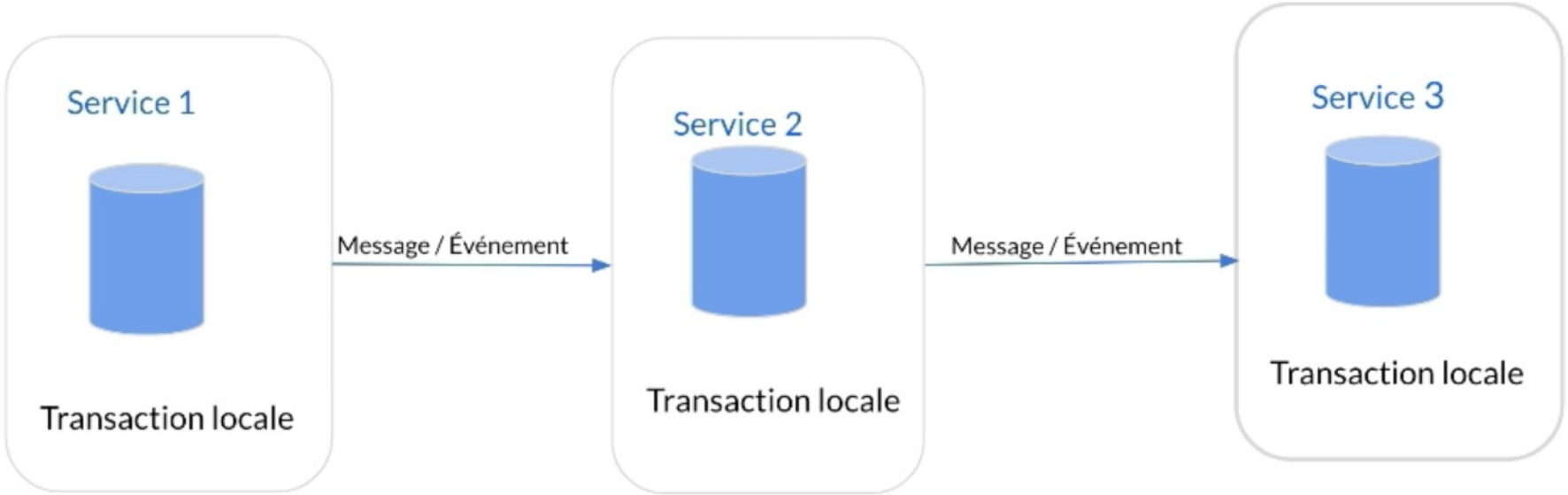
- ATOMIQUE : ensemble indivisible et irréductible d'opérations qui doivent toutes être exécutées (ou pas)
- COHERENCE : la transaction fait passer les données d'un état valide à un autre état valide. Ne laisse jamais une données dans un état intermédiaire
- ISOLEMENT : les transactions exécutées simultanément donnent le même état des données que si elles étaient appliquées séquentiellement
- DURABILITE : les transactions qui sont validées restent valides même en cas de panne (système, réseau)

Cohérence des données

- Le mécanisme de **gestion des transactions distribuées** peut être utilisé
  - Ex.: protocole de validation en 2 phases (2-phase commit, 2PC)
- Mais cela exige que tous les participants à une transaction donnée effectuent un *rollBack* ou un *commit* avant l'exécution de la transaction
- Fonctionne **pour les BD relationnelles**, quid des autres systèmes de persistance présents dans les AMS ?
- Pose des **problèmes de synchronisation** de la communication inter-processus des Systèmes d'Exploitation

SAGA : gestion des transactions

- **Répond au problème de communication interprocessus ou de limitations entre les transactions**
- **SAGA = une séquence de transactions locales** pour répondre à une seule requête utilisateur

