ARCHITECTURE LOGICIELLE: DETAILS

III. Styles d'architecture

III.1 Appels et Retours

III.2 Couches

III.3 Flot de données

III.4 Architecture distribuée

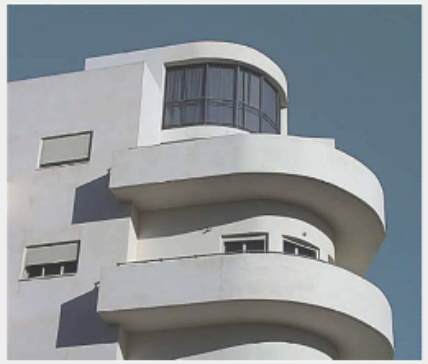
III.5 Architecture orientée évènements

III.6 Architecture centrée sur les données

III.7 Styles hybrides

III. Styles d'architecture

Notion de styles


- A l'instar de l'architecture traditionnelle, l'architecture logicielle peut se caractériser par des **styles**.
- Selon le résultat attendu, un système pourra
  - utiliser *plusieurs styles*
  - ou *imbriquer des styles* les uns dans les autres (*fractilité*)

Découpe générique hors style

Système => Système complet

└— **Sous-Système** *=> service business* (ex: Gestion des comptes)

=> unité d'exécution

└— **Composant** (ex: AccountService.jar)

=> unité de déploiement

└— **Module** *=> unité logique* (ex: Account + Transaction)

└— **Classe | Fonction** (ex: Account, Transaction)

styles principaux

- Architecture en appels et retours
- **Architecture en couches**
- Architecture en flot de données
- **Architecture distribuée**
- Architecture centrée sur les données
- **Architecture orientée objets** (déjà vu)

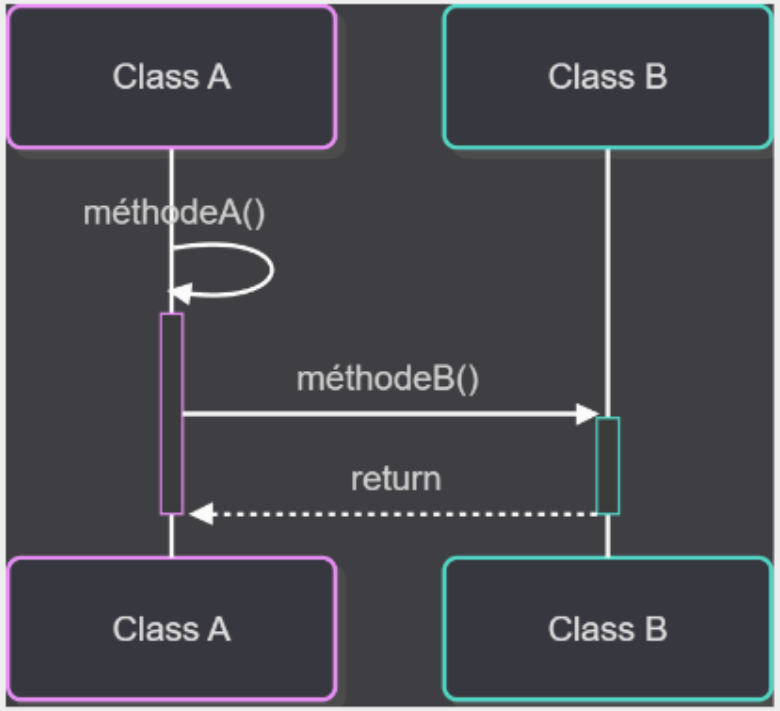
III.1 Appels et Retours

- Principe de **décomposition fonctionnelle de Niklaus Wirth**
  - décomposer la fonctionnalité initiale en sous fonctionnalités jusqu'à obtenir des problèmes simples
  - principe **K**eep **I**t **S**imple **S**tupid
- Caractéristiques:
  - *synchrone*
  - *hiérarchie d'appels* (pile d'appel LIFO)
- *+++* : **diminution de la complexité.**
- **---** : **diminution de la performance**

Appels et Retour: exemples

```
main()    # programme principal
 ├─ fonction1()    # sous-routine
 └─ fonction2()
     └─ fonction3()    # imbrication
```

pile d'appel



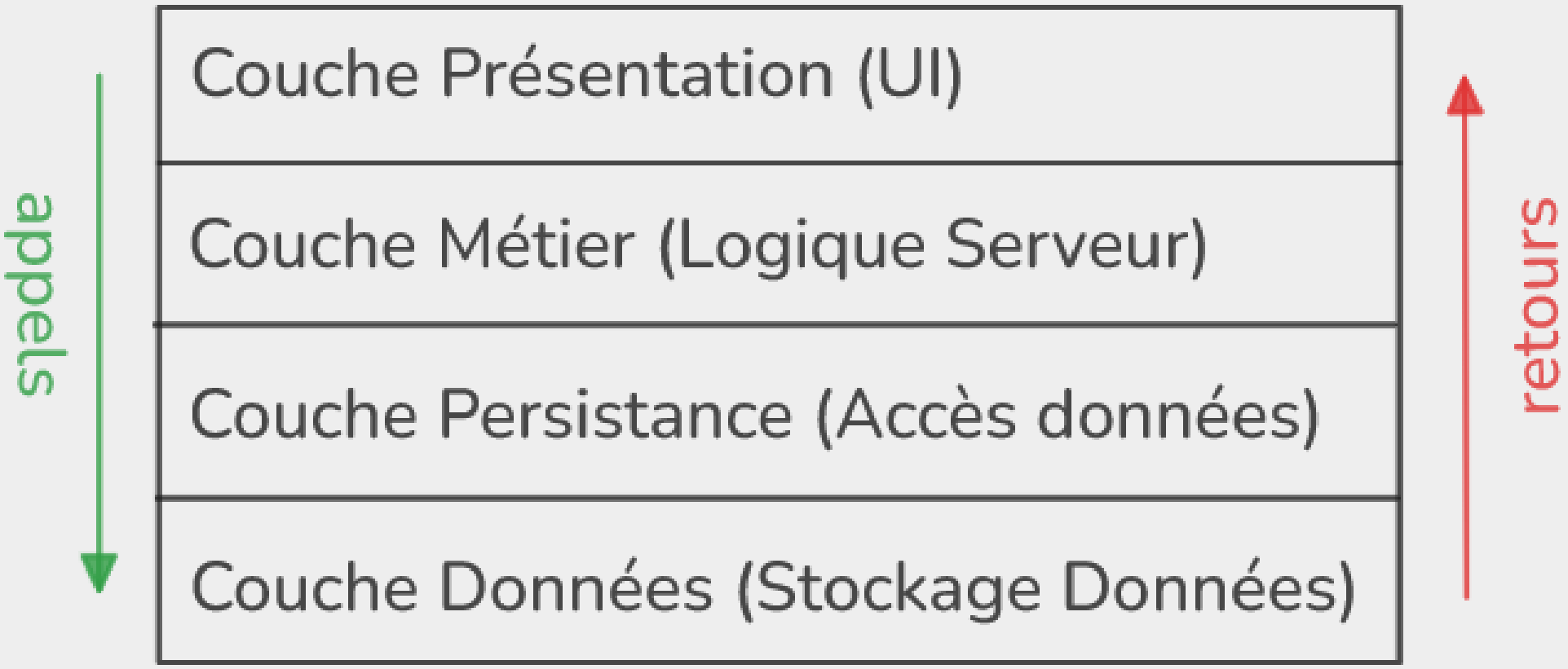
un service est une unité d'exécution (sous-système)
exécuté sur un processeur / processus isolé
et qui communique sur le réseau

un service n'est pas un élément d'architecture

III.2 couches

- Le système est décomposé en *couches horizontales*,
  - utilisant le **SRP** pour garantir une **responsabilité spécifique**.
- **Ré-utilisabilité** des couches inférieures par les couches supérieures, par *appels et retours*
  - *jamais dans l'autre sens !*
- **Découpe Technique**: persistance / métier / vue *(3-tiers)* / interlogiciels ... *(n-tiers)*

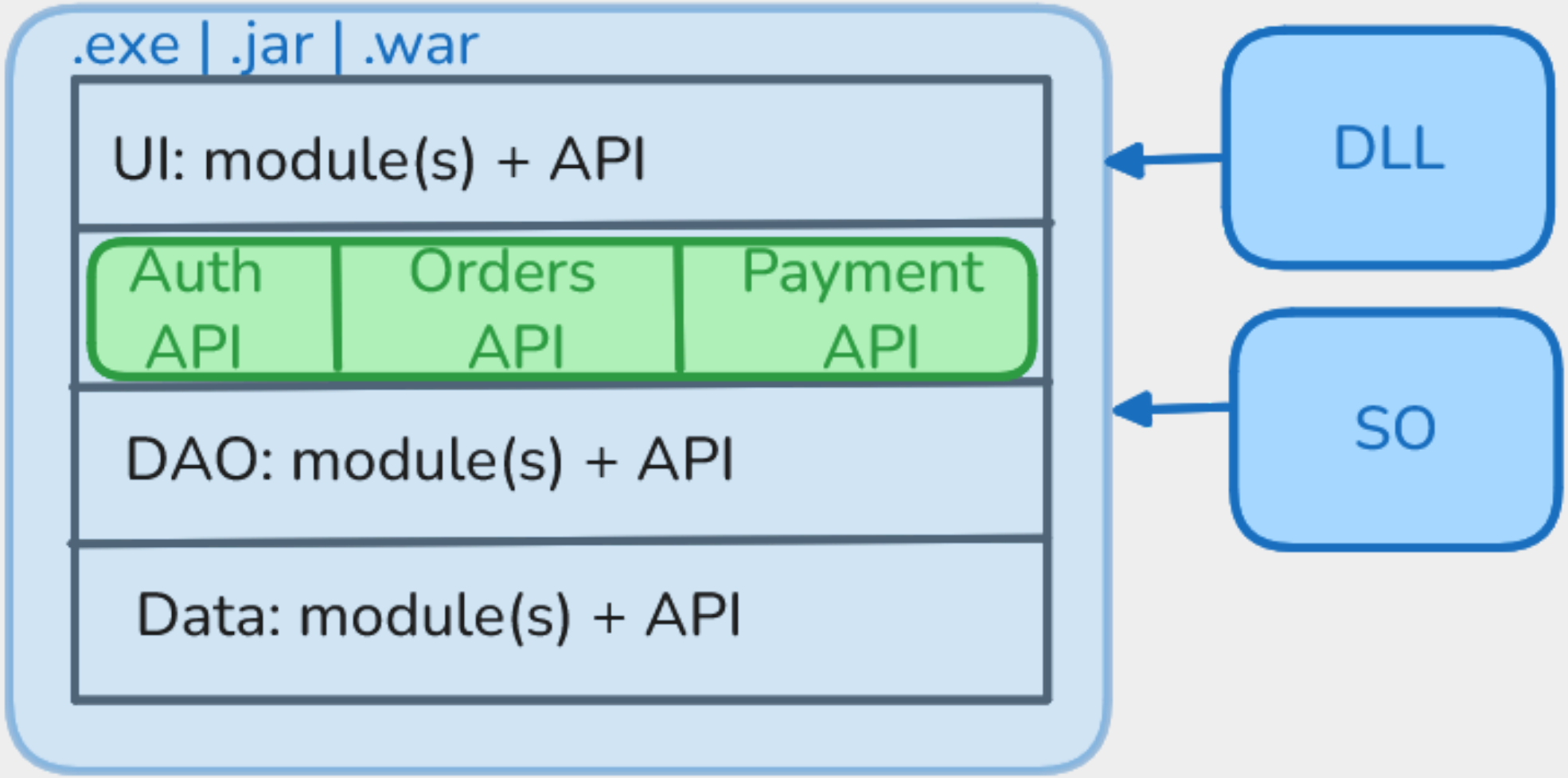
couches: schéma



disposition des couches: monolithe

- *toutes les couches* sont contenues dans un *exécutable unique*
- **monolithe**
  - *+++* facile à déployer
  - **---** difficile à faire évoluer
- **monolithe modulaire**
  - *+++* architecture propre avec périphs en tant que plug-ins*

“ *ex: application desktop* ”


.exe | .jar | .war
UI: module(s) + API
Auth API
Orders API
Payment API
DAO: module(s) + API
Data: module(s) + API
DLL
SO

disposition des couches: client-serveur

- **utilisation de services distants**

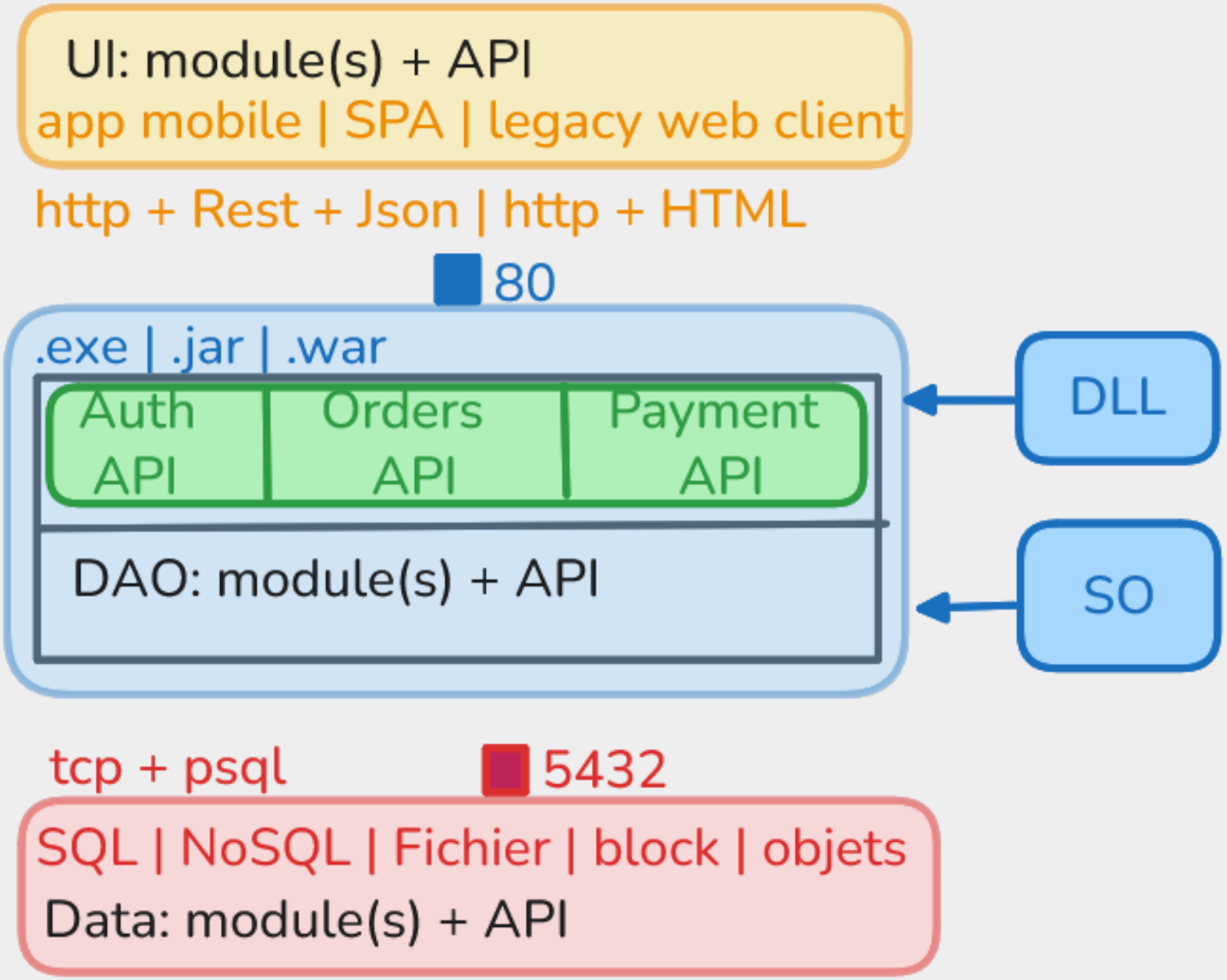
client lourd / web	serveur d'app	serveur SGDB
vue	logique métier	persistance des données

