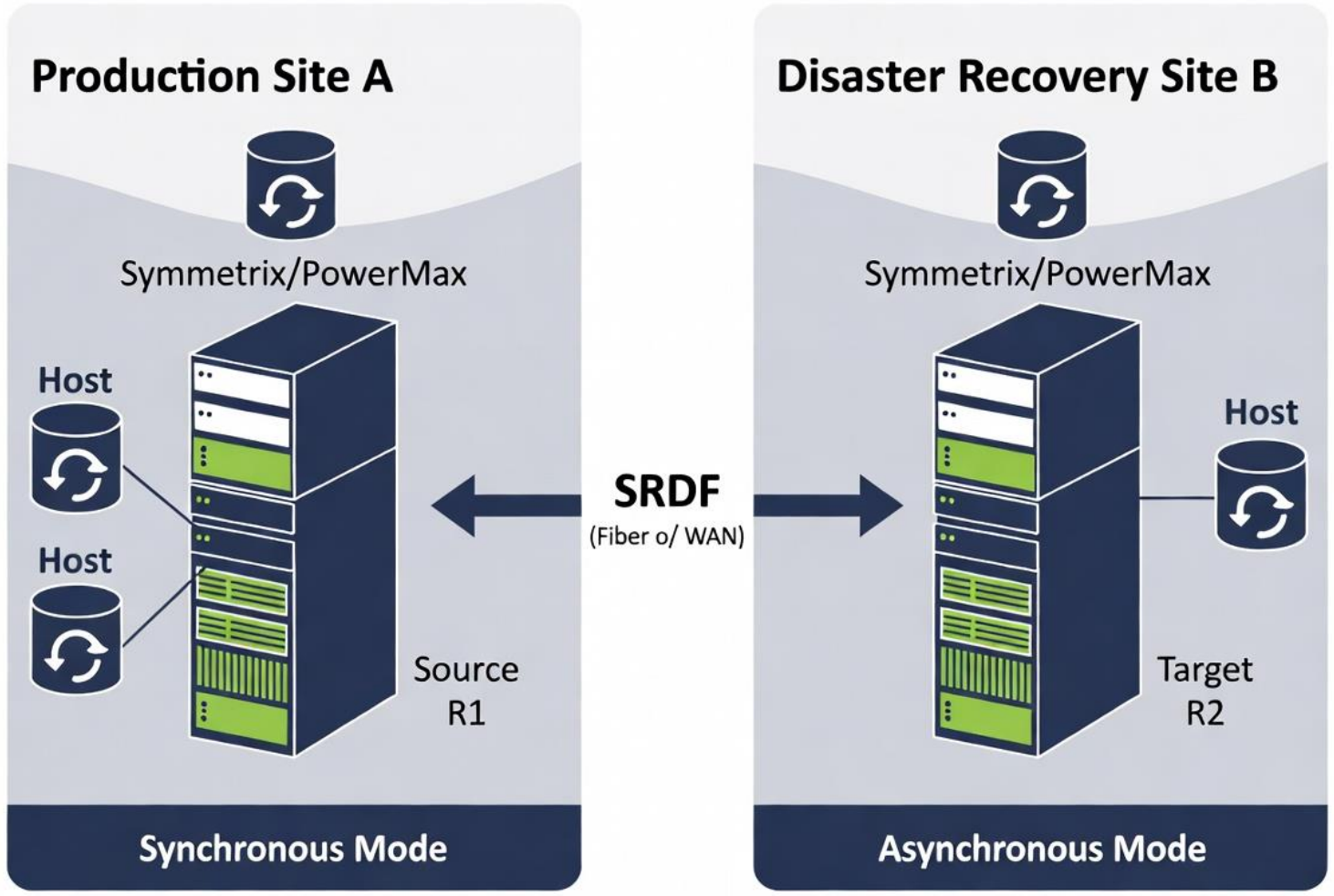
IDEO-LAB

March 2026

v 1.0

SRDF FOR STARTUPS



Présentation du Projet : SRDF for Startups

Le Pitch : La résilience "Enterprise" enfin accessible

Il y a 20 ans, **EMC** révolutionnait le stockage avec **SRDF** (Symmetrix Remote Data Facility), imposant la référence absolue pour la réplication de données et la haute disponibilité. C'était puissant, c'était robuste, mais c'était aussi complexe, propriétaire et surtout hors de prix pour quiconque n'était pas une banque du Fortune 500.

Aujourd'hui, nous changeons la donne. Mon projet, **"SRDF for Startups"**, reprend l'héritage de cette fiabilité légendaire pour l'offrir aux infrastructures modernes (Linux, bases de données distribuées) sous une forme **scalable, abordable et simple à déployer**.

SRDF : une révolution pour votre infrastructure

Le projet ne se contente pas de copier le passé ; il l'augmente pour les besoins de 2026 :

- **Intelligence SRDF Native :** Gestion fine de la cohérence (Consistency Groups) et déduplication intelligente par clé logique (ne transférez que la dernière mise à jour utile).
- **Architecture "Wagon" & Delta-Only :** Optimisation drastique de la bande passante grâce à un système de capture continue et d'expédition par batchs compressés et chiffrés.
- **Agnotisme Stockage :** Là où l'original imposait des baies Symmetrix, notre solution tourne sur n'importe quel serveur **Linux**, libérant les startups du *vendor lock-in*.
- **Scalabilité Horizontale :** Conçu pour encaisser les charges des startups en hyper-croissance, avec une gestion transparente des files d'attente (Inbound/Outbound).

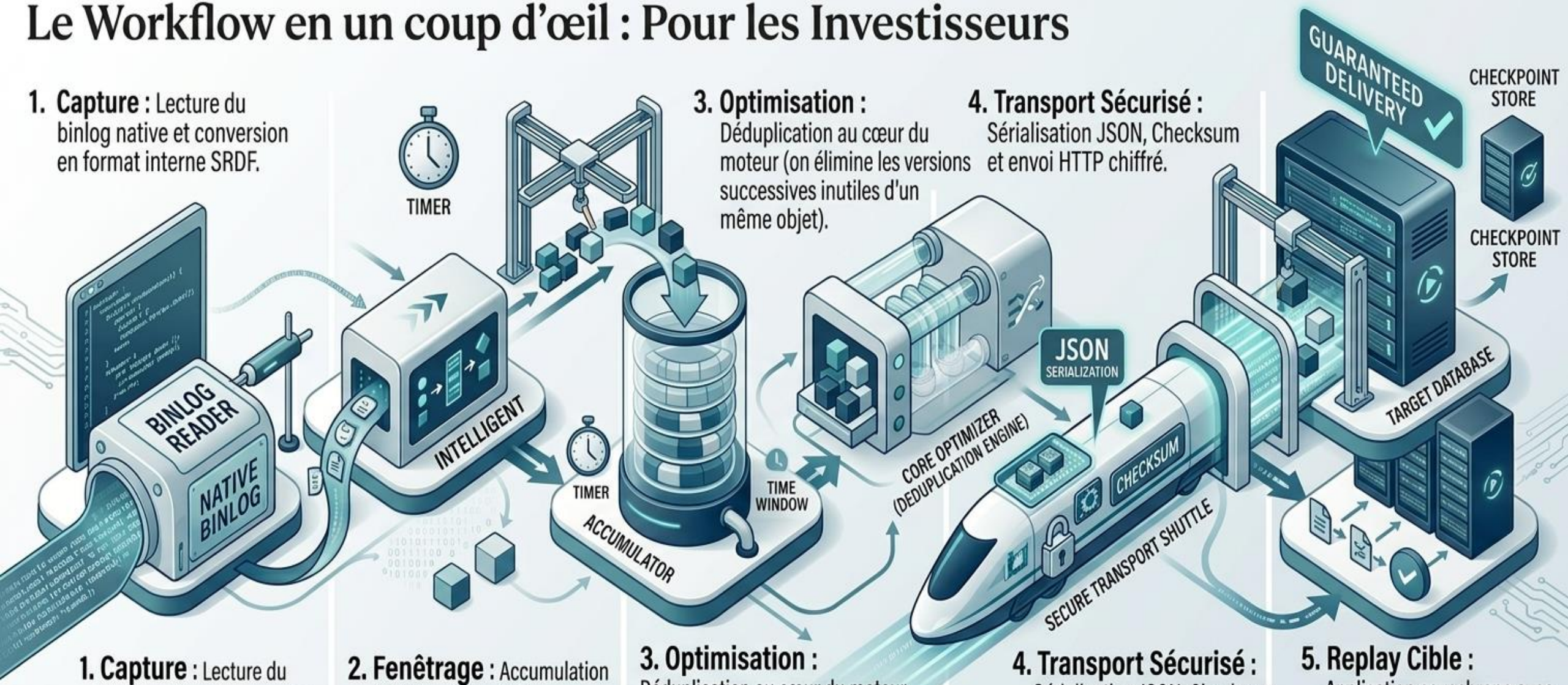
Le Workflow en un coup d'œil

1. **Capture :** Lecture du binlog native et conversion en format interne SRDF.
2. **Fenêtrage :** Accumulation intelligente des événements pour éviter le bruit réseau.
3. **Optimisation :** Déduplication au cœur du moteur (on élimine les versions successives inutiles d'un même objet).
4. **Transport Sécurisé :** Sérialisation JSON, Checksum et envoi HTTP chiffré.
5. **Replay Cible :** Application asynchrone avec gestion de checkpoint pour une garantie de livraison totale.