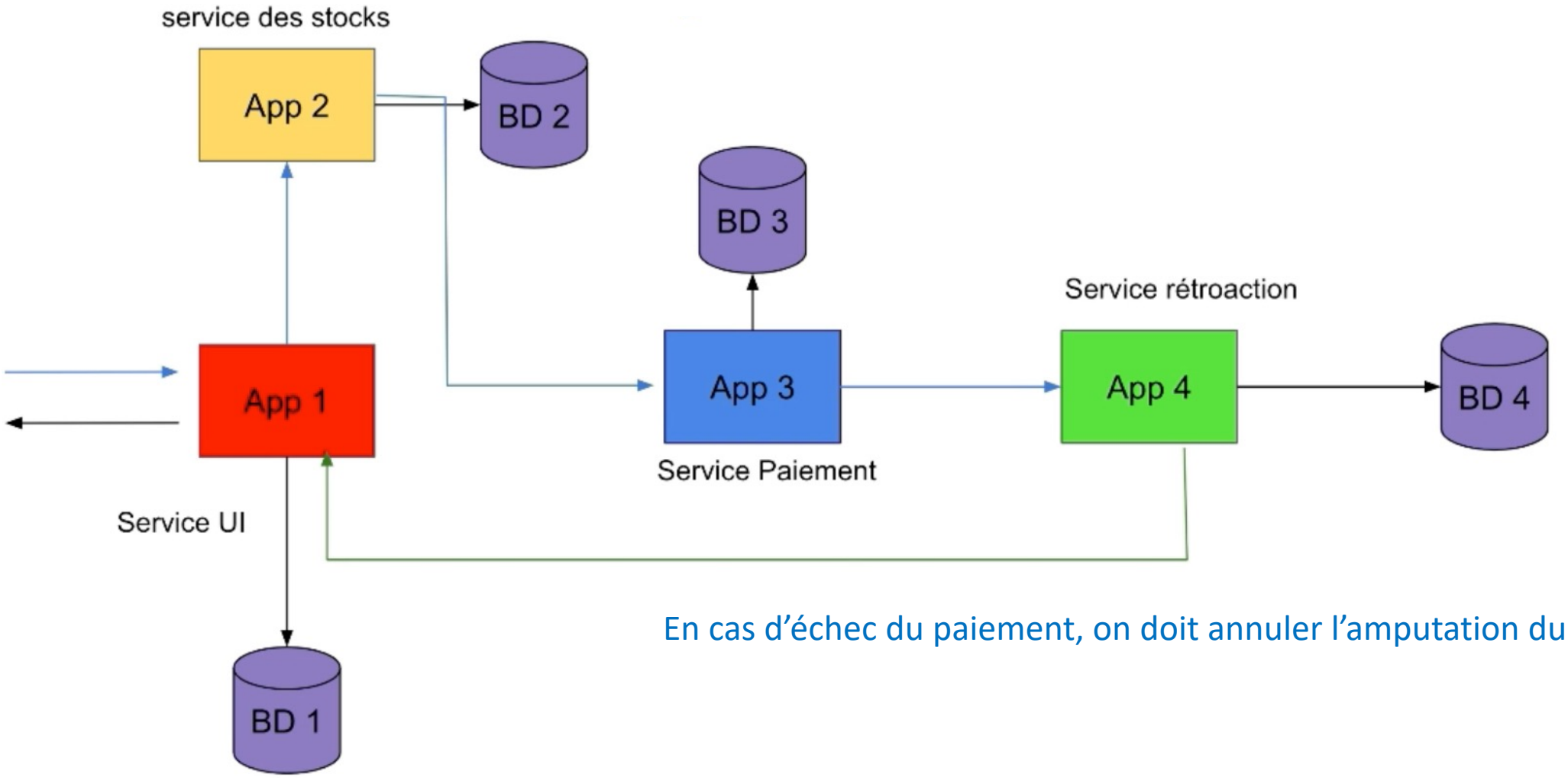


Fonctionnement SAGA

- Une transaction locale génère un message ou un événement capable de lancer la 2^{e} transaction locale, etc.
- En cas d'échec d'une transaction locale, SAGA effectue des transactions de compensation qui annule les transactions précédentes
- DEF **transactions de compensation** = la réciproque d'une transaction qui fait l'effet inverse
- DEF **transaction pivot** = le point de départ ou de non départ (GO ou NOGO point)

Illustration : un achat

Une seule requête qui couvre 4 services.
ATOMICITE : soit l'achat est réalisé, soit il échoue.
Pas d'issue transitoire du processus.