+++ couplage faible: code métier indépendant des périphs enfichables => **archi héxagonale**

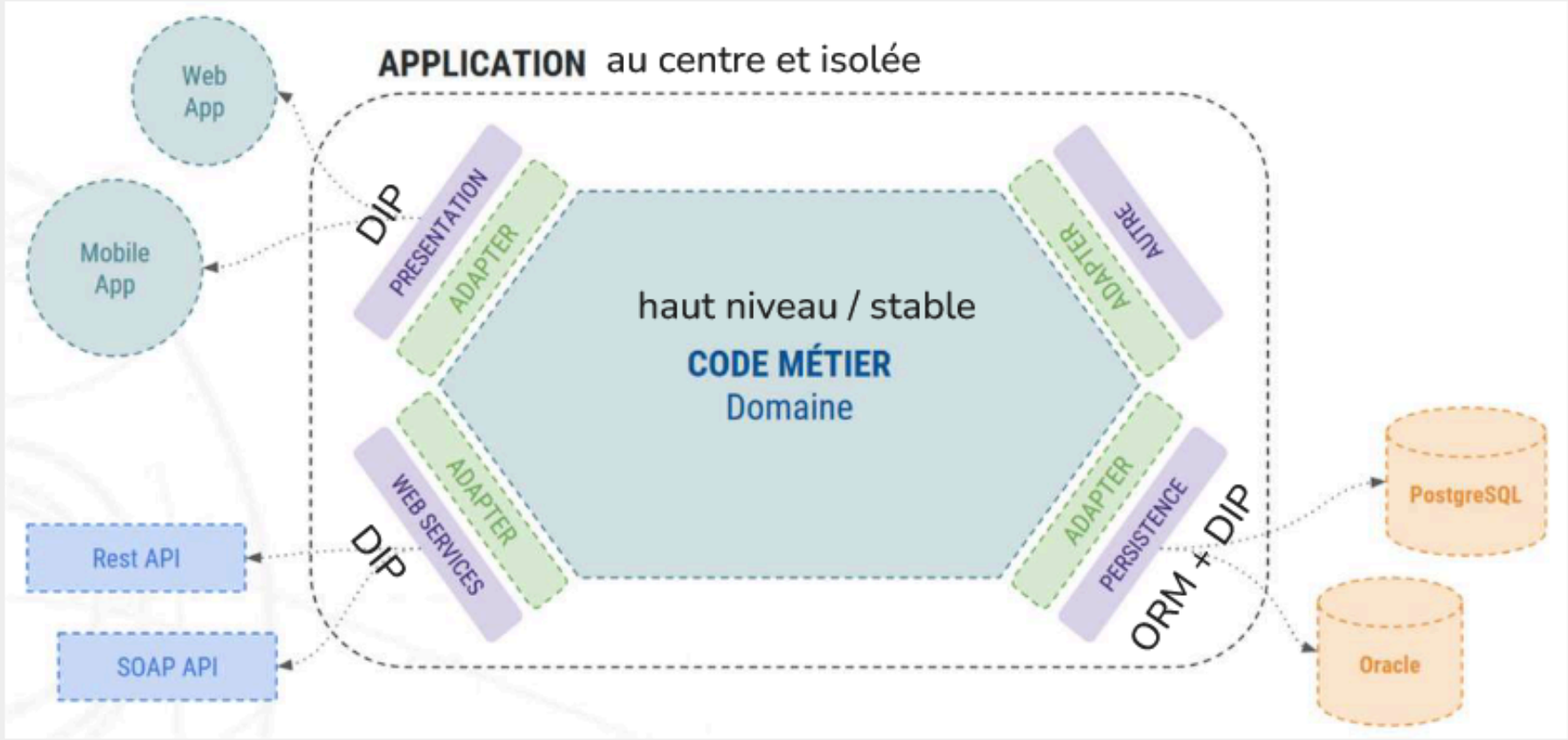
+++ séparation physique des responsabilités

--- *dépendance* au réseau et *latence* des appels distants

--- déploiement plus complexe


UI: module(s) + API
app mobile | SPA | legacy web client
http + Rest + Json | http + HTML
80
.exe | .jar | .war
Auth API
Orders API
Payment API
DAO: module(s) + API
DLL
SO
tcp + psql
5432
SQL | NoSQL | Fichier | block | objets
Data: module(s) + API

couches: architecture héxagonale



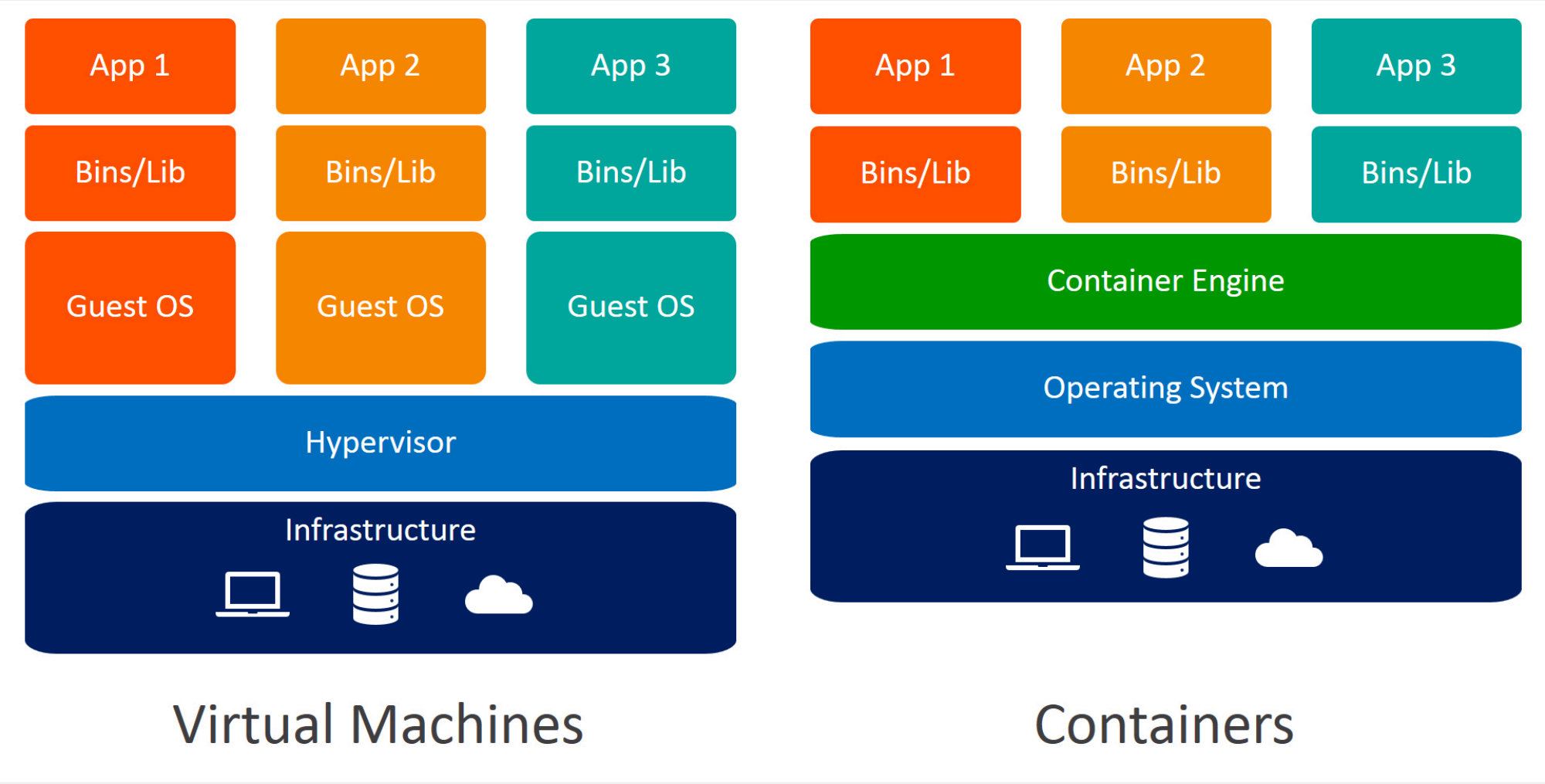
“ reformulation de l'archi propre pour les couches ”

couches: unité d'exécution

- utilisation de services
  - isolation des processeurs
  - et communication à travers le réseau (sockets)

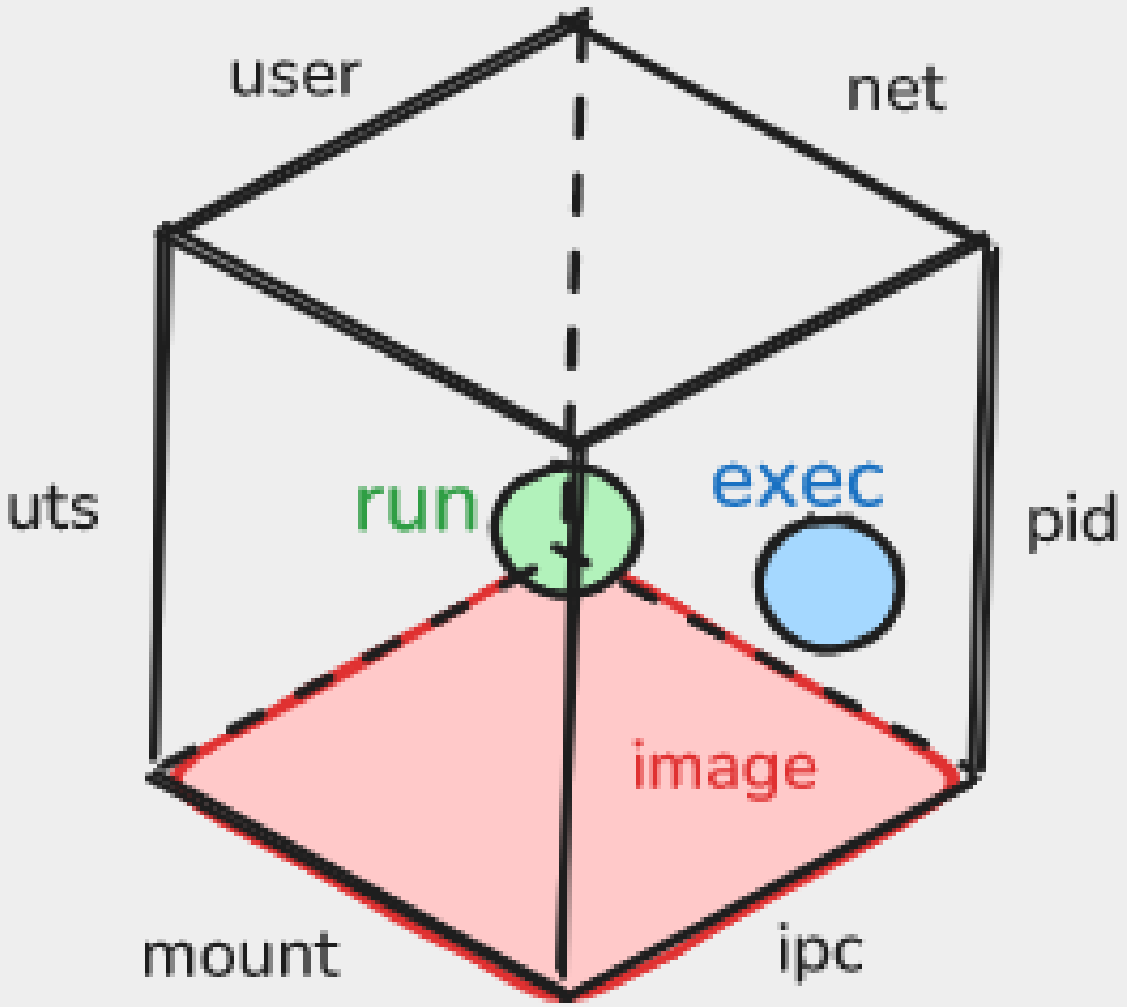
> **on peut aussi isoler les processus**

unité d'exécution: VM vs conteneur

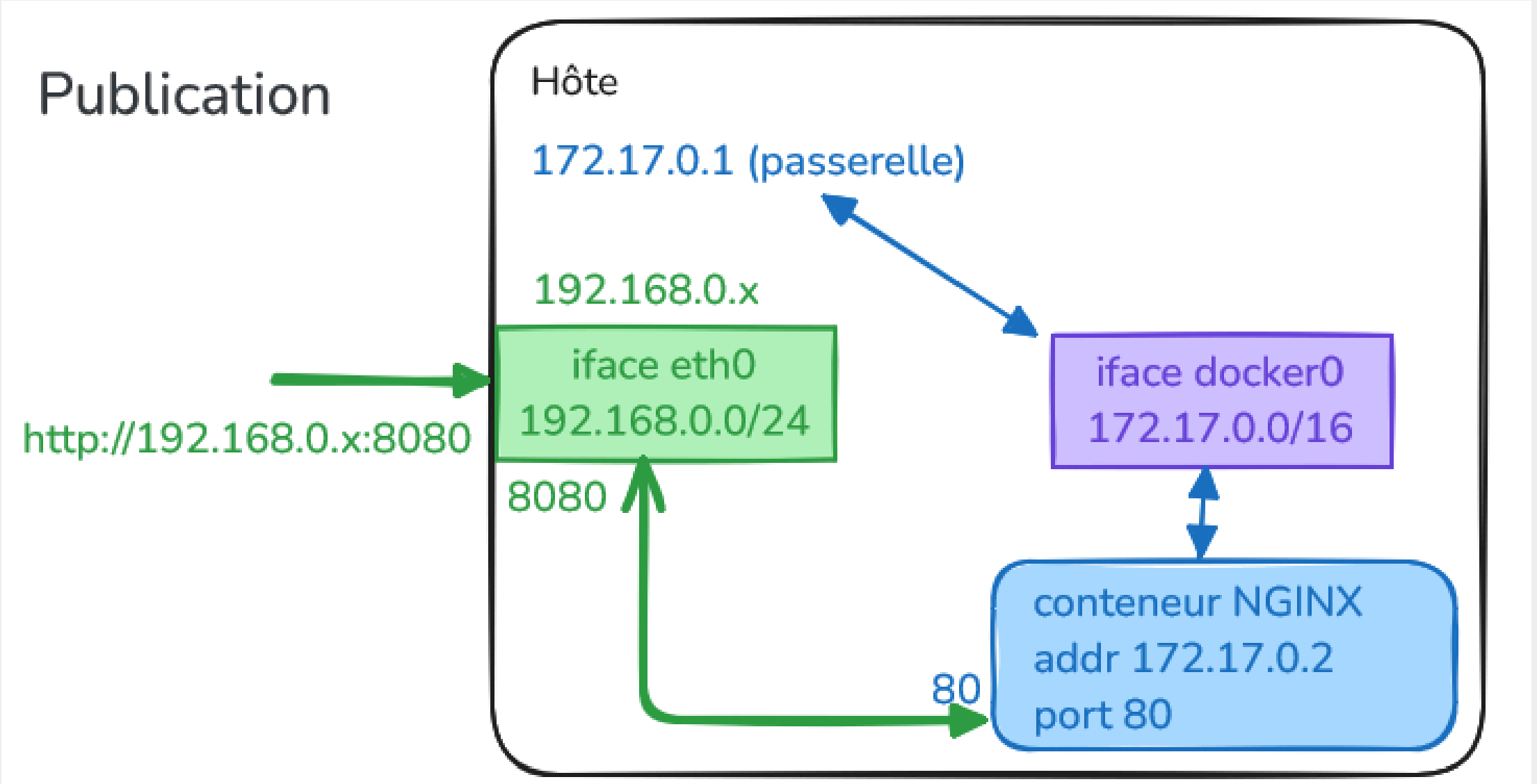


conteneur docker

- processus isolé
- par des **namespaces** du noyau linux
- assis sur un *système de fichier* **"image"** délimité
- communique sur le **réseau**



mise en réseau Docker



III.3 Flux de données

- Le système est décomposé en *composants de traitement*
  - connectés par des *canaux de flux de données*
  - stratégie **« Pipes & Filters »**

```
cat file.txt | grep "error" | sort | uniq -c | head -10
     ↑           ↑            ↑        ↑          ↑
  source      filter       sort     count      limit
```

- lien àvec la **programmation fonctionnelle**
  - composition de *fonctions pures*
  - *immutabilité*

Flux de données: « Filtres » généraux

Filtre

[1,2,3,4,5]  [2,4]

pairs

Transformation/Mapping

[1,2,3,4,5]  [1,4,9,16,25]

carrés

Aggrégation

[1,2,3,4,5]  15

somme

Validation

[1,2,3,4,5] 

{
"O": [1,2,3],
"X": [4,5]
}

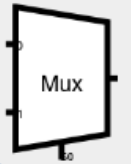
< 4

conversion / parsing

[DT object]  ["2025-11-16"]

-> formater "%Y-%m-%d"
<- parser chaîne

Enrichissement

[{"id":1},...] 