📈 Pourquoi investir dans cette approche ?

Problème Actuel (Startups)	Solution "SRDF for Startups"	Impact Business
Coûts Cloud explosifs	Déduplication intelligente à la source	Réduction directe de la facture OPEX
Complexité Infra	Solution "Software-Only" sur Linux	Time-to-market accéléré
Risque de corruption	Groupes de cohérence transactionnels	Résilience de niveau "Grand Compte"

Le Workflow en un coup d'œil : Pour les Investisseurs



1. Capture : Lecture du binlog native et conversion en format interne SRDF.

2. Fenêtrage : Accumulation intelligente des événements pour éviter le bruit réseau.

3. Optimisation : Déduplication au cœur du moteur (on élimine les versions successives inutiles d'un même objet).

4. Transport Sécurisé : Sérialisation JSON, Checksum et envoi HTTP chiffré.

5. Replay Cible : Application asynchrone avec gestion de checkpoint pour une garantie de livraison totale.

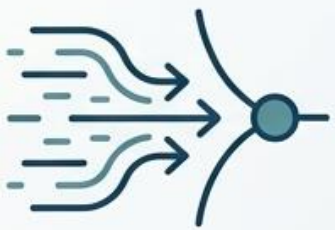
Principes de Fonctionnement : L'Intelligence SRDF au service de la donnée

Le projet repose sur une architecture de **Data Stream Management** qui priorise la cohérence métier et l'économie de ressources. Contrairement aux solutions de réplication classiques qui saturent le réseau, notre approche est **sélective et optimisée**.

Principes de Fonctionnement - SRDF for Startups

1 CAPTURE

Lecture Native du Binlog
(Zero Impact Source)



Capture Asynchrone
(Binlog Reader)

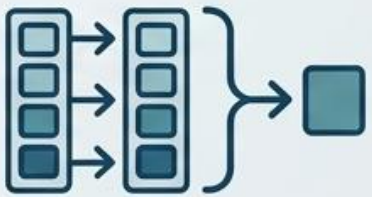
2 FENÊTRAGE (CONSISTENCY GROUP)

Accumulation Intelligente
dans des Wagons

Regroupement
Transactionnel /
Gating temporel

3 DÉDUPLICATION (SRDF CORE OPTIMIZER)

Élimination des
Redondances
(Par Clé Logique)



Analyse et Priorisation
de la Dernière Image Utile

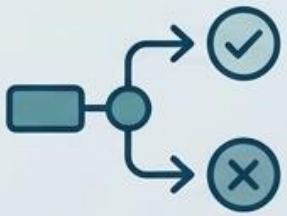
4 TRANSPORT SÉCURISÉ

Compression, Chiffrement,
Transport Optimisé

Batching JSON
+ Checksum
(envoi HTTP/HTTPS)

5 APPLY (GUARRANTEED DELIVERY)

Application Asynchrone
Cible + Validations



Confirmation de l'Écriture
(Validation Cible) /
Mise à Jour du Checkpoint

ZÉRO DATA LOSS
(Garantie Total)

1. Capture Native & Abstraction (Zero Impact)

Nous ne surchargeons pas la base de données source. Le système lit directement les **binlogs (logs transactionnels)** de manière asynchrone.

- **La Valeur :** Performance maximale de l'application de production, sans ralentissement lié à la réplication.

2. Le "Consistency Group" (L'assurance Vie des données)

La donnée n'est pas envoyée de manière atomique et désordonnée. Nous regroupons les événements dans des **groupes de cohérence** (Wagons).

- **La Valeur :** Garantit que même en cas de coupure, la base cible est toujours dans un état cohérent et exploitable (pas de données orphelines).

3. Moteur de Déduplication Logique (Le cœur de l'Intelligence)

C'est ici que nous créons de la marge opérationnelle. Si un client met à jour son profil 10 fois en 2 minutes, nous n'envoyons pas 10 transactions. Le moteur analyse la **clé logique** et ne conserve que la **dernière image utile**.

- **La Valeur :** Réduction drastique (jusqu'à **80%**) de la bande passante et des coûts de stockage cloud par rapport à une réplication standard.

4. Transport Sécurisé & "Batching"

Les données sont packagées, compressées et chiffrées avant d'être expédiées via des tunnels HTTP standards.

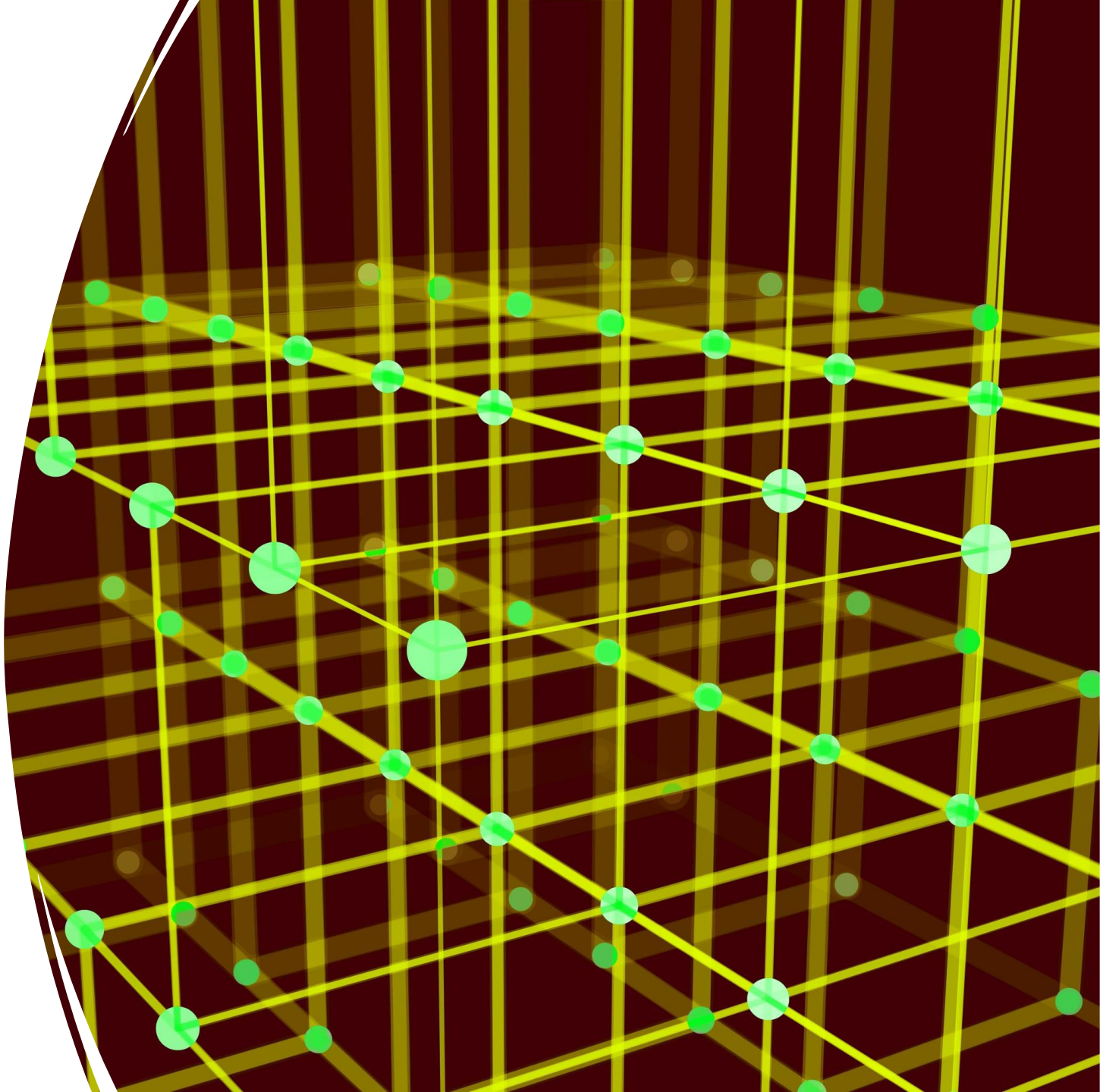
- **La Valeur :** Sécurité de niveau bancaire et facilité de passage à travers les firewalls, sans infrastructures VPN complexes et coûteuses.