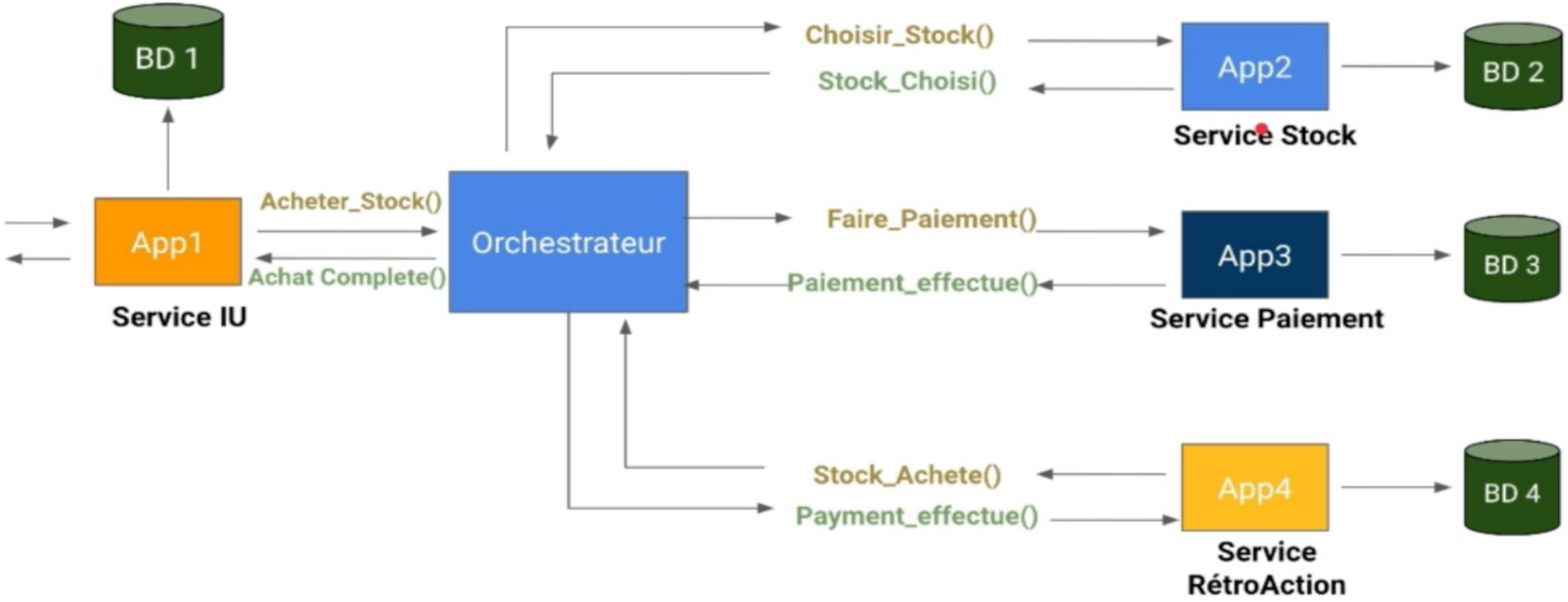


En cas d'échec du paiement, on doit annuler l'amputation du stock.