[{"id":1,"age":22,...},...]

données utilisateurs

Tri

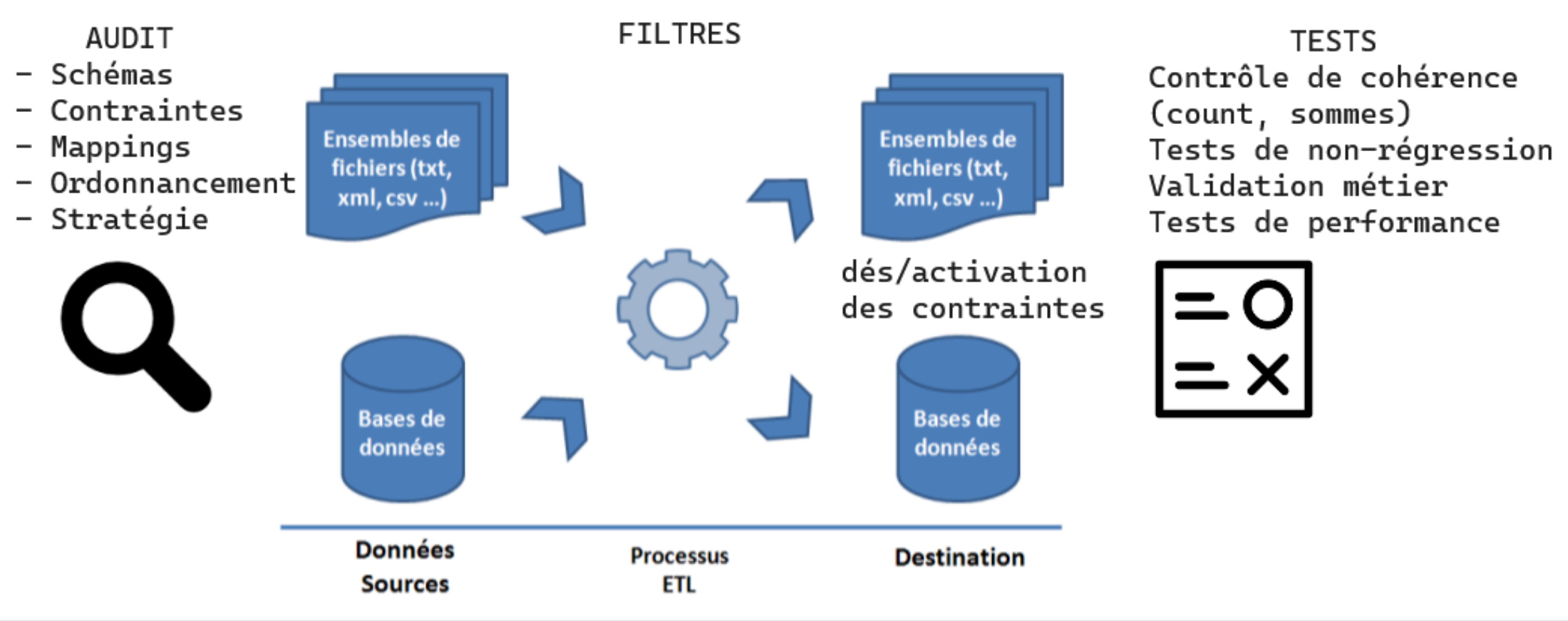
[2,1,5,3,4]  [1,2,3,4,5]

croissant

Flux de données: usages remarquables

- ETL: **E**xtract, **T**ransform, **L**oad
  - *Transformation entre plusieurs sources de données*
- ELT: **E**xtract, **L**oad, **T**ransform
  - *multiples Analyses/Transformations après Chargement*
- CI/CD: **C**ontinuous **I**ntegration / **C**ontinuous **D**elivery
  - *Build* (parse), *Test* (validate), *Deploy* (enrich)
- Exploitation des Logs: *Parse* → *Enrich* → *Index* (sort)

ETL: migration de modèles de données



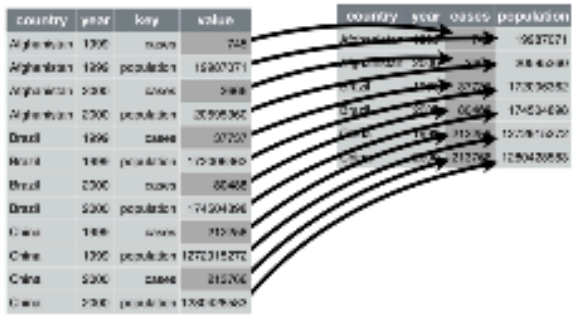
progressive (par modules) et/ou *incrémentale* (par batches)

big bang (tout synchrone d'un seul coup) / *parallèle* (verrous)

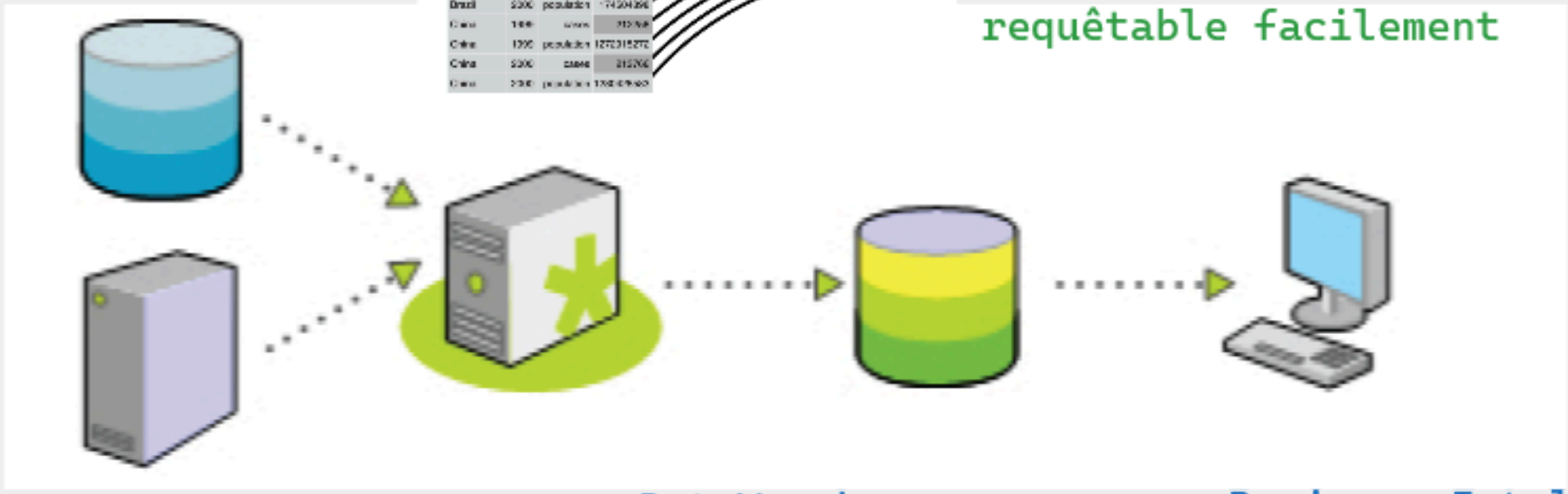
ETL: intégration décisionnelle

Modèle normalisé
grand nb de
tables longues
avec bcp d'indexes

ETL



Modèle dénormalisé
petit nb de
tables larges et jointes
peu d'indexes
avec aggrégats
requêtable facilement



Bases Transactionnelles

DataWarehouse
schema étoile ou flacon

Business Intelligence

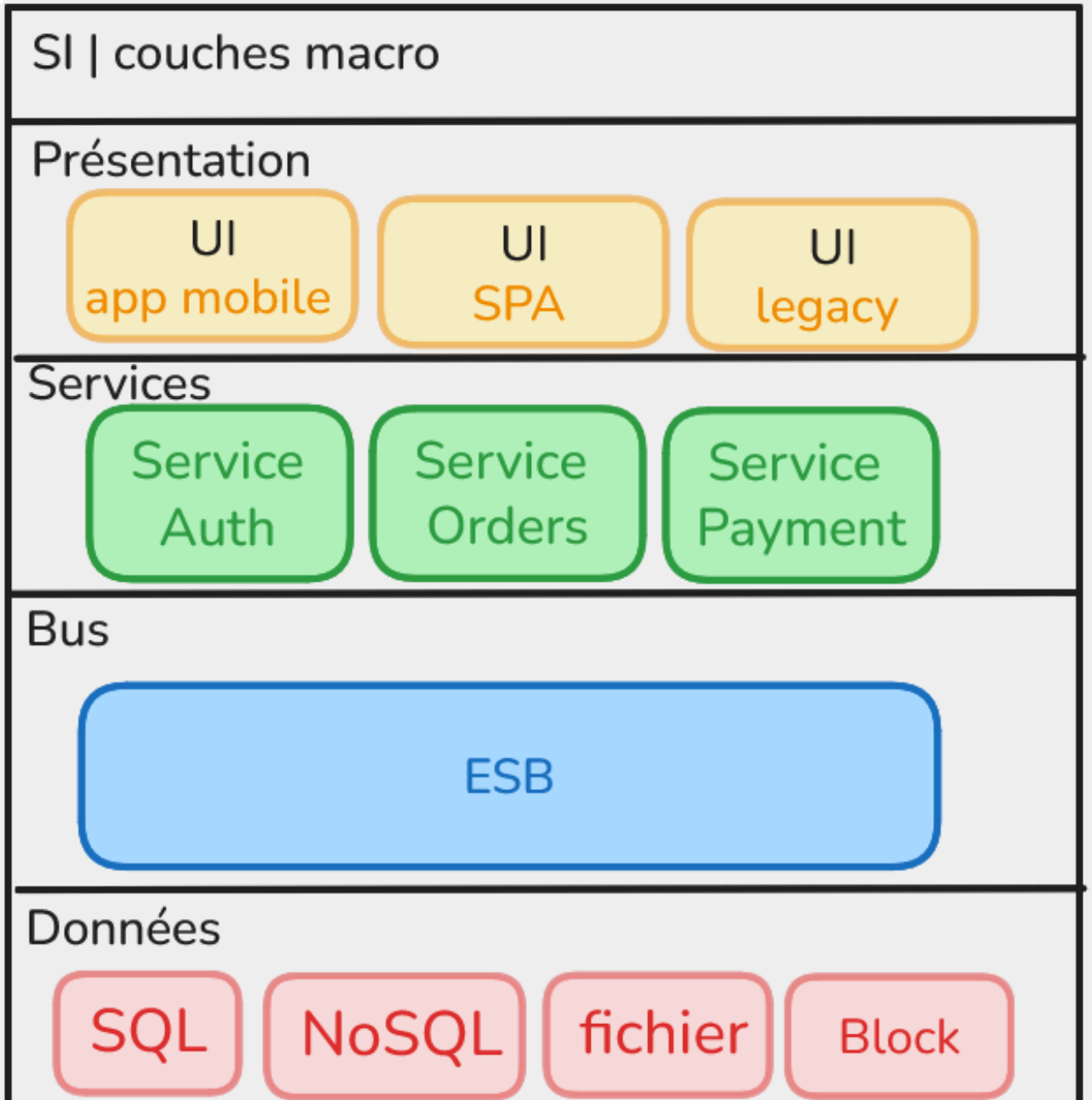
III.4 Architecture distribuée

- *stricto sensu*: système réparti sur plusieurs nœuds physiques
  => client-serveur et n-tiers sont distribués
- architectures distribuées sans couches:
  - **Orientée Services (SOA)**
  - **Microservices**
  - **Orientée Agents / Service Mesh**

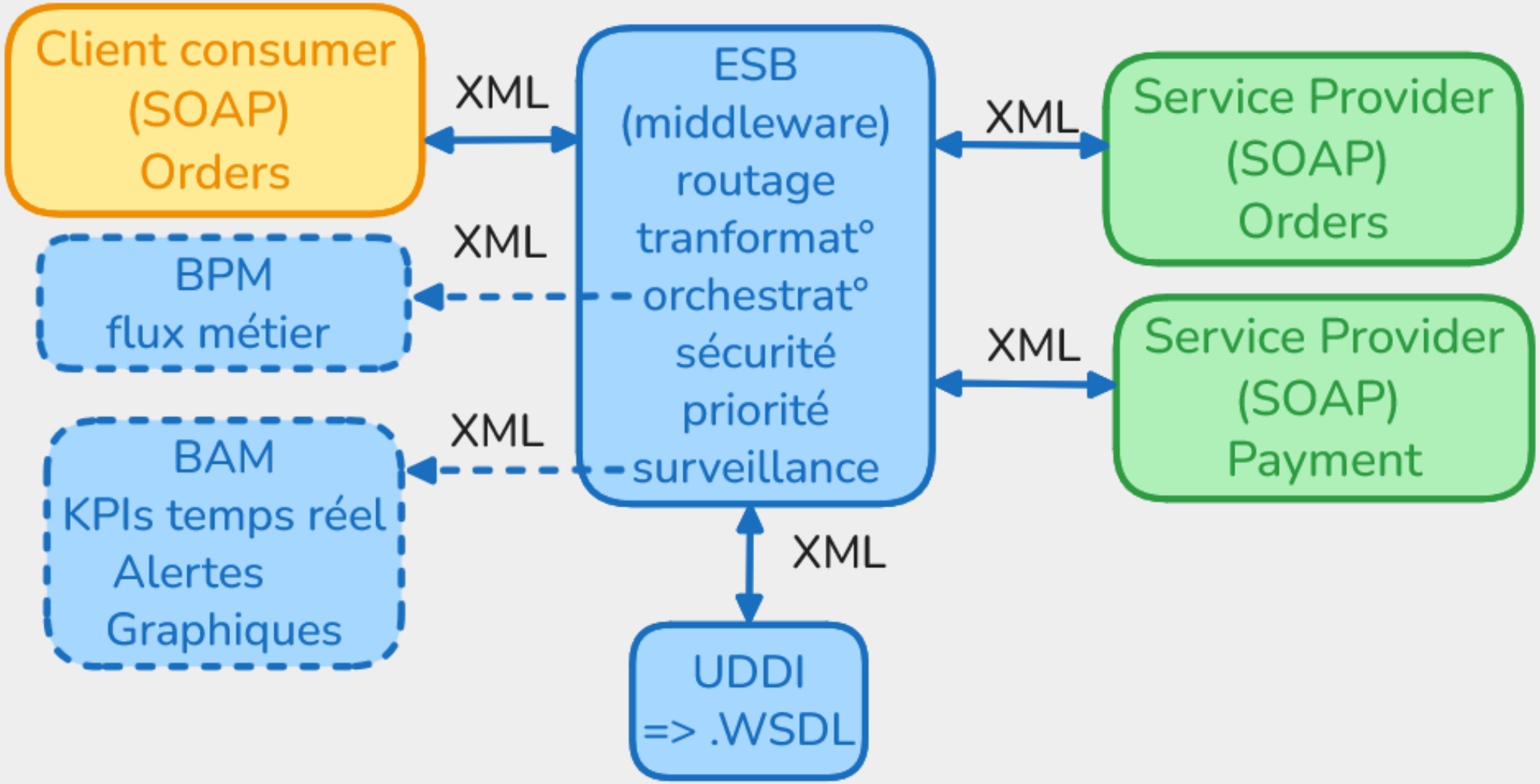
SOA

- architecture installée sur le SI global d'un **réseau d'entreprise**
- dont les sous-systèmes représentent les *services business*
- ces sous-systèmes haut niveau sont des **services autonomes**
  - intégrés et communiquant via un **bus de services d'entreprise (ESB)**.