5. Mécanisme d'Apply avec Checkpoint

À destination, un worker applique les changements et ne valide le **checkpoint** qu'une fois l'écriture confirmée sur la cible.

- **La Valeur : Garantie de livraison totale.** En cas de crash, le système sait exactement où reprendre. "Zero Data Loss".

PRINCIPES DE BASE



PRINCIPES DE BASE POUR UNE DB - Local

- 1. **Créer un agent (*Local*) qui captera en temps réel toutes opérations** sur la base de données (*et plus largement sur le Moteur SQL MariaDB*) :
  - DML : Insert, Update, delete
  - DDL : Alter, Drop, ...
  - Création de verrous afin de gérer les conflits
- 2. **Ces Opérations seront regroupées** afin d'envoyer un « wagon » vers le serveur Mariab DB cible
  - Optimiser les commandes (DML, DDL, ...)
  - De-dupliquer les commandes (afin d'executer une seule fois les Updates)
  - Enregistrer dans des Tables les Objets Mis à jour (Tables Origine/Envoi)
- 3. **Préparer un envoi de ce « wagon » vers la DB Cible** :
  - Envoi reseau du Wagon de données mises à jour
  - Interrogation de l'agent sur le serveur Cible (dispo, busy, running, en erreur,..)
    - Controle de vérification des objets à mettre à jour (Insert, Update, delete, Drop, Alter, ..)
- 4. **Créer un agent à l'écoute des retours de l'agent remote** :
  - Marquer les mises à jour comme executées sur la DB MariaDB remote
  - Release du Locks

SYNCHRONISATION DB LOCALE À CIBLE

PRINCIPES DE BASE POUR UNE DB - Remote

- 1. **Créer un agent (Remote)** :
  - Phase d'écoute du canal de communication (*element recu de la base Origine*)
  - Vérification de l'intégrité de la demande (DB, Table, Colonne, DML, DDL, ..)
  - Stocker la demande dans une Table SQL réception:
    - avec des ID unique (origine demande)
    - Notion de retry (éventuellement)
  - Mettre dans une queue Local (*Celery par exemple*) la demande de Maj
  - Inscrire la demande à Celery dans une table SQL
  - Mettre à jour le système de Logging
  - Executer la Mise à jour (*sur la Base de données Target mariaDB*)
  - Stocker le résultat dans une table SQL « execution »
  - Renvoyer l'accusé réception de l'execution à l'agent (*Origine, Serveur origine*)

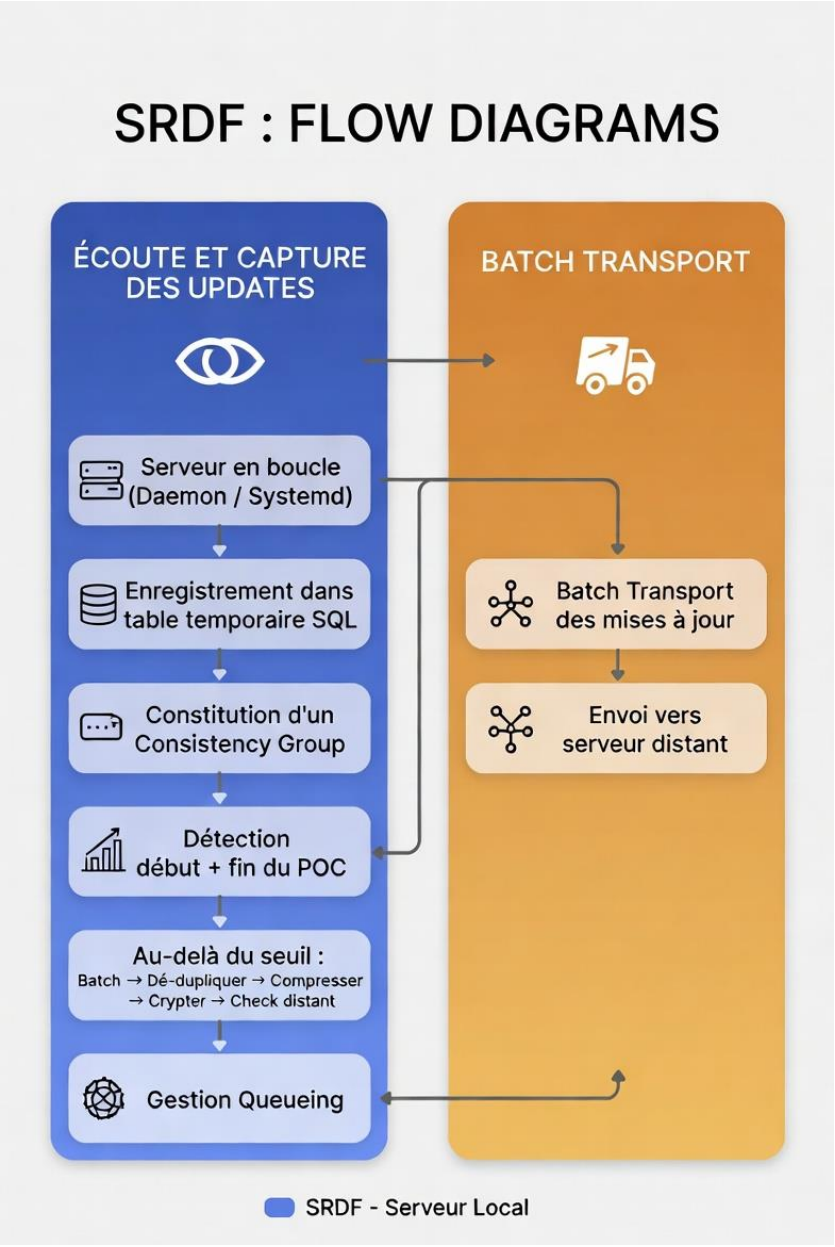
SYNCHRONISATION DB REMOTE

SRDF : FLOW DIAGRAMS

- SERVER LOCAL : ECOUTE ET CAPTURE DES UPDATES
  - SERVER QUI TOURNE EN BOUCLE (DAEMON / SYSTEMD)
  - ENREGISTREMENT DES UPDATES DANS UNE TABLE TEMPORAIRE (SQL)
  - CONSTITUER UN NOUVEAU CONSISTENCY GROUP
  - DETECTER LE DEBUT ET LA FIN DU POC
  - AU DELA D UN SEUIL (Time / Nombre d'Updates / END of Commit)
    - Constituer un Batch
    - De-Dupliquer
    - Compresser
    - Cryptage
    - Check avec le Serveur distant
  - GESTION QUEUING

- SERVER LOCAL : BATCH TRANSPORT

SRDF FLOW DIAGRAMS



PLAN DE CODING DU PROJET EN PYTHON

Recommandation de démarrage réaliste

V1

- agent Python
- control plane Django
- PostgreSQL streaming replication
- MariaDB GTID replication
- rsync/lsyncd fichiers
- failover manuel assisté
- witness simple
- dashboard + audit