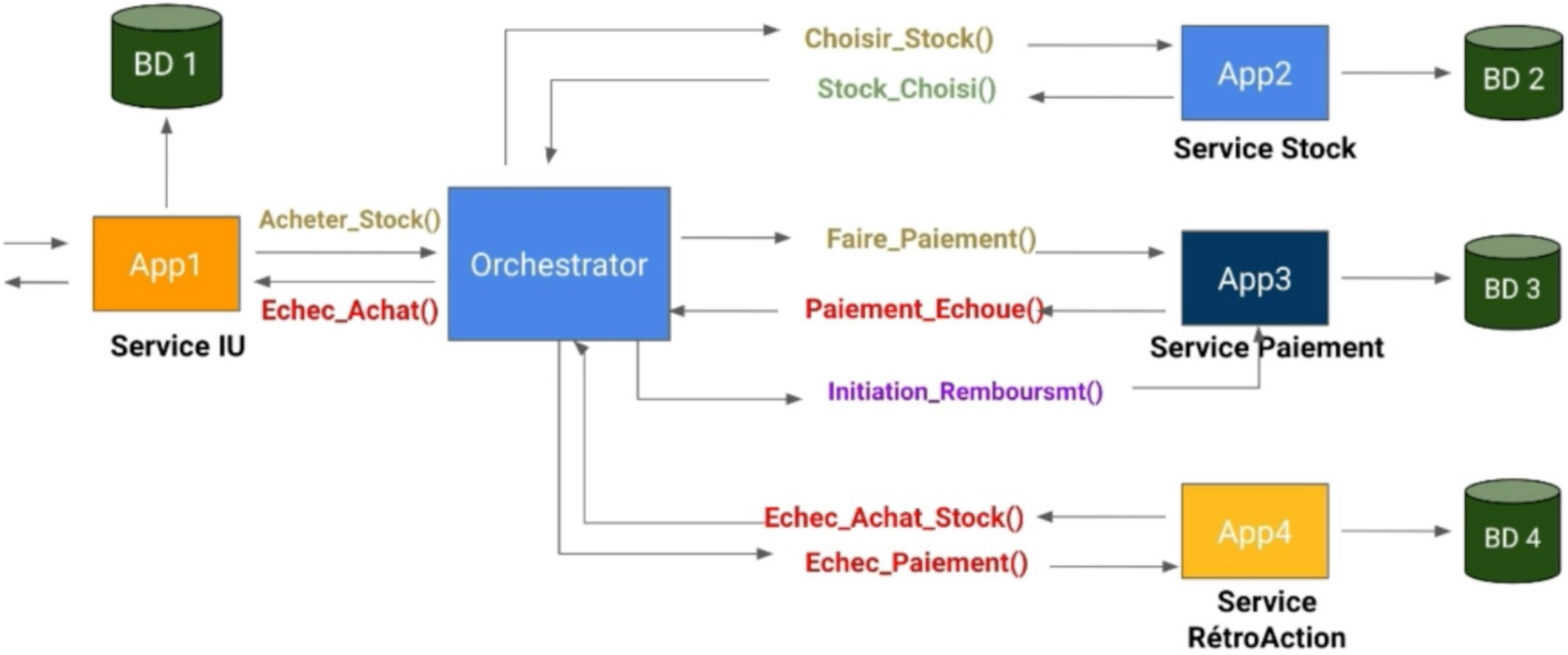
Chaine de transactions locales

- Chacune modifie la donnée d'un seul service
- Recommandé pour des transactions longues, de plus de 5 microservices
- 2 implémentations possibles :
  - Basée sur les événements : **chorégraphie**
    - Ex.: StockChoisi, EffectuerPaiment, PaiementConfirmé, AchatConfirmé
    - Plusieurs MS peuvent écouter un même événement
    - Souvent : un évt est associé à un domaine
  - Basée sur les commandes : **l'orchestration**
    - Les MS n'interagissent pas directement les uns avec les autres, c'est un orchestrateur qui gère les *commandes*

Implémentation de SAGA avec un orchestrateur



Gestion des échecs



Orchestrateur vs Chorégraphie

- Si peu de microservices (3 à 5), **Chorégraphie** plus simple à implémenter et plus facile à comprendre
- Mais risque de **dépendances cycliques** si beaucoup de services
  - Echecs de transactions durs à gérer

- **Orchestrateur** : souvent utilisé avec des outils supplémentaires, ajoutent de la complexité
  - Mis on peut ajouter d'autres services à l'orchestrateur

SAGA : pro/cons

- **Avantages** :
  - Permet la consistance des données en cas de transactions distribuées dans une AMS
  - Les MS ne sont pas bloqués les uns par rapport aux autres.