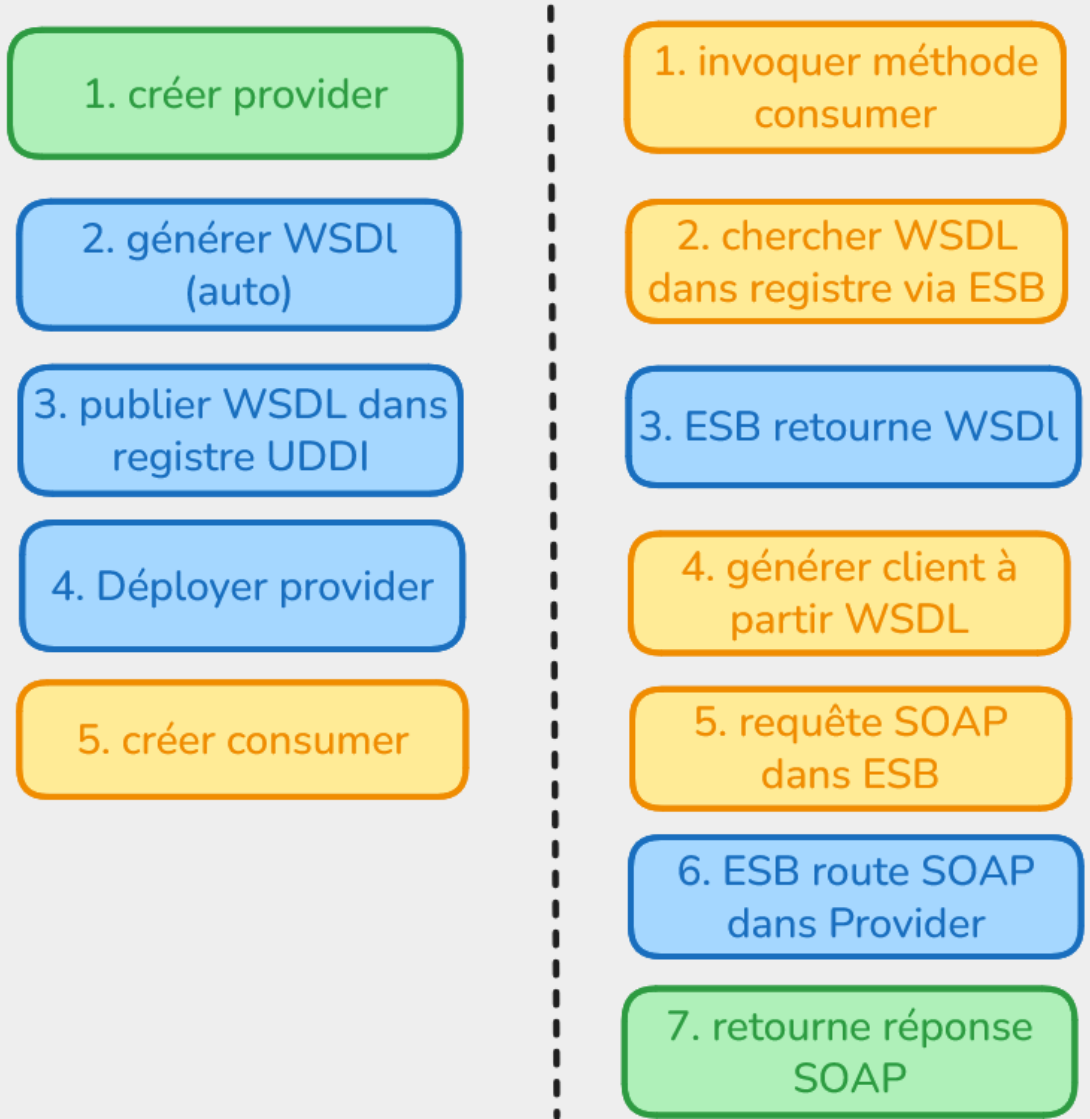
> approche EAI: **E**ntreprise **A**pplication **I**ntegration


SI | couches macro
Présentation
UI
app mobile
UI
SPA
UI
legacy
Services
Service
Auth
Service
Orders
Service
Payment
Bus
ESB
Données
SQL
NoSQL
fichier
Block

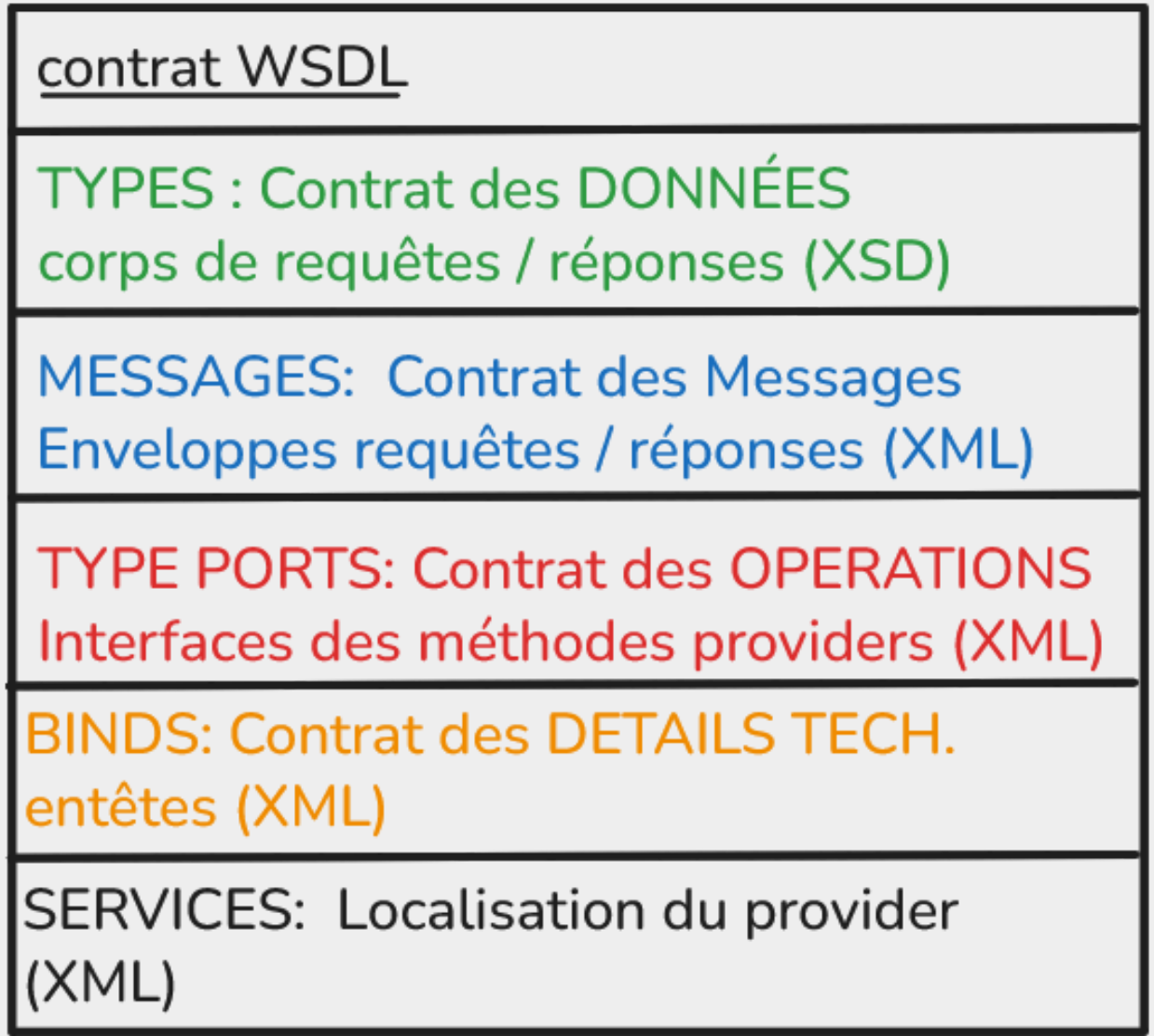
SOA: composants techniques



SOA: flux WSDL / SOAP



SOA: contrats d'interface WSDL



SOA: avantages / inconvénients

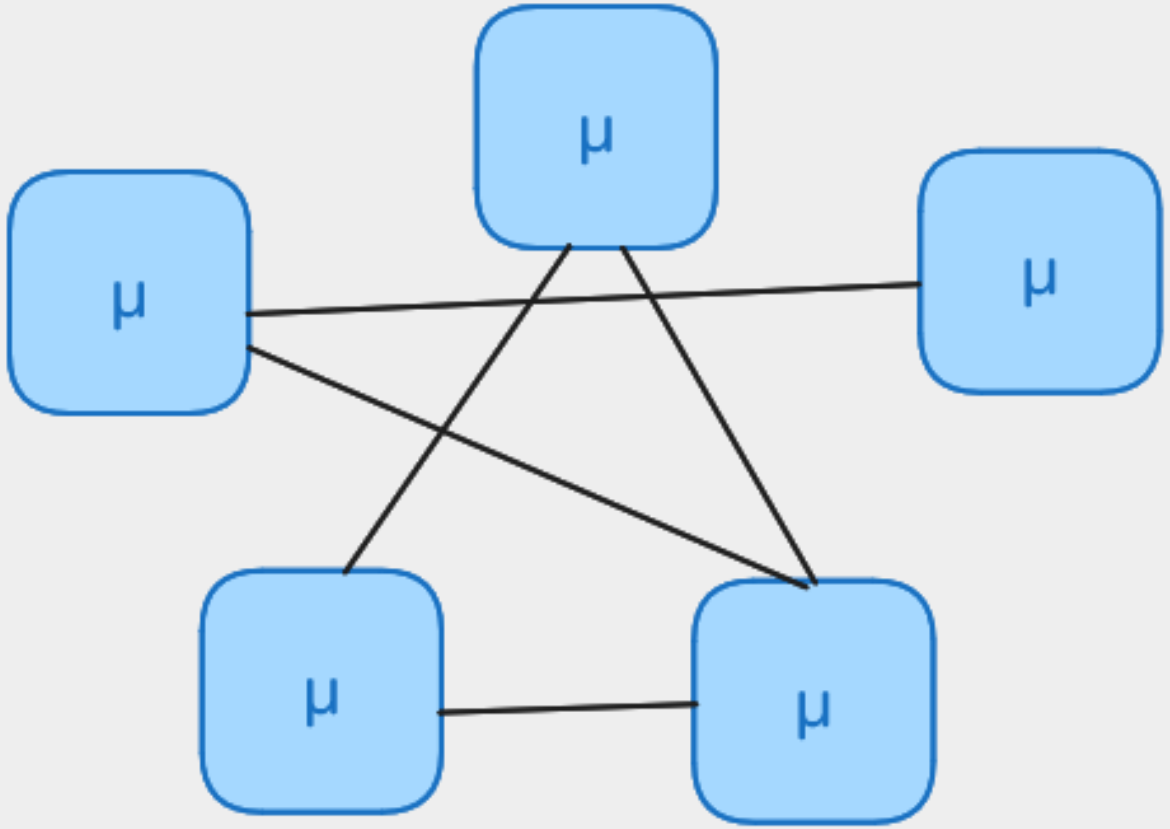
- *+++* : couplage inexistant entre services
- *+++* : réutilisation des services pour nombreux cas d'usage
- **---** : lourdeur des messages SOAP / WSDL
- **---**: pas de client léger => proxy backend pour les consumers
- **---** : complexité de mise en œuvre d'un ESB
- **---** : l'ESB est un gros SPOF

Microservices

stricto sensu:

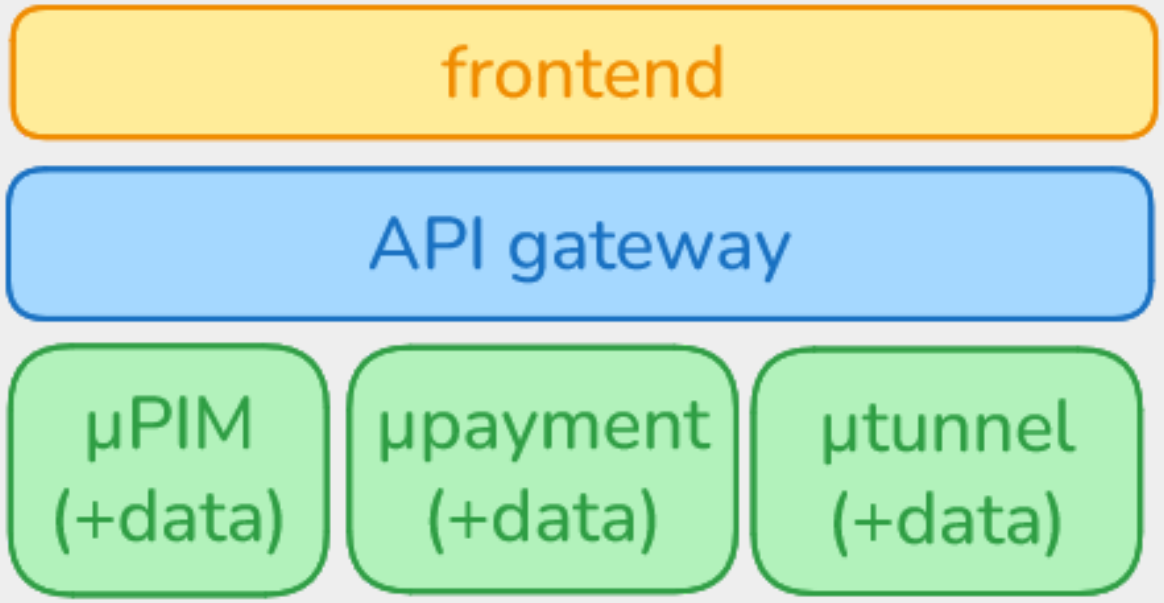
- évolution de la *SOA*
- chaque **µ**service est une **application légère autonome**
  - déployée indépendamment
  - communiquant via des *APIs légères* (REST / GraphQL)
- chaque service implémente un **cas d'usage métier spécifique**
  - et gère ses *propres données*