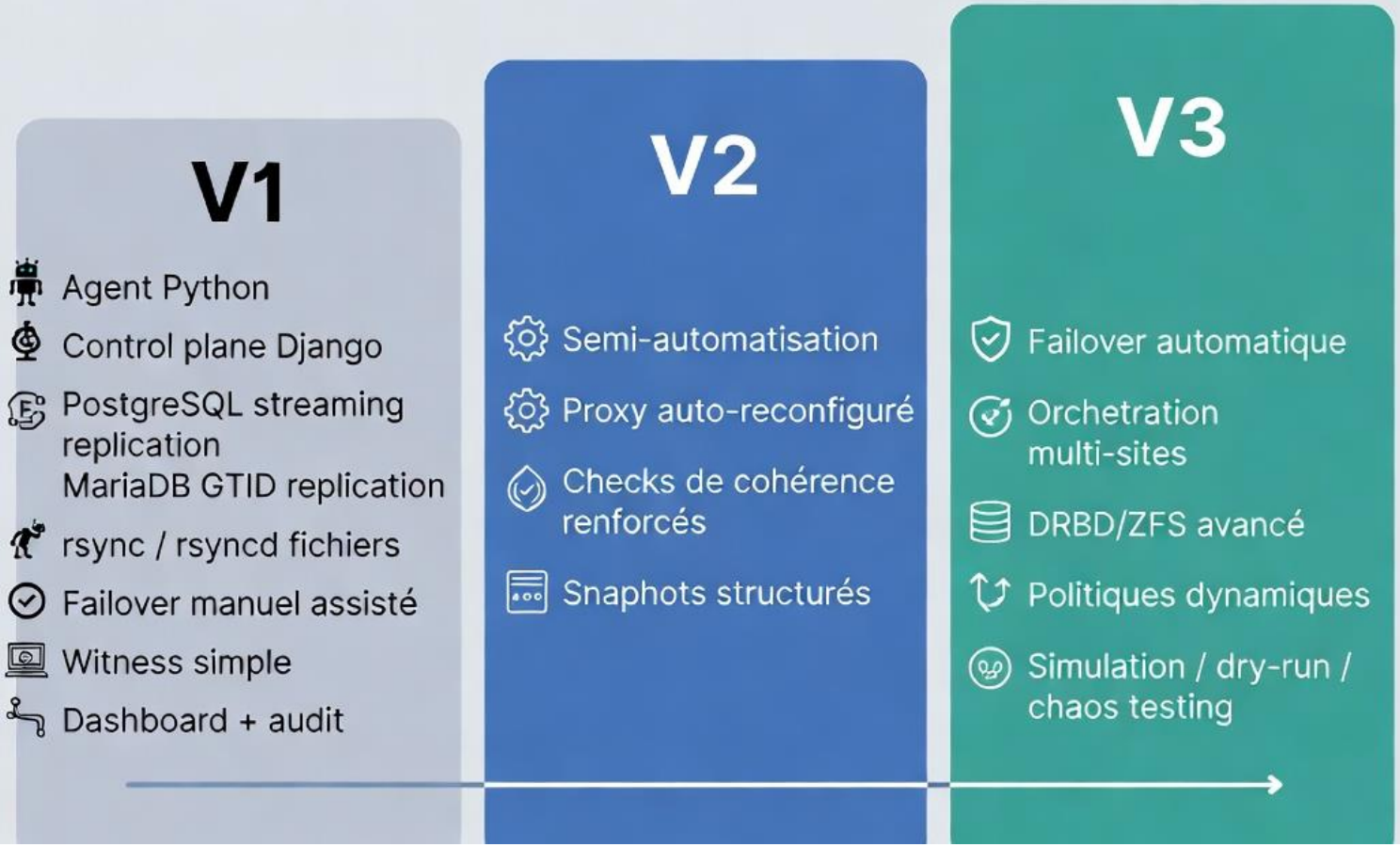
V2

- semi-automatisation
- proxy auto-reconfiguré
- fencing
- checks de cohérence renforcés
- snapshots structurés

V3

- failover automatique
- orchestration multi-sites
- DRBD/ZFS avancé
- politiques dynamiques
- simulation / dry-run / chaos testing

ÉVOLUTION SRDF : V1 → V2 → V3



On part donc sur une **V1 codable immédiatement**, propre, progressive, sans tomber dans le piège du “pseudo-SRDF monolithique ingérable”.

Le bon choix pour la V1 est :

- **agent Python sur chaque serveur**
- **control plane Django central**
- **réplication DB native existante** pilotée et surveillée par notre plateforme
- **réplication fichiers via rsync/lsyncd**
- **failover manuel assisté**
- **audit complet**
- **aucun auto-failover en V1**
- **aucun DRBD en V1**
- **aucune magie dangereuse**

L’idée fondamentale est simple :

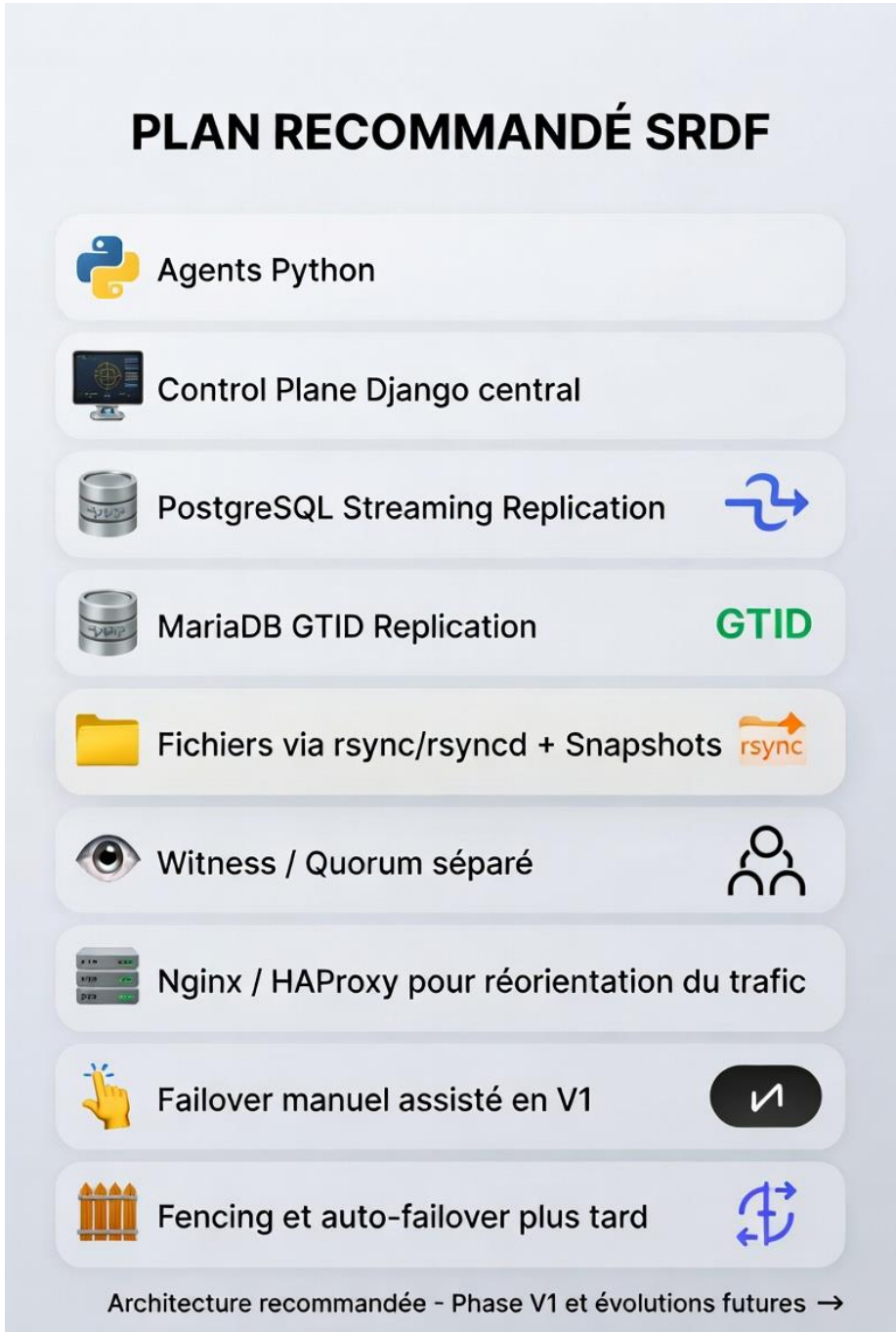
> en V1, **on ne réécrit pas PostgreSQL replication ni MariaDB replication.**
> On construit la **couche d’orchestration, de supervision, d’audit et d’exécution contrôlée.**

Synthèse architecture cible

plan recommandé

- agents Python sur chaque serveur
- control plane Django central
- PostgreSQL streaming replication
- MariaDB GTID replication
- fichiers via rsync/lsyncd + snapshots
- witness/quorum séparé
- Nginx/HAProxy pour la réorientation du trafic
- failover manuel assisté en V1
- fencing et auto-failover plus tard

PLAN RECOMMANDE



Périmètre V1 exact

A. Un agent Linux Python

Installé sur chaque serveur :

- collecte statut machine
- collecte statut PostgreSQL/MariaDB/Nginx/rsync
- expose une API locale sécurisée
- exécute des commandes autorisées
- remonte heartbeat + health

B. Un control plane Django

Centralisé :