- **Inconvénients** :
  - L'orchestrateur = surcharge de travail
  - Ajout de transactions de compensation : complexe
  - Transactions distribuées : toujours plus difficiles à débugger
  - On doit penser aussi au cas où l'action de compensation échoue

SAGA : quels usages ?

- Transactions distribuées sans couplage étroit
- Quand on a besoin d'annuler ou compenser quand la séquence échoue

SAGA moins adapté si :

- Les transactions sont étroitement couplées
- S'il existe des dépendances cycliques

Bonnes pratiques DevOps

Conception d'architectures

Systèmes de fichiers distribués centralisés : problématique du cache

Comment assurer la consistance du cache ?

- Le cas UNIX : il gère les états d'un fichier → si un usager ouvre un fichier en lecture/écriture, alors tous les autres usages ne peuvent plus l'ouvrir qu'en lecture seule
- Pour un client NFS v4 (Network File System) ?
  - Le client vérifie la formule logique (T-Tc<t)v(Tmclient=Tmserveur) en deux temps :
  - Il regarde d'abord -*sans envoyer de requête au serveur*- si (T-Tc<t) est valide (c'est à dire si cela fait moins de t secondes que ledit bloc a été vérifié, où t€[3;30] pour un fichier et t€[30;60] pour un répertoire)
  - Puis le client demande au serveur quand est-ce que ce bloc a été modifié en dernier (noté Tmserveur) afin de pouvoir comparer cette réponse à son Tmclient

Consistance du cache avec AFS

- Lorsqu'un client modifie un fichier sur le serveur, le serveur envoie une notification à tous les autres clients pour leur dire que leur copie cachée de ce fichier n'est plus valide et donc que ces clients devront récupérer la nouvelle version du fichier lorsqu'ils voudront y accéder.

- Pour s'assurer qu'aucune notification n'ait été perdue, des notifications sont envoyées régulièrement aux clients (toutes les quelques minutes) pour indiquer si les fichiers cachés sont valides ou non.

Pattern Mise en Cache

- Dans le cadre de la gestion des données, la quantité et la taille des échanges de données entre microservices via les appels API peut finir par coûter très cher.
- Il est ainsi recommandé de mettre en place un ou plusieurs **systèmes de mise en cache** :
  - *in-memory*,
  - *key-value store*,
  - Etc.

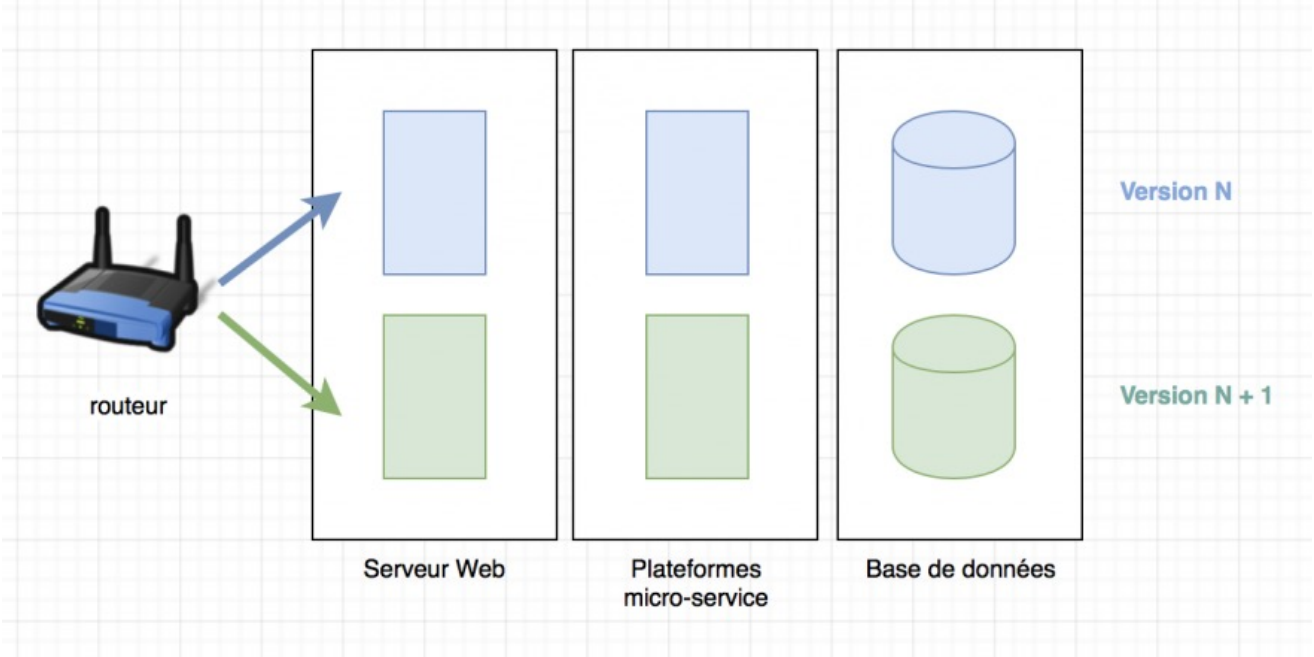
Chiffrement des données : principe Zero Trust