Microservices: schéma général



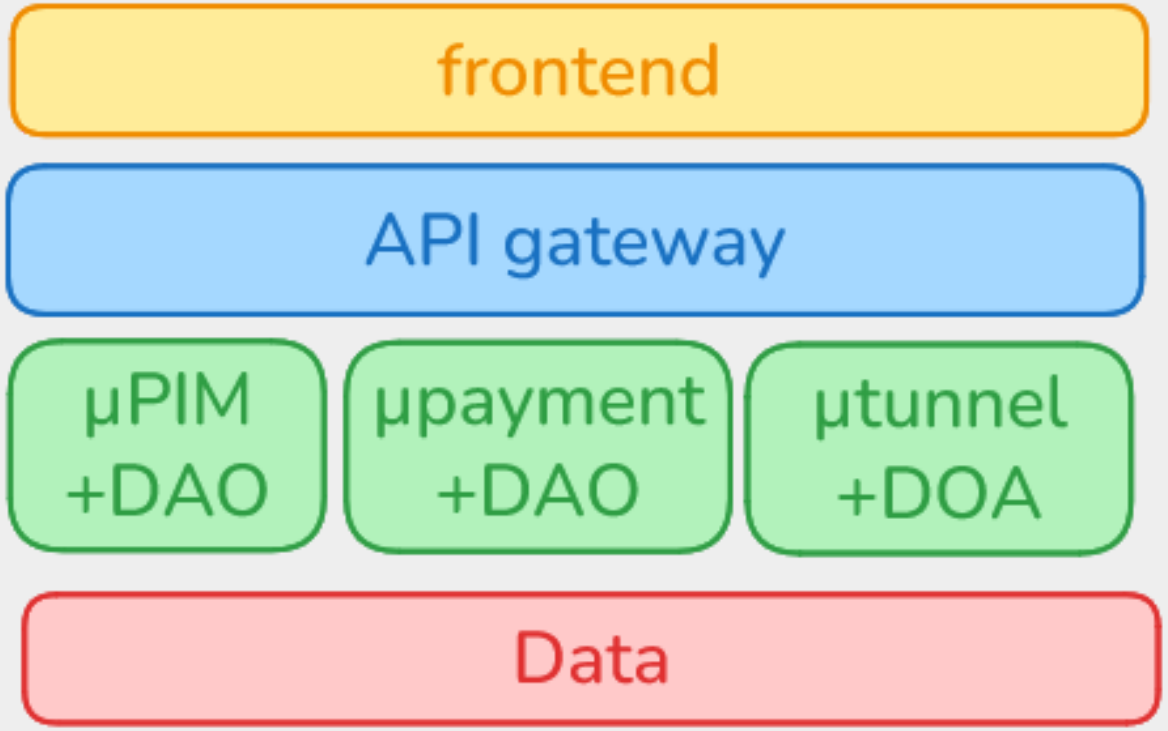
“ **peu pratiquable sans orchestration** ”

Microservices: dans une archi "3-tiers"

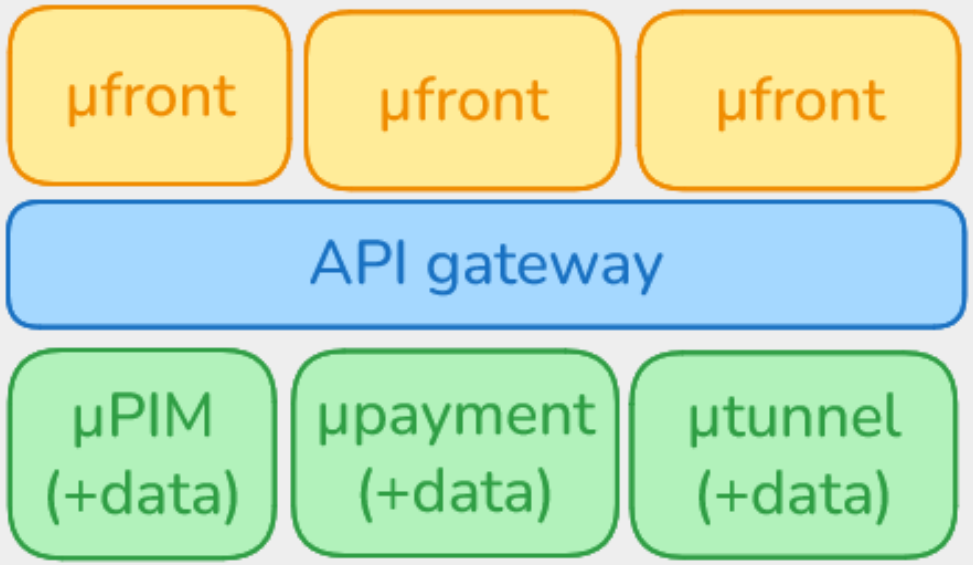


- µservices comme briques ERP
- la **passerelle API** route les requêtes aux bons services

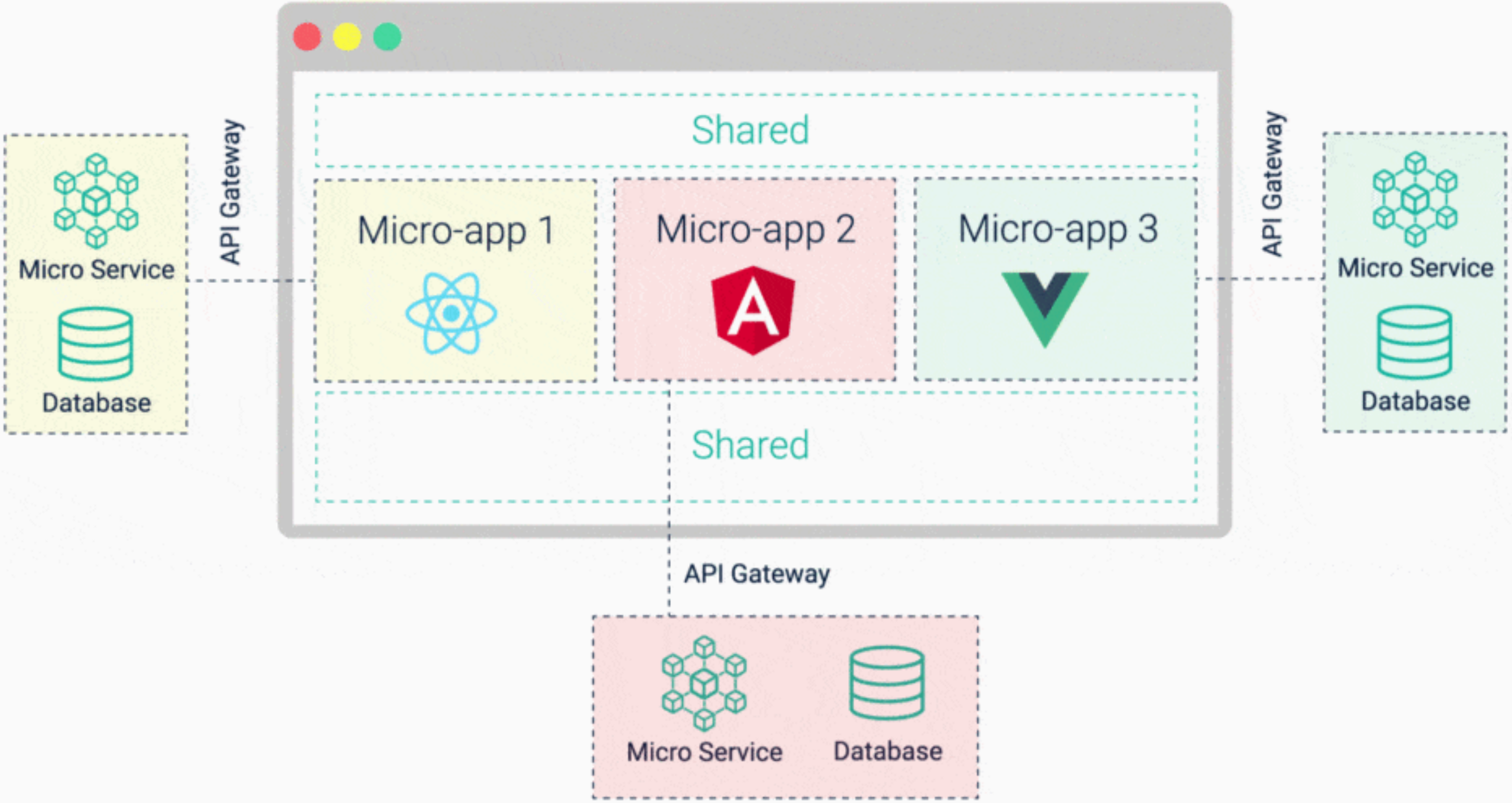
Microservices: variation



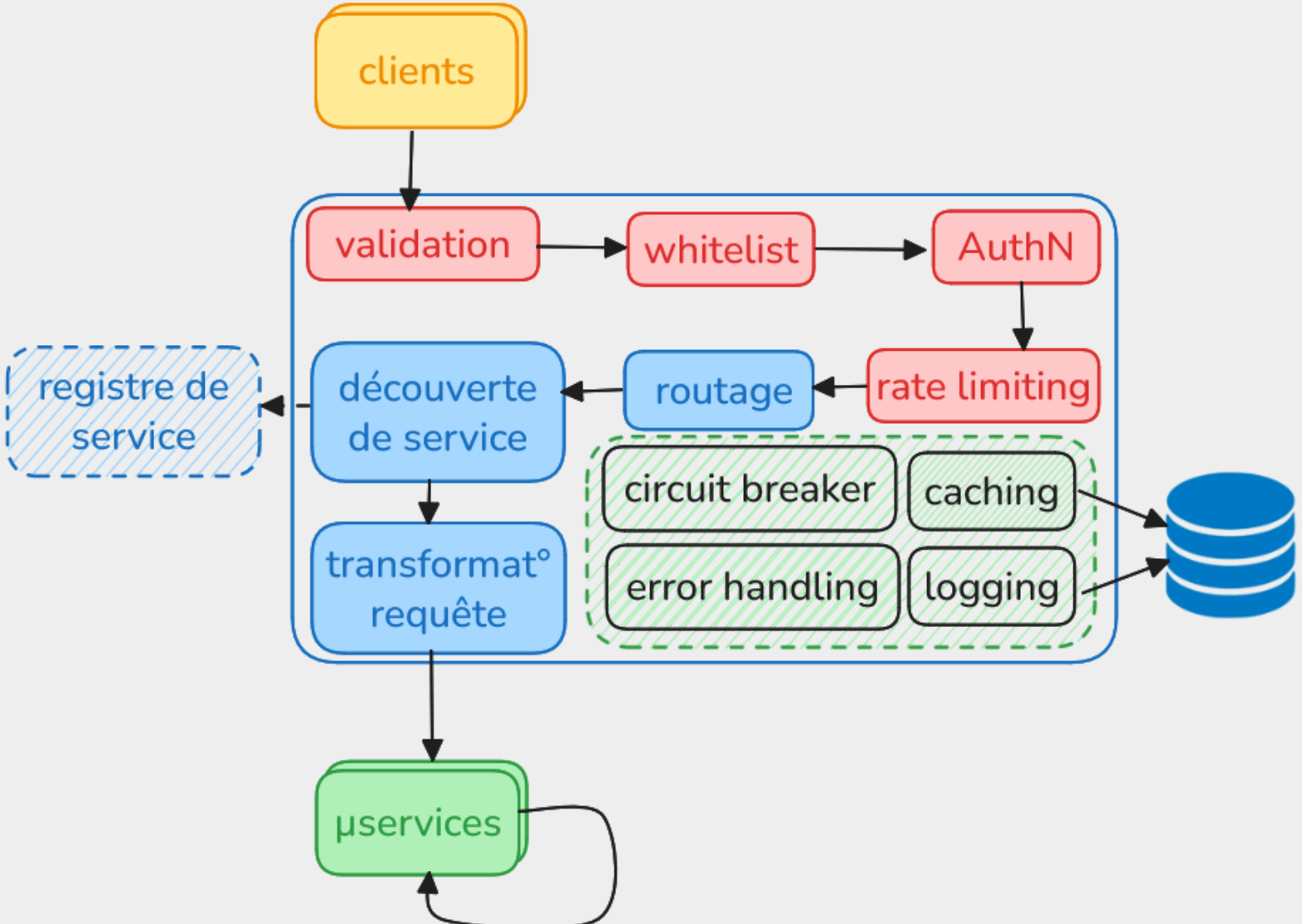
Microservices: micro frontends



- un micro front est un composant UI autonome
- le *gabarit* est souvent une *SPA* (Single Page Application)
- *+++* : déploiement indépendant des équipes UI / UX
- **---** : complexité de gestion des états globaux


Micro Service
Database
API Gateway
Shared
Micro-app 1
Micro-app 2
Micro-app 3
Shared
API Gateway
API Gateway
Micro Service
Database
Micro Service
Database

Microservices: gateway API



Microservices - gateway API: fonctionnalités

- la paserelle est analogue à un **serveur proxy inverse**
  - *routage* des requêtes aux services appropriés
  - *agrégation* des réponses de plusieurs services
- les possibilités de transformation sont *plus réduites*
  - url rewriting
  - conversion de protocoles (ex: REST → gRPC)
- les fonctionnalités création / stockages
  - doivent utiliser des services **stateful** délégués
  - pour *préserver la scalabilité de la passerelle* **stateless**

Microservices: pourquoi les utiliser ?