- inventaire des nœuds
- définition des services répliqués
- collecte des états
- dashboard
- journal d'événements
- déclenchement de procédures manuelles assistées

ARCHITECTURE SRDF V1

Périmètre V1 exact -B

C. Un moteur de jobs

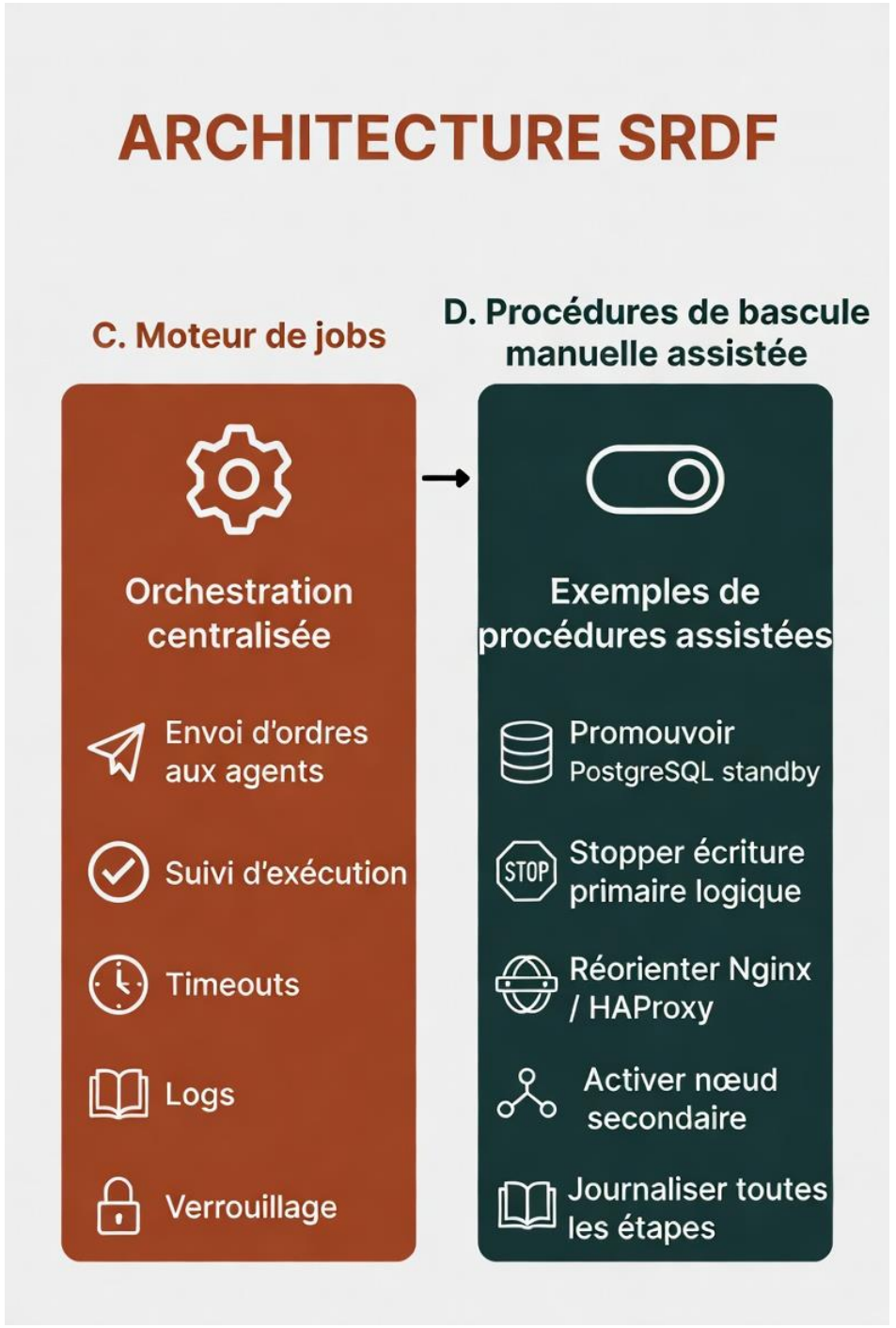
- envoi d'ordres aux agents
- suivi d'exécution
- timeouts
- logs
- verrouillage

D. Des procédures de bascule manuelle assistée

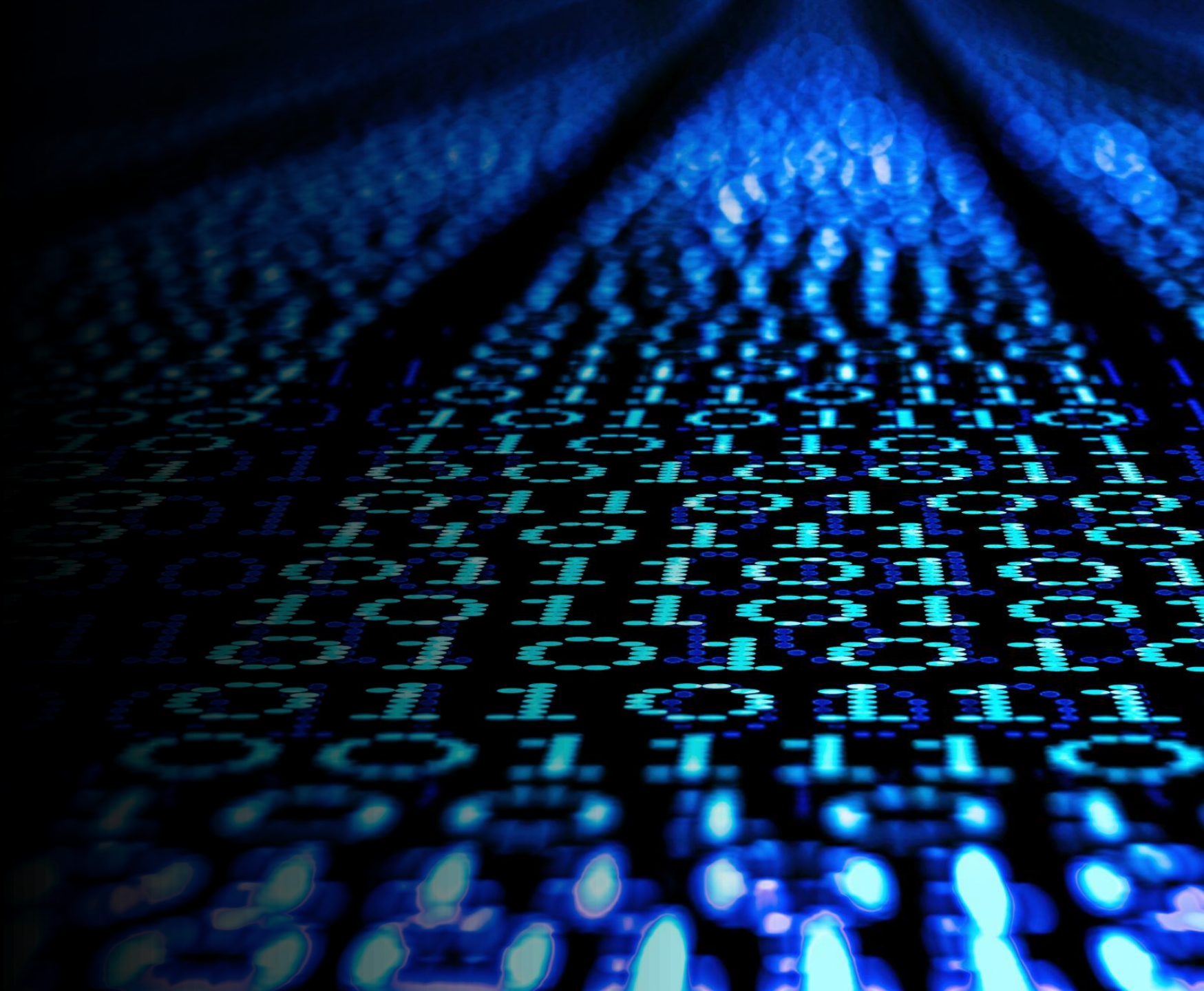
Exemples :

- promouvoir PostgreSQL standby
- stopper écriture primaire logique
- réorienter Nginx/HAProxy
- activer nœud secondaire
- journaliser toutes les étapes

SRDF ENGINES & PROCEDURES



1. Ce qu’on code en V1



1.1 Périmètre V1 exact

A. Un agent Linux Python

Installé sur chaque serveur :

- collecte statut machine
- collecte statut PostgreSQL/MariaDB/Nginx/rsync
- expose une API locale sécurisée
- exécute des commandes autorisées
- remonte heartbeat + health

B. Un control plane Django

Centralisé :

- inventaire des nœuds
- définition des services répliqués
- collecte des états
- dashboard
- journal d'événements
- déclenchement de procédures manuelles assistées

C. Un moteur de jobs

- envoi d'ordres aux agents
- suivi d'exécution
- timeouts
- logs
- verrouillage

D. Des procédures de bascule manuelle assistée

Exemples :

- promouvoir PostgreSQL standby
- stopper écriture primaire logique
- réorienter Nginx/HAProxy
- activer nœud secondaire
- journaliser toutes les étapes

7. Catalogue d’actions autorisées agent

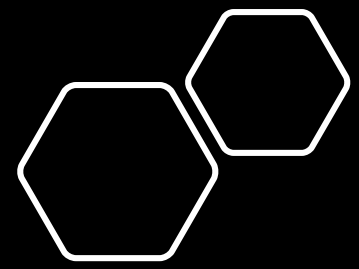
Je te propose une whitelist dès la V1.