- Le chiffrage des données est possible à la fois au niveau de la base de données et au niveau des fichiers.

- Il existe deux types de chiffrement, complémentaires :
  - **Le chiffrement en transit,** lors du passage des données d'un système à un autre : points de terminaison en HTTPS et *backends* pourvus de certificats afin de garantir aux clients leur identité.
  - **Le chiffrement au repos**, à l'intérieur même des bases de données.

Le Pattern Blue/Green Deployment

- Il permet de déployer une version n+1 (*Green* ici) sur l'environnement de la version n (*Blue*)
  - Basculer automatiquement les utilisateurs sur la nouvelle version
  - Rollback vers la version n en cas de problèmes découverts
- Permet le concept du Zero Downtime Deployment (ZDD)



Blue/Green Deployment Pattern

- Un classique des patterns DevOps, très répandu
- **Objectif** : déployer sans jamais interrompre les services
  - C'est le concept du **Zero Downtime Deployment** (ZDD)
  - Livrer régulièrement de façon **transparente** et en mode **quasi instantané**
- Existe depuis un certain nombre d'années
  - Mais cela avait un coût car à l'époque, serveurs **physiques**

- Avantage : déploiement progressif possible
  - Permet de router x% de ses utilisateurs sur le Green et le reste sur le Blue (cf Canary Pattern et Dark Launch Pattern)
  - On peut ainsi tester **si tout va bien avant de déployer** sur l'ensemble des utilisateurs

Les problèmes à traiter

1. Session utilisateur
2. Gérer la continuité des transactions en BD

Le Canary Pattern

- Objectif : tester les performances du **back-office** et déceler les éventuels dysfonctionnements
  - Des bugs
  - Une montée de charge exagérée apparente
  - Etc.
- Comment ?
  - Tester sur une **durée représentative** des modifications apportées
  - Ex. Facebook observe les valeurs de ses indicateurs **pendant 24h,** sur ses employés : si les résultats sont positifs, ils basculent tous les utilisateurs vers la nouvelle version