- Avantages / Inconvénients
  - *+++* : déploiement indépendant des services
  - *+++* : scalabilité horizontale fine des services critiques
  - **---** : complexité de gestion des communications inter-services
- quand le *monolithe devient trop gros / lent / complexe*
- quand les équipes de dev sont *nombreuses et spécialisées*

> la loi de Conway (1967)
> **L'architecture d'un système reflète la structure de communication de l'organisation qui le conçoit**

Archi orientée agents

- chaque **agent** est un **composant logiciel autonome**
  - qui possède une *intelligence* pour
    - percevoir son environnement
    - prendre des décisions
    - agir de façon autonome

> l'agent est un véritable **microservice intelligent**

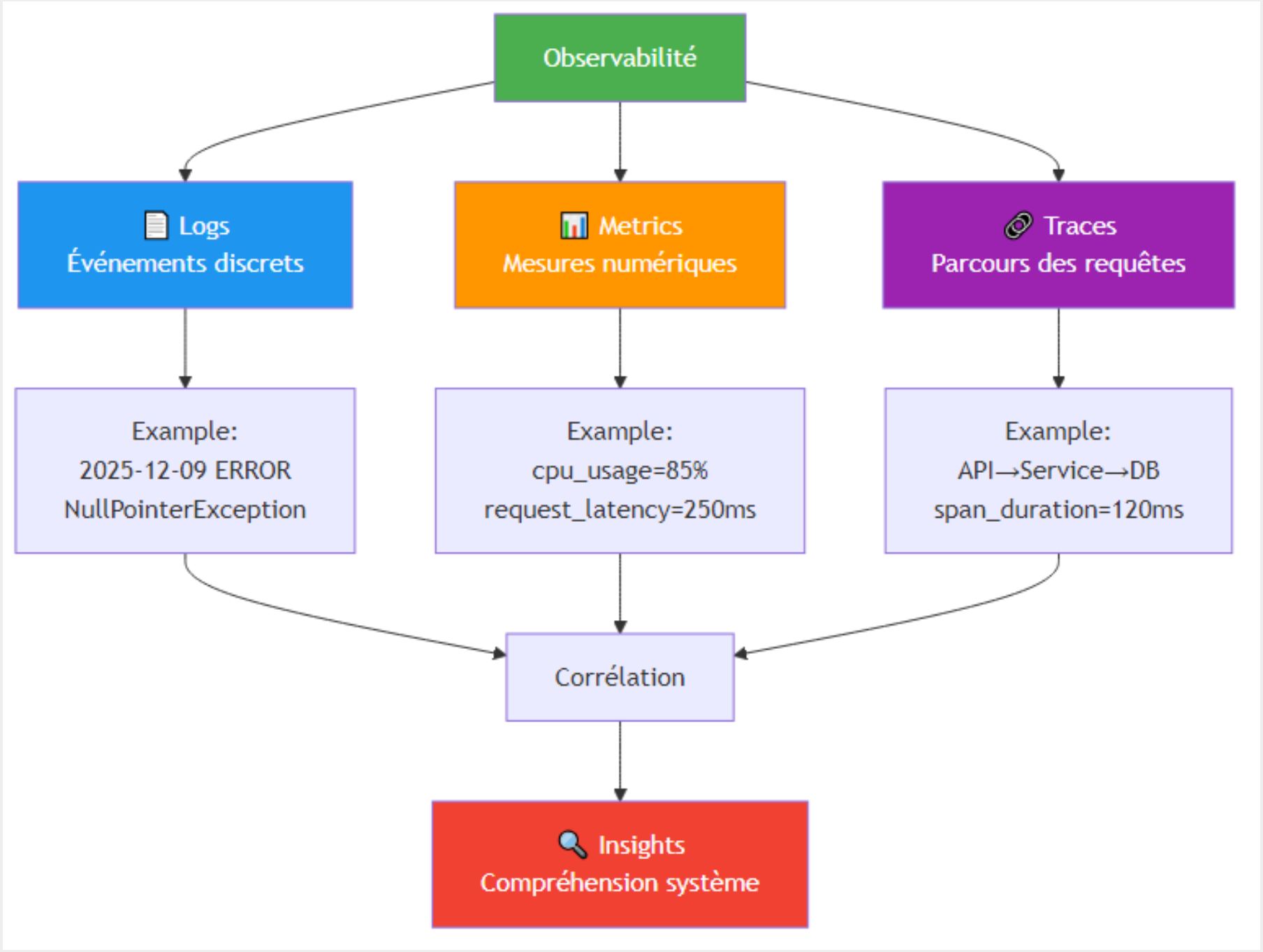
Usage de l'Archi agents: Observabilité

- **monitoring**:

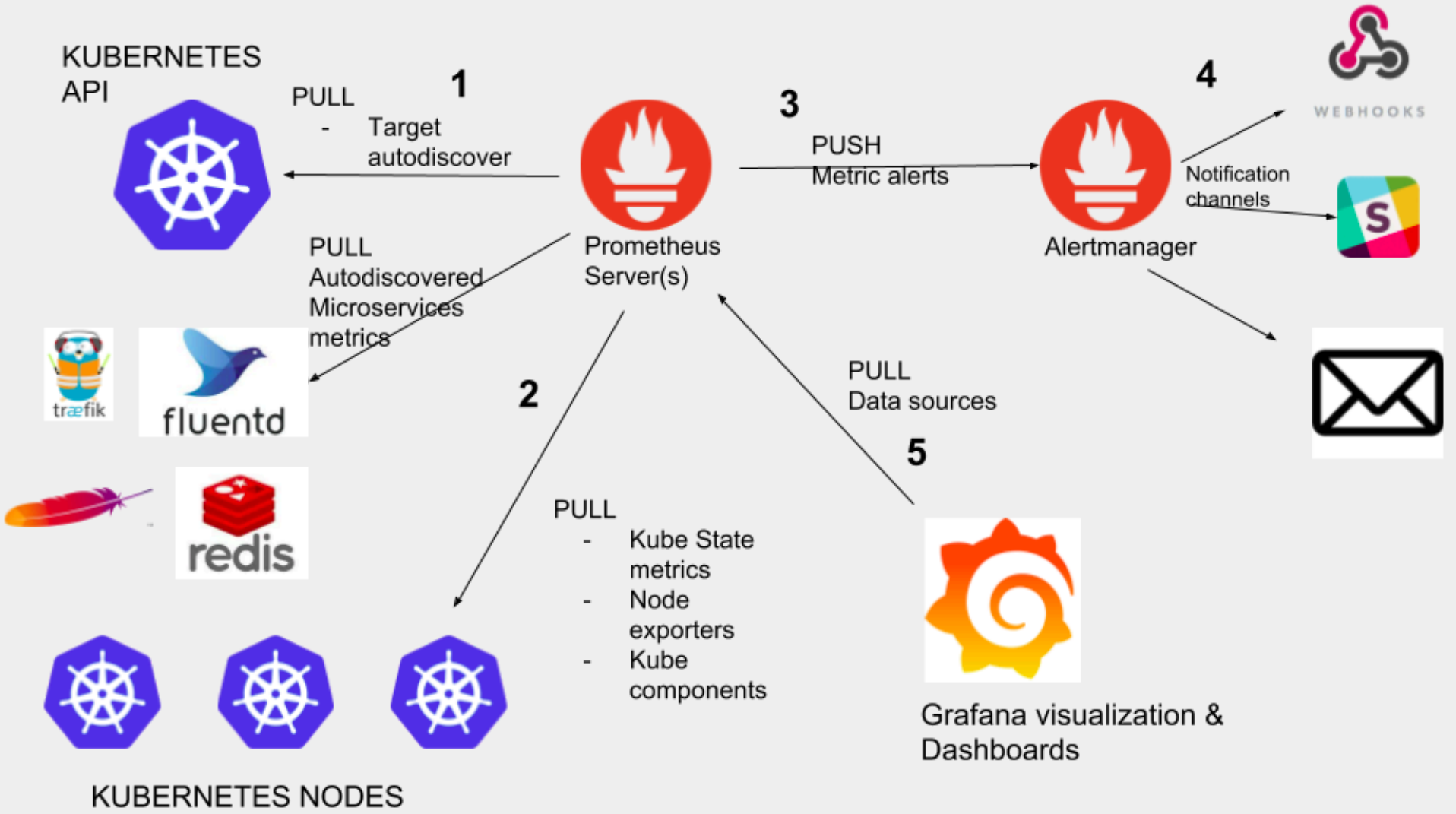
> surveillance continue et automatisée d'un système informatique pour *vérifier qu'il fonctionne correctement* et détecter les anomalies **==> voyants du tableau de bord (t°, vitesse)**

- **Observabilité**:

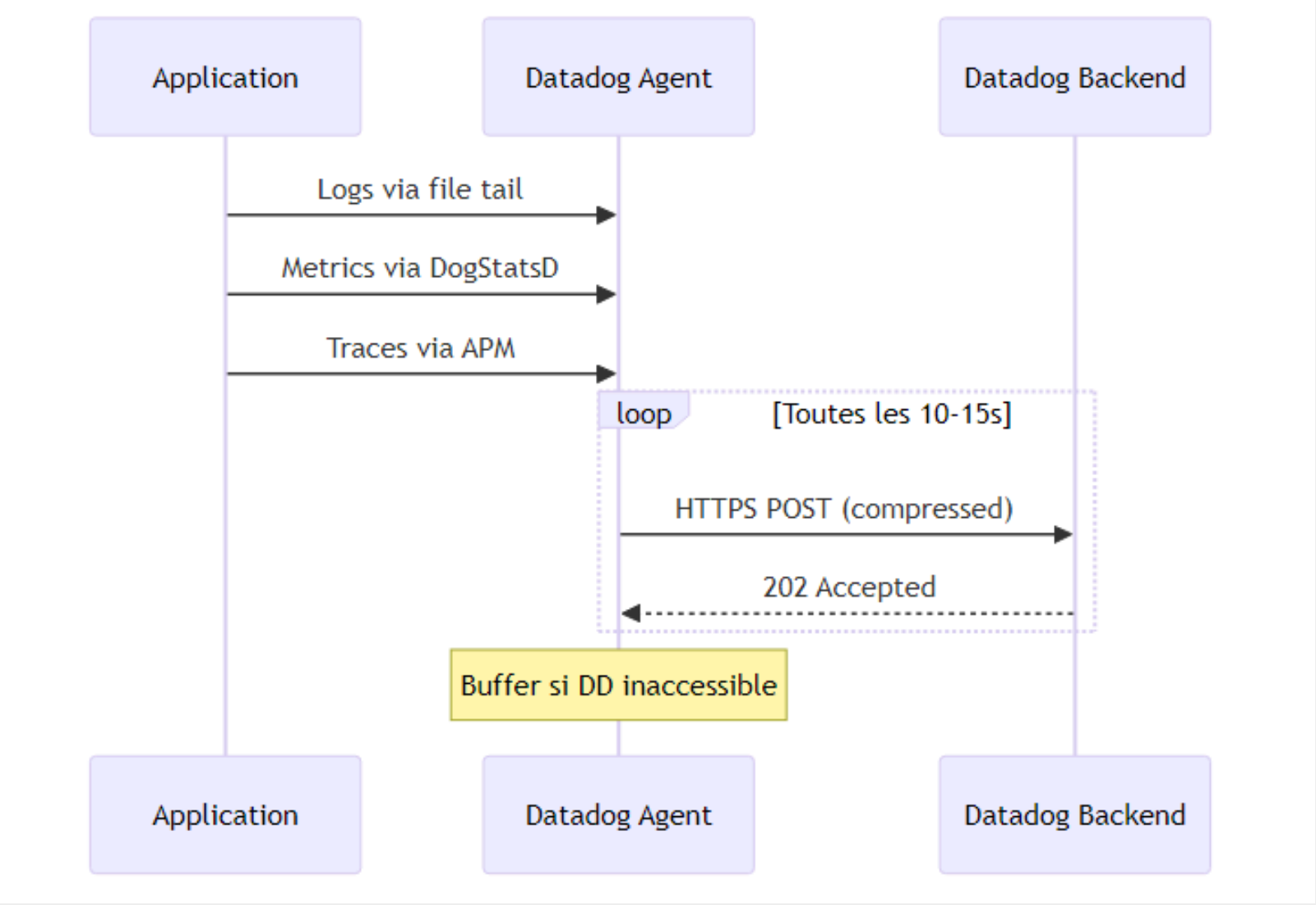
> *capacité à comprendre l'état interne* d'un système en examinant ses sorties externes (logs, métriques, traces). C'est une propriété du système qui permet de diagnostiquer des problèmes sans avoir à le modifier **==> Boîte noire + tous les capteurs**


Observabilité
Logs
Événements discrets
Metrics
Mesures numériques
Traces
Parcours des requêtes
Example:
2025-12-09 ERROR
NullPointerException
Example:
cpu_usage=85%
request_latency=250ms
Example:
API→Service→DB
span_duration=120ms
Corrélation
Insights
Compréhension système