7.1 Actions système

- `system.collect_status`
- `system.reload_config`
- `system.restart_service`
- `system.start_service`
- `system.stop_service`

7.2 PostgreSQL

- `postgres.status`
- `postgres.promote`
- `postgres.check_recovery`
- `postgres.pause_writes`
- `postgres.resume_writes`



7.3 MariaDB

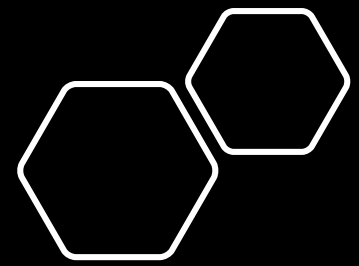
- `mariadb.status`
- `mariadb.set_read_only`
- `mariadb.set_read_write`
- `mariadb.stop_replica`
- `mariadb.start_replica`
- `mariadb.reset_replica`

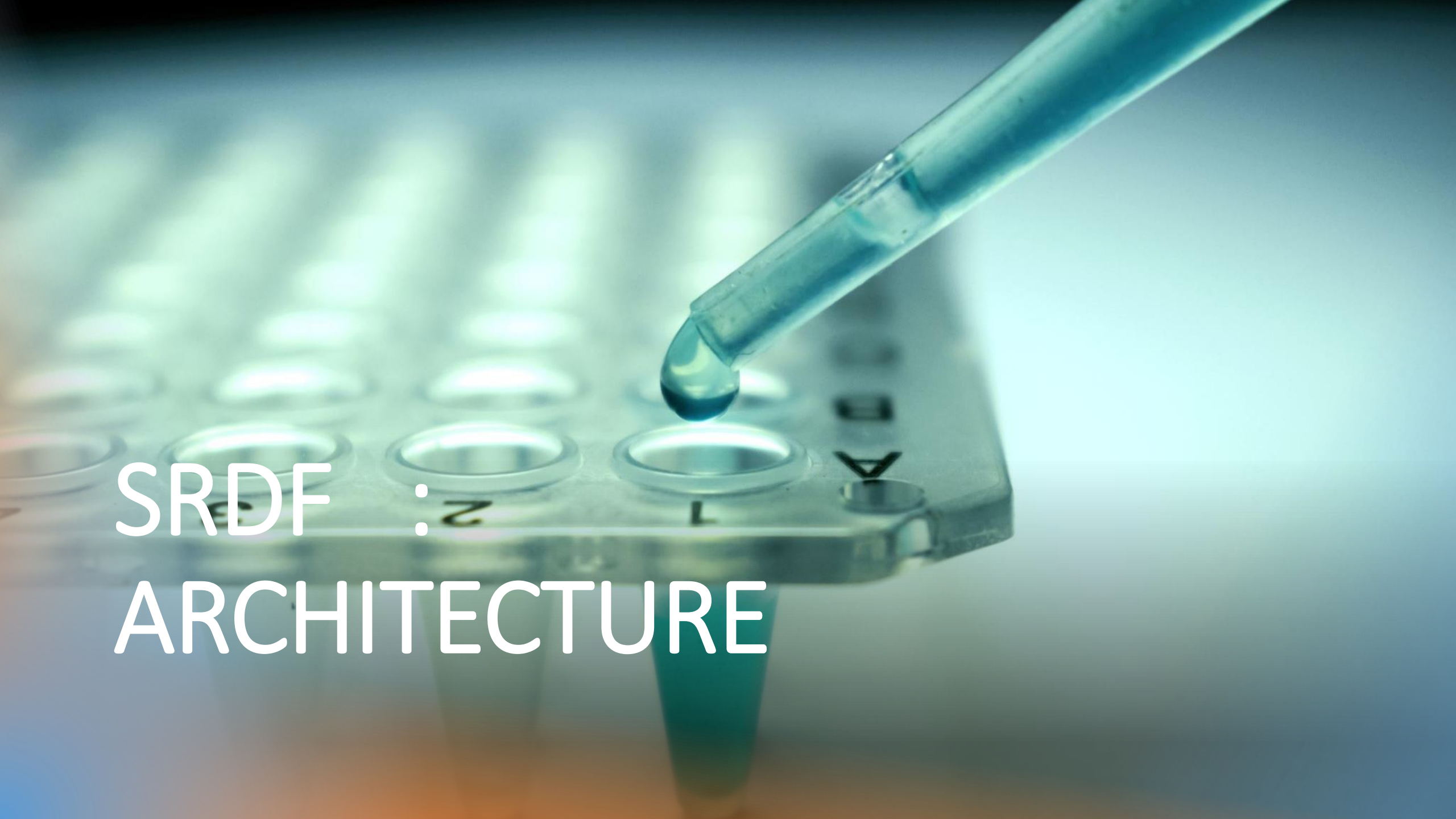
7.4 Nginx / proxy

- `nginx.test_config`
- `nginx.reload`
- `nginx.switch_upstream_profile`

7.5 File sync

- `filesync.run_rsync`
- `filesync.snapshot_create`
- `filesync.verify_checksum`





SRDF : ARCHITECTURE

LOG BIN

SETUP LOGGIING

```bash
mysql -uroot -P3306
```

Puis :

```sql
SHOW VARIABLES LIKE 'log_bin';
```

👉 attendu :

```
log_bin = ON
```

SET LOGGING ON

❌ Si OFF

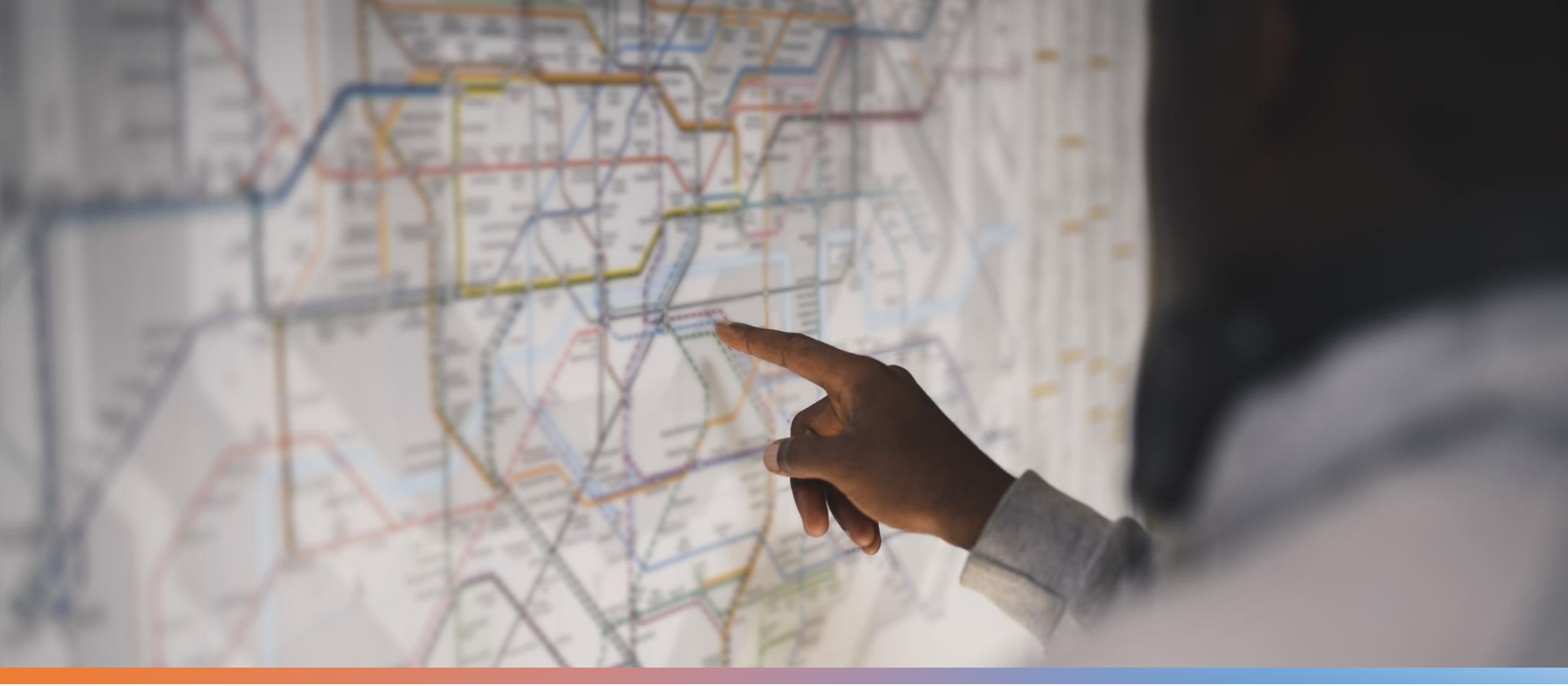
Dans `my.ini` MySQL :

```ini
[mysqld]
server-id=1
log_bin=mysql-bin
binlog_format=ROW
```