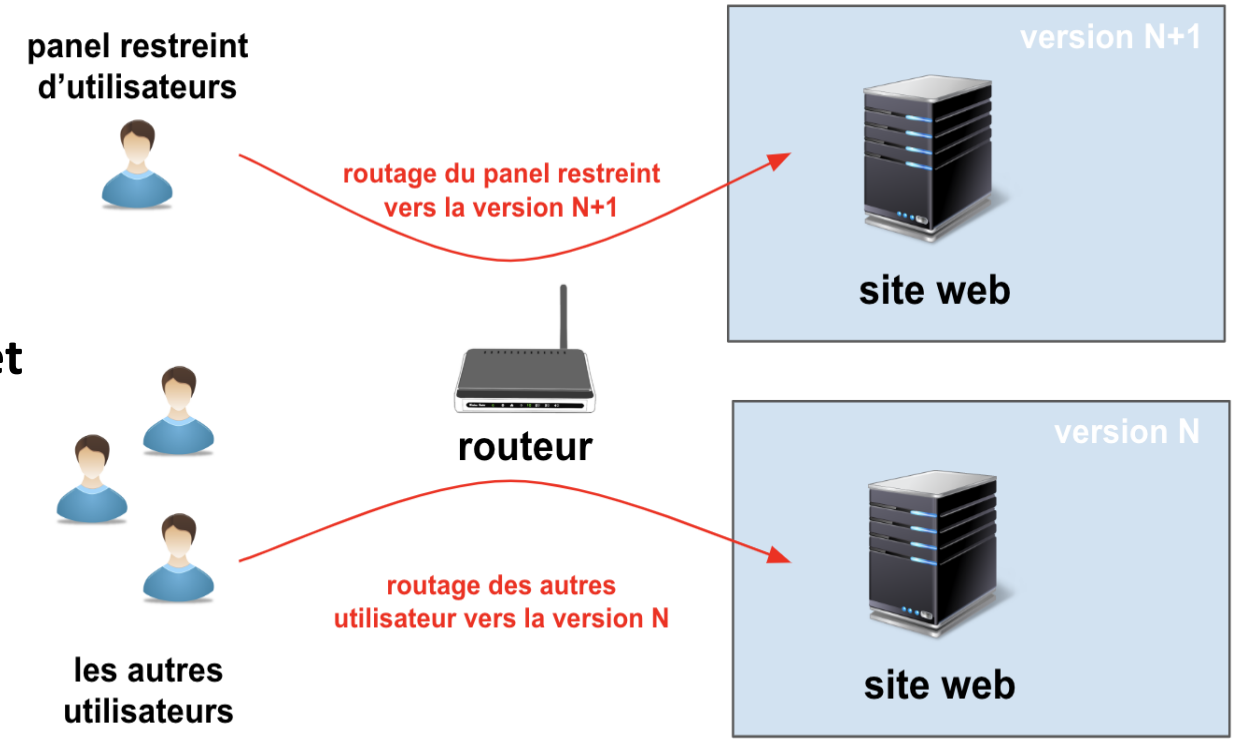
Le Dark Launch

- Objectif : tester le **comportement des utilisateurs** sur de **nouvelles fonctionnalités**.
- Comment ?
  - On définir un panel réduit d'utilisateurs (cobayes).
  - On les route vers la nouvelle version sans mise en avant des nouveautés ni d'aide, afin d'analyser le comportement **par défaut** des utilisateurs.
  - Repose sur un **ensemble de mesures** sur les comportements
- Quand ?
  - Très utilisée au sein d'une organisation qui fonctionne en *Lean startUp*
  - L'entreprise élabore une idée, la teste sur un panel d'utilisateurs, mesure les résultats et ajuste ainsi sa solution
    - Cela tant que les résultats ne sont pas satisfaisants.

Exemples de mesures utilisées en Dark Launch

Pour évaluer la performance de la nouvelle version :

- Télémétrie sur les **données de performance, vitesse de traitement, erreurs et événements de sécurité des systèmes ;**
- Taux d'utilisateurs ayant testé la fonctionnalité ;
- Taux d'utilisateurs ayant utilisé la fonctionnalité à 100% ;
- Feedbacks des utilisateurs.



Conclusion

Sur l’architecture logicielle

- L'architecture générale d'un logicielle doit être **choisie très tôt**, en phase d'analyse, puisqu'elle conditionne l'organisation du projet
  - Elle **structure** l'application
  - Requiert une certaine stabilité
- Mais elle doit pouvoir **évoluer** au fur et à mesure de la conception
- Il faut donc permettre le évolutions et repousser les décisions majeures le plus tard possible.

- **Plusieurs architectures** (3-tiers, MV-VM, microservices) peuvent être combinées au sein d'une même projet