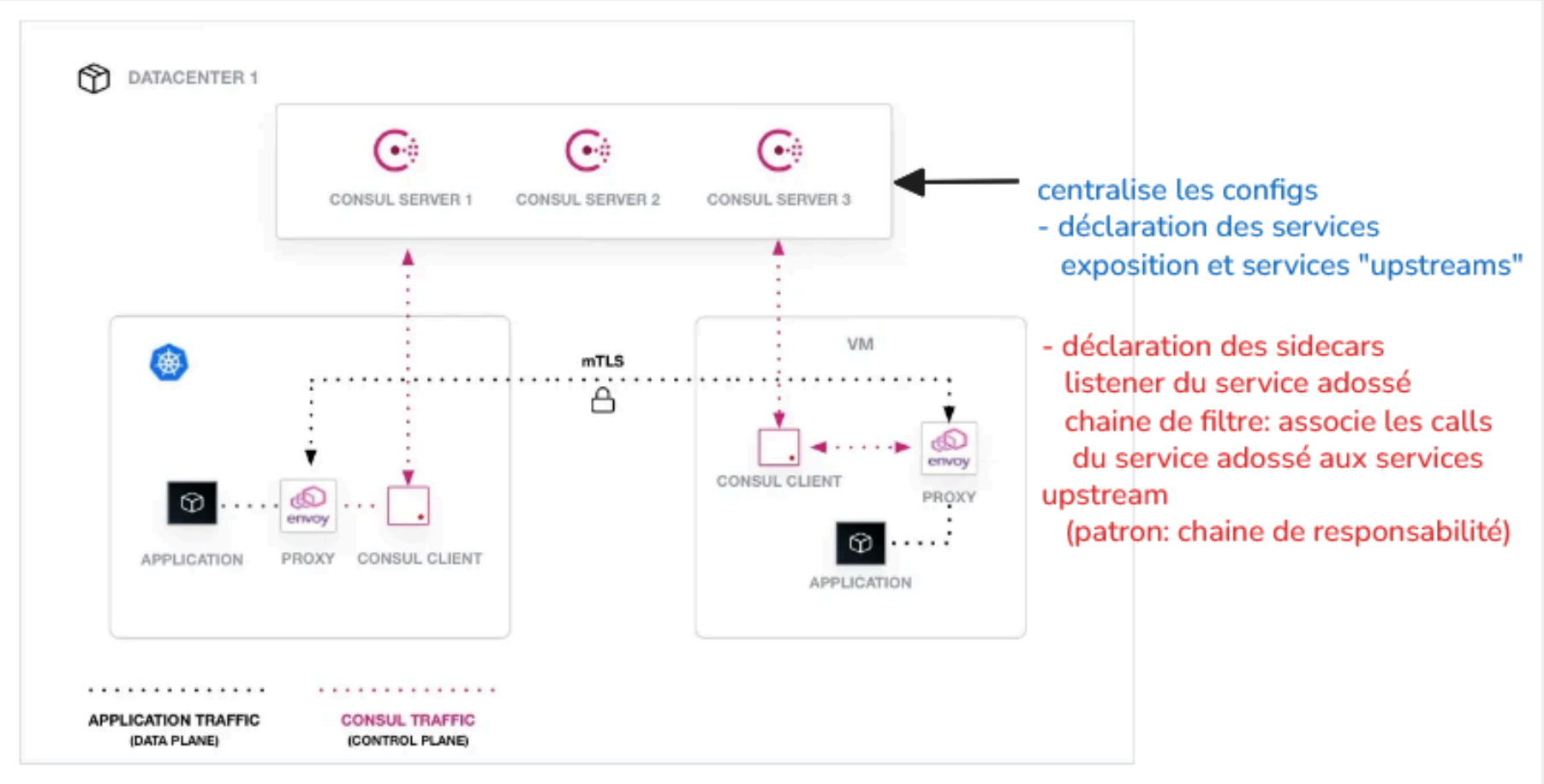
Archi agents: exemple monitoring



Archi agents: exemple observabilité



Usage de l'Archi agents: Service Mesh

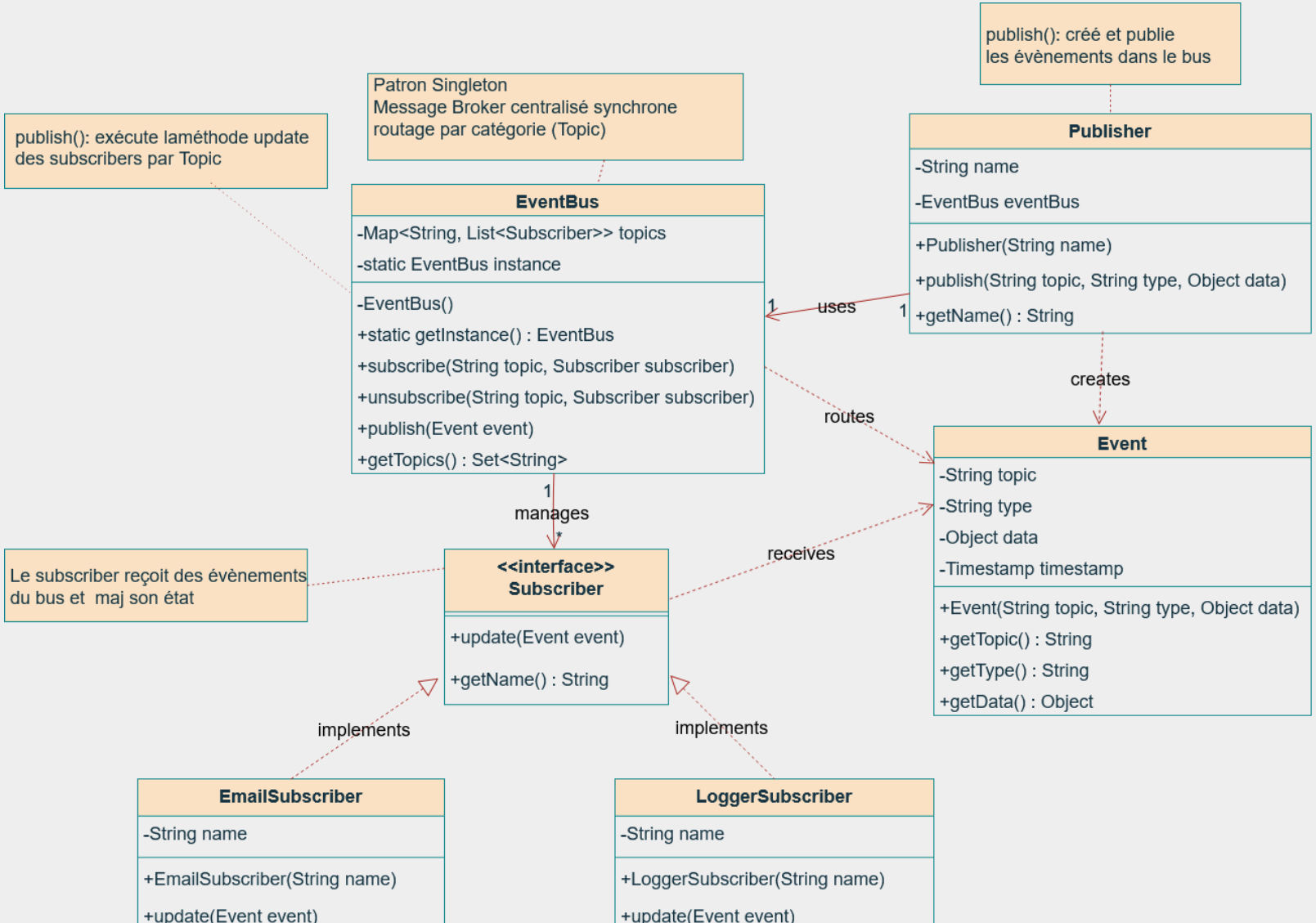


“ **les proxies "side car" forment une passerelle API distribuée !**

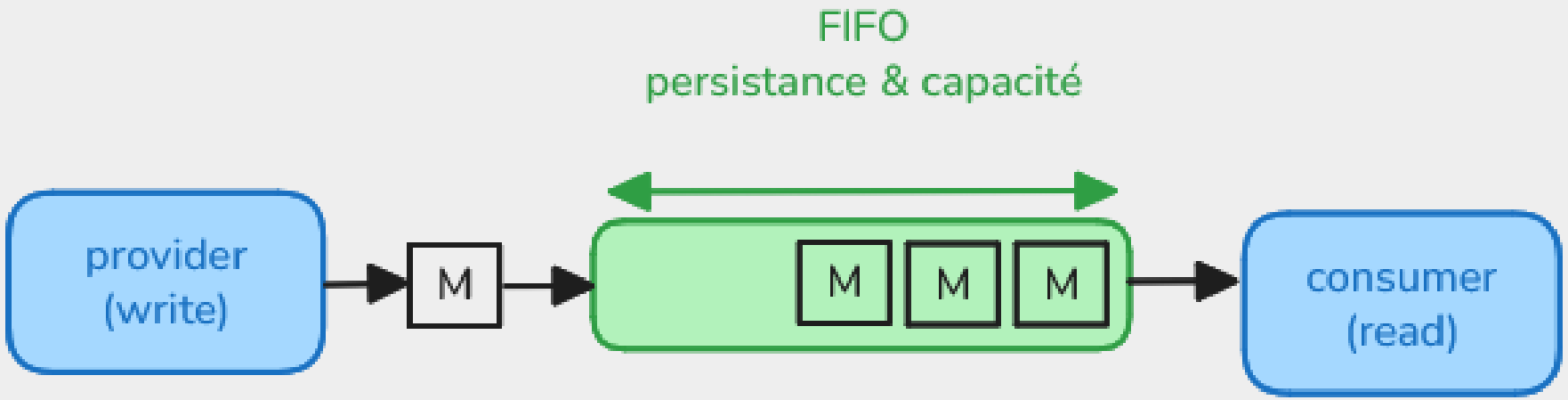
III.5 Architecture orientée évènements

- **Event-Driven Architecture (EDA)**
- les composants communiquent via des **évènements asynchrone**
  - produits par des *producteurs d'évènements*
  - consommés par des *consommateurs d'évènements*
  - les évènements sont des *"faits" => immutables*

EDA: EventBus synchrone



EDA: Message Oriented Middleware (MOM)



- provider , FIFO, et consumer sont des *services*
- les écritures / lectures sont **asynchrones**
  => Pas d'attente de réponse, pas de blocage
- grâce à une *file de messages persistante*

III.6 Architecture centrée sur les données

> Place les données au centre du système, où plusieurs composants/services lisent et écrivent dans un **référentiel de données partagé**.

- ce référentiel représente une Source de Vérité unique (des *données critiques*)

SSOT

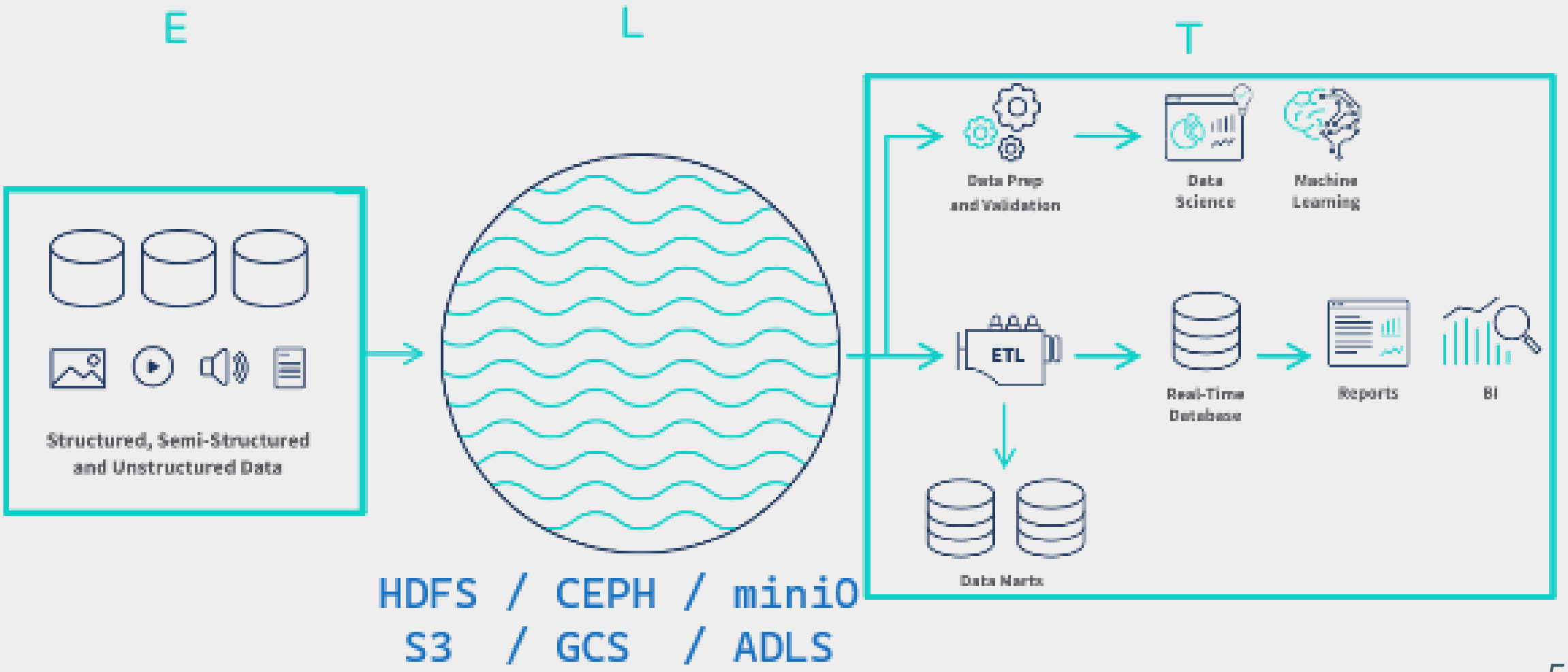
> **Une donnée = Un seul emplacement de référence**

- peut être
  - une base de données / stockage en direct non distribuée
  - un *service* vers le précédent
  - *Data lake* en data engineering
  - moteur de règles dans un *système expert*

Data Lake

- **données brutes** stockées dans leur *format natif avec un ELT*
- ingestion de données provenant de *sources diverses*
- **source unique** pour analyses / machine learning / dénormalisation (data warehouse)
- **Schema-on-read:** convertir les données au moment de la lecture en un format exploitable
- **Append-mostly:** les mises à jour sont surtout immutables
  - modifications rares pour *otpimisations ou RGPD*

Data Lake: architecture



Event Sourcing

- chaque changement d'état d'une entité est enregistré comme un **évènement immuable** dans un *event store*
- l'état courant de l'entité n'est pas une valeur modifiée => mutable
  - mais la **reconstruction** de la séquence des évènements

+++ **traçabilité et auditabilité**
+++ **rejouabilité des évènements**
--- complexité accrue de gestion des évènements

BDD: relationnelle

```
clients                          commandes
                   relation
    ---------------------------------------
    |                                     |
| id | nom   | email           |   | id | client_id | montant | date       |
-------------------------------  -------------------------------------------
| 1  | Alice | alice@ex.com    |   | 1  | 1         | 100.00  | 2024-01-15 |
| 2  | Bob   | bob@ex.com      |   | 2  | 1         | 50.00   | 2024-01-16 |
| 3  | Carol | carol@ex.com    |   | 3  | 2         | 200.00  | 2024-01-17 |
```

- pour les systèmes transactionnels
- ACID: **A**tomicité, **C**ohérence, **I**solation, **D**urabilité
- rigides

BDD: colonnes

```
Relationnel (par ligne) : FORMAT LONG
Row 1: | id:1 | nom:Alice | age:30 | ville:Paris |
Row 2: | id:2 | nom:Bob   | age:25 | ville:Lyon  |
Row 3: | id:3 | nom:Carol | age:35 | ville:Paris |

### TRANSFORMATION: Pivoter la table

Colonnes (par colonne) : FORMAT LARGE
Column id:    [1, 2, 3]
Column nom:   ["Alice", "Bob", "Carol"]
Column age:   [30, 25, 35]
Column ville: ["Paris", "Lyon", "Paris"]
```

- ex: Apache Cassandra, HBase
- **calculs d'agrégats** massifs, **Peer-to-Peer**

BDD: documents

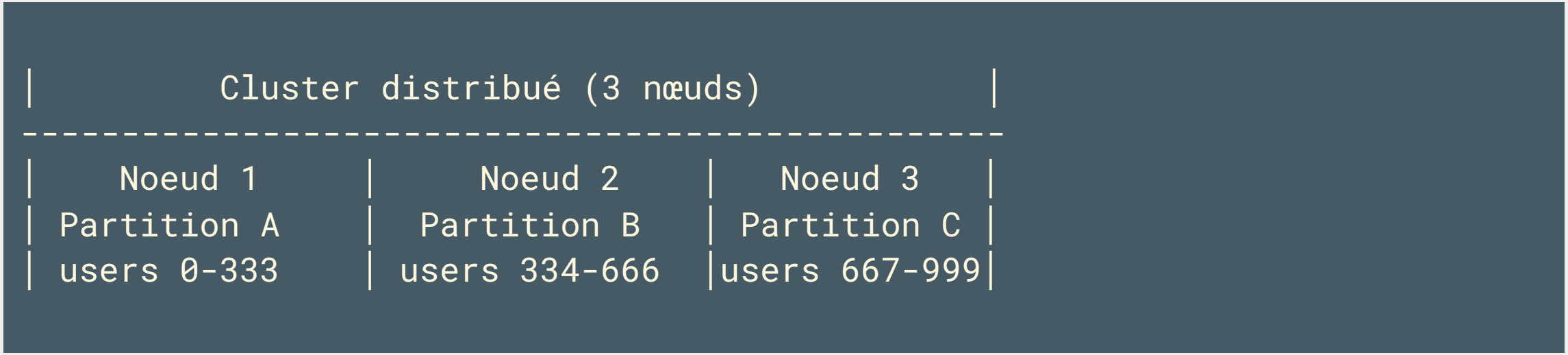
```
// MongoDB : Collection "users": DONNEES NON STRUCTUREES MAIS ! ACID
{
  "_id": "user123",
  "nom": "Alice",
  "email": "alice@example.com",
  "commandes": [
    {
      "id": "order1",
      "montant": 100.00,
      "date": "2024-01-15",
      "items": [
        { "produit": "Livre", "prix": 20.00 },
        { "produit": "Stylo", "prix": 5.00 }
      ]
    },
    {
      "id": "order2",
      "montant": 50.00,
      "date": "2024-01-16"
    }
  ],
  "tags": ["VIP", "Newsletter"]
}
```

BDD: séries temporelles

```
Timestamp           Sensor    Location    Temp    Humidity
____________________________________________________________
2024-01-15 10:00    sensor1   Paris       22.5    60.0
2024-01-15 10:01    sensor1   Paris       22.6    59.8
2024-01-15 10:02    sensor1   Paris       22.4    60.2
2024-01-15 10:00    sensor2   Lyon        18.3    65.0
```

- Ex: Prometheus, InfluxDB
- pour les **données horodatées**
- algorithmes de compression spécifiques

BDD: distribuées (sharding)