Puis redémarre MySQL.

```ini
[mysqld]
server-id=1
log_bin=mysql-bin
binlog_format=ROW
binlog_row_image=FULL
```

La documentation du projet indique explicitement l'usage pour recevoir les événements insert/update/delete, et mentionne pour MySQL 8+ des réglages comme `binlog_row_image='FULL'` / `binlog_row_metadata='FULL'` selon version. GitHub



plan logique SRDF

SRDF : CONCEPTS DE BASE

- ARCHITECTURE DE BASE
- CHUNKING / BATCHING
- Compression puis chiffrement
- Accusé de réception et idempotence
- Retry / backoff / dead letter
- GESTION DES ERREURS
- Choix d’exécution : Celery ou pas
- Pipeline concret recommandé
- Pièges : Ce qu’il faut éviter absolument

SRDF : CONCEPTS DE BASE

ARCHITECTURE DE BASE

- `srdf_change_event`
- `srdf_outbox_batch`
- `srdf_batch_item`
- `srdf_delivery_attempt`
- `srdf_remote_ack`

CHUNKING / BATCHING

3. Le chunking / batching

Tu as totalement raison : il faut grouper.

Un batch ne doit pas partir "à chaque modif", mais selon des règles du type :

- **seuil en nombre d'events**
  ex. 100, 500, 1000
- **seuil en taille**
  ex. 256 KB, 1 MB, 5 MB compressé/non compressé
- **seuil temporel**
  ex. si au bout de 5 ou 10 secondes le batch n'est pas plein, on l'envoie quand même
- **segmentation par destination / flux / priorité / type de donnée**

- `destination_id`
- `channel`
- `priority`
- `chunk_no`
- `event_count`
- `raw_size`
- `compressed_size`
- `checksum`
- `encryption_mode`
- `status`

Compression puis chiffrement

4. Compression puis chiffrement

Ordre recommandé :

serialize → compress → encrypt → sign/checksum

Jamais l'inverse.

Pipeline proposé :

1. sérialisation JSON compacte ou binaire
2. compression `zstd` ou `gzip`
3. chiffrement `AES-GCM` ou équivalent moderne
4. hash / checksum / signature technique

Pour être pragmatique :

- **compression** : `zstd` est un très bon candidat
- **chiffrement** : `AES-256-GCM`
- **checksum** : `SHA-256`
- **versioning de trame** indispensable

Accusé de réception et idempotence

Le distant doit pouvoir répondre :

- `accepted`
- `partially_accepted`
- `rejected`
- `duplicate`
- `invalid_payload`

Et côté source, il faut pouvoir marquer :

- `pending`
- `sealed`
- `sending`
- `sent`
- `acked`
- `retry`
- `failed`
- `dead_letter`

L'idempotence est critique :

- `event_uuid`
- `batch_uuid`
- `fingerprint`
- `dedupe_key`

Retry / backoff / dead letter

Un vrai système robuste doit intégrer :

- retry immédiat léger
- retry exponentiel
- seuil max de tentatives
- passage en **dead letter queue**

Exemple :

- tentative 1 : +10 sec
- tentative 2 : +30 sec
- tentative 3 : +2 min
- tentative 4 : +10 min
- tentative 5 : +1 h
- ensuite DLQ

GESTION DES ERREURS

Il faut aussi distinguer :

- erreurs réseau temporaires
- erreurs applicatives distantes
- erreurs de chiffrement
- erreurs de format
- erreurs fonctionnelles irréparables

Choix d’exécution : Celery ou pas

Option B — Cron/daemon/management commands

Franchement, pour SRDF/CRDF, ça me paraît souvent plus propre :

- `build_pending_batches`
- `send_ready_batches`
- `retry_failed_batches`
- `poll_remote_acks`
- `purge_old_events`
- `reconcile_missing_acks`

Tu peux les lancer via :

- cron toutes les X secondes/minutes
- `systemd`
- superviseur
- ou boucle permanente contrôlée

Pipeline concret recommandé

A. Capture

- création de `srdf_change_event`

B. Normalisation

- enrichissement minimal
- calcul fingerprint/dedupe key

C. Groupement

- sélection des events `pending`
- création d'un `srdf_outbox_batch`

D. Sealing

- sérialisation
- compression
- chiffrement
- checksum
- passage à `ready_to_send`

E. Delivery

- envoi HTTP/gRPC/TCP au distant
- stockage de la réponse

F. ACK

- marquage `acked`
- libération/archivage des events source

G. Retry / DLQ

- gestion des échecs

Pièges : Ce qu’il faut éviter absolument

Quelques erreurs classiques à ne pas faire :

- envoi réseau dans la transaction métier
- un batch sans identifiant immuable
- pas de checksum
- pas de version de protocole
- pas d’idempotence côté réception
- dépendre uniquement de Celery/Redis sans persistance DB
- supprimer trop tôt les events locaux
- batchs trop gros non rejouables
- chiffrer sans métadonnées de version / rotation de clé

Modèles minimum à prévoir

- `SRDFNode`
- `SRDFChannel`
- `SRDFChangeEvent`
- `SRDFOutboxBatch`
- `SRDFBatchItem`
- `SRDFDeliveryAttempt`
- `SRDFAck`
- `SRDFDeadLetter`

Et éventuellement :

- `SRDFKeyRing`
- `SRDFProtocolLog`
- `SRDFFlowMetric`

Réglages essentiels

Il faut des paramètres configurables par flux :

- `batch_max_events`
- `batch_max_bytes`
- `batch_max_age_seconds`
- `compression_enabled`
- `compression_codec`
- `encryption_enabled`
- `retry_max_attempts`
- `retry_backoff_policy`
- `ack_timeout_seconds`
- `purge_after_days`