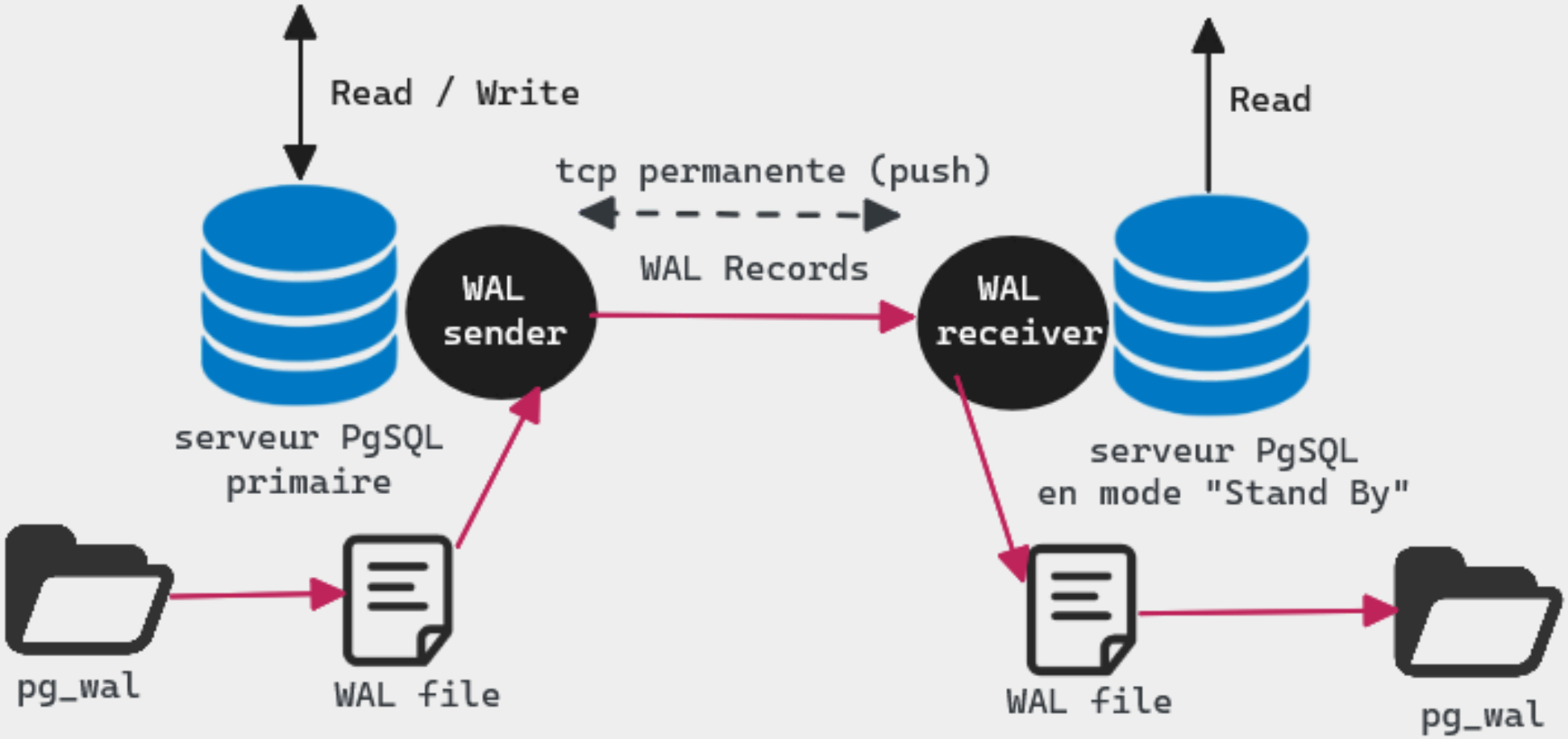
- Ex: Elasticsearch
- pour les très grandes volumétries de données

III.7 Styles hybrides

- la plupart des architectures logicielles **combinent plusieurs styles**
  - ex: flux de données + évènementielle + agents + leader / follower
    => *calcul distribué*

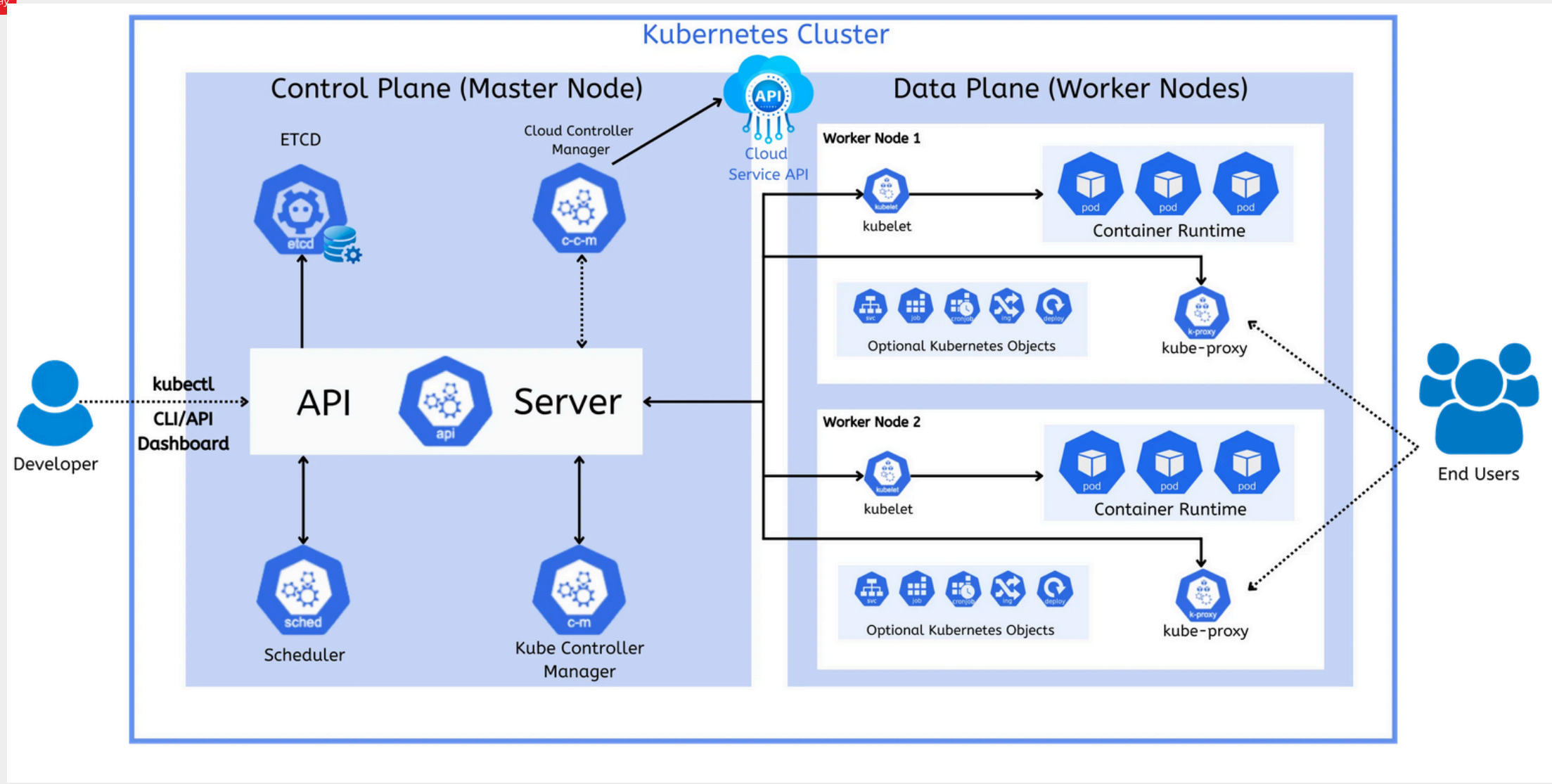
archi leader / follower

- un composant **leader** centralise les décisions
  - et distribue les tâches aux composants **followers**
- ex: **réplication "Statefull"** de *PostGreSQL*
- ex: **Kubernetes**
  - Gestion du *cycle de vie* des applications conteneurisée
  - *contrôle et optimise l'état* des applications souhaité par les développeurs

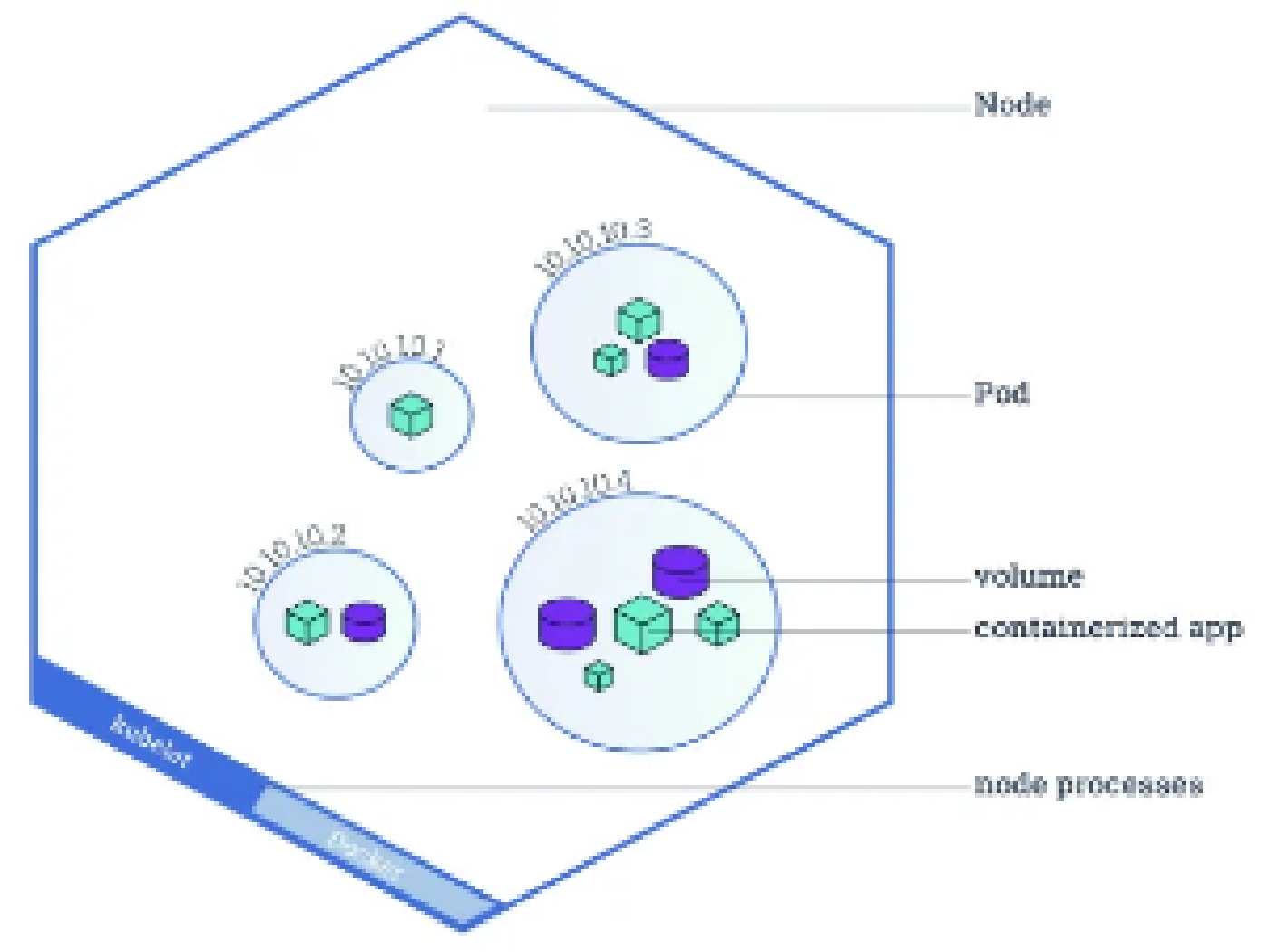


WAL: Write Ahead Logs
- journalisation des écritures avant exécution

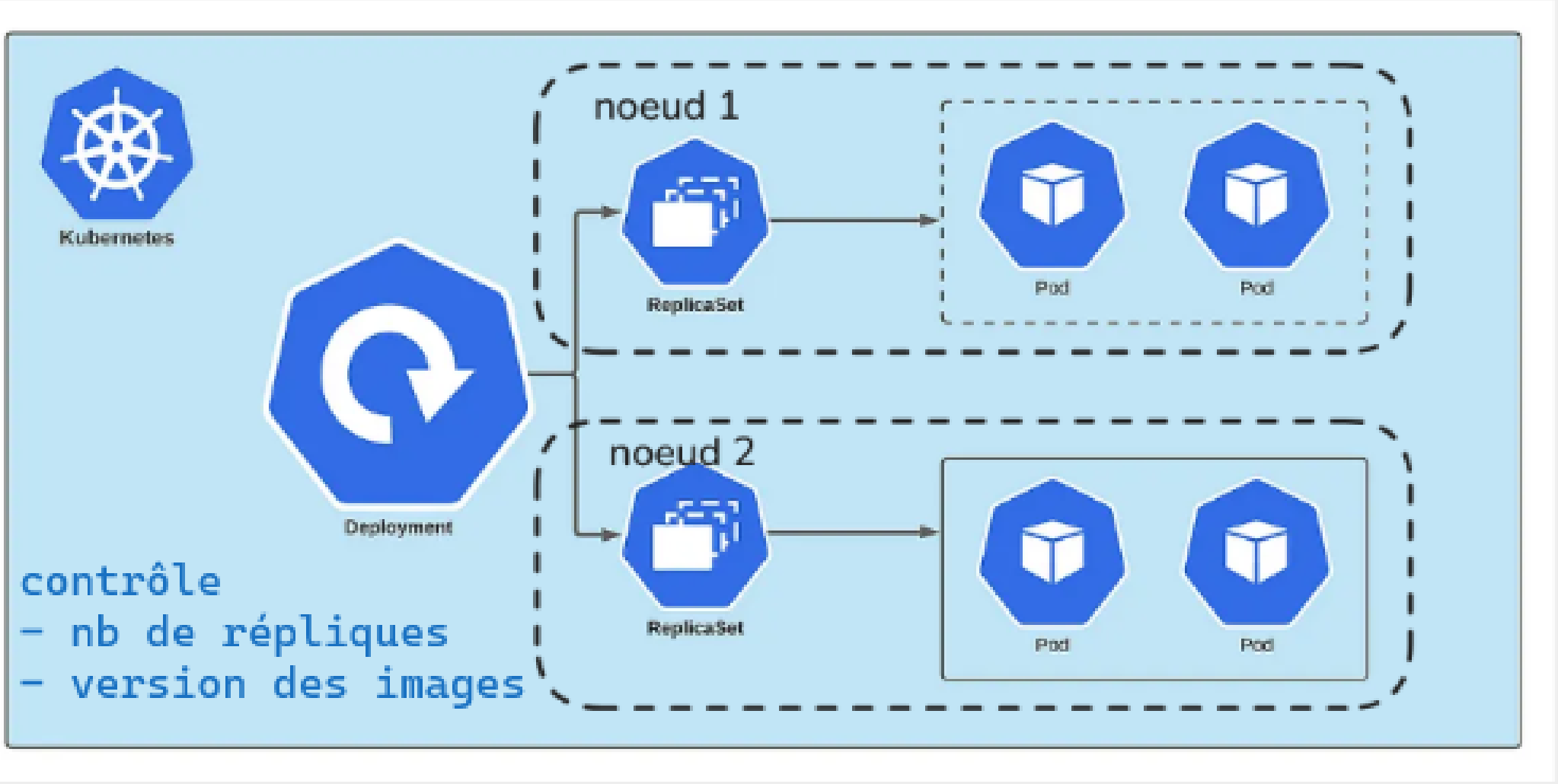
Streaming:les enregistrements sont transférés en continu avant la fin du fichier



pod kubernetes



Déploiement kubernetes



mise en réseau kubernetes