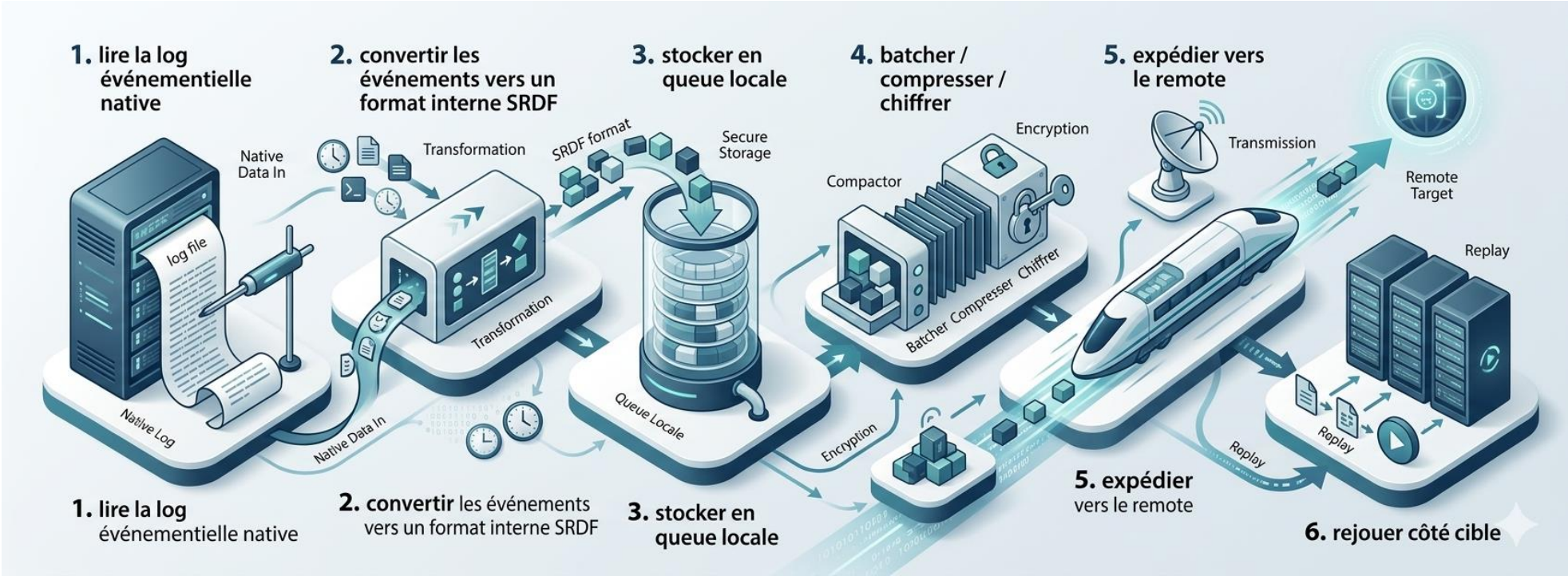
Le vrai coeur du projet

- capturer sans ralentir
- grouper intelligemment
- transporter proprement
- rejouer sans doublon
- survivre aux pannes

SRDF : OBJECTIFS CLES

La vraie base SRDF

1. lire la log événementielle native
2. convertir les événements vers un format interne SRDF
3. stocker en queue locale
4. batcher / compresser / chiffrer
5. expédier vers le remote
6. rejouer côté cible



Ma recommandation d’architecture SRDF Lite

V1 solide

- bootstrap minimal agent source/cible
- paire de réplication persistée
- baseline token
- mode seed “filesystem/code/database”
- queue des deltas
- traitement par batchs/chunks
- reprise sur checkpoint
- état dashboard complet

V2

- mode image/snapshot cloud piloté
- compatibilité AWS/GCP/Azure
- checksum avancé
- détection de divergence fine
- politique de throttling
- priorités de réplication

V3

- quasi-CDP logique
- consistency groups
- multi-volumes / multi-DB
- point-in-time restore
- orchestration de failover/failback

AGENDA PLAN INITIAL SRDF

- Phase A — Initialisation / Point de synchro
- Phase B — Capture continue
- Phase C — Fenêtre temporelle
- Phase D — Consistency Group
- Phase E — Déduplication
- Phase F — Envoi
- Phase G — Inbox côté cible
- Phase H — Apply
- Phase I — Contrôle comparatif

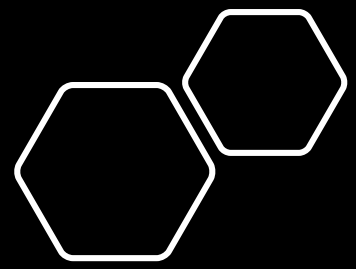
Phase A — Initialisation / Point de synchro

Avant d'envoyer quoi que ce soit, il faut définir le POC = **point de cohérence initial**.

Concrètement :

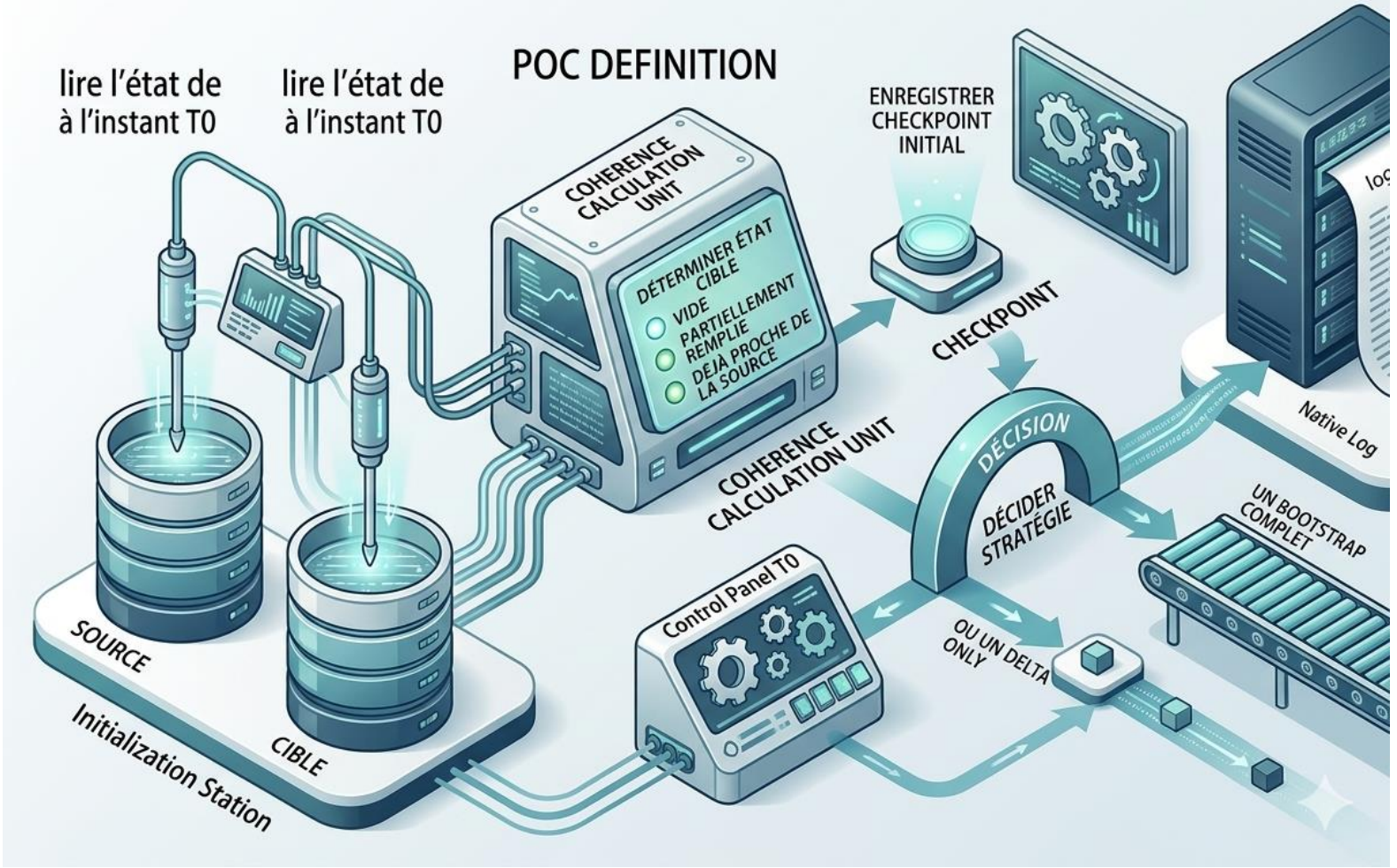
- lire l'état de la source à l'instant T0
- lire l'état de la cible à l'instant T0
- déterminer si la cible est :
  - vide
  - partiellement remplie
  - déjà proche de la source
- enregistrer un **checkpoint initial**
- décider si on fait :
  - un bootstrap complet
  - ou un delta only

👉 Sans ça, on ne sait pas quelles updates envoyer.



Phase A — Initialisation / Point de synchro

Avant d'envoyer quoi que ce soit, il faut définir le POC = point de cohérence initial.



Phase B — Capture continue

À chaque update SQL sur la source :

- le binlog est lu
- on crée une ligne dans `OutboundChangeEvent`

Donc ta question :

> à chaque Update (sql) on log dans une table (c'est déjà fait ????)

👉 **Oui, c'est déjà fait côté source.**

C'est précisément le rôle de `OutboundChangeEvent` models

Phase C — Fenêtre temporelle

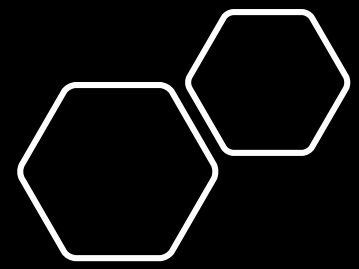
On ne doit pas envoyer chaque ligne immédiatement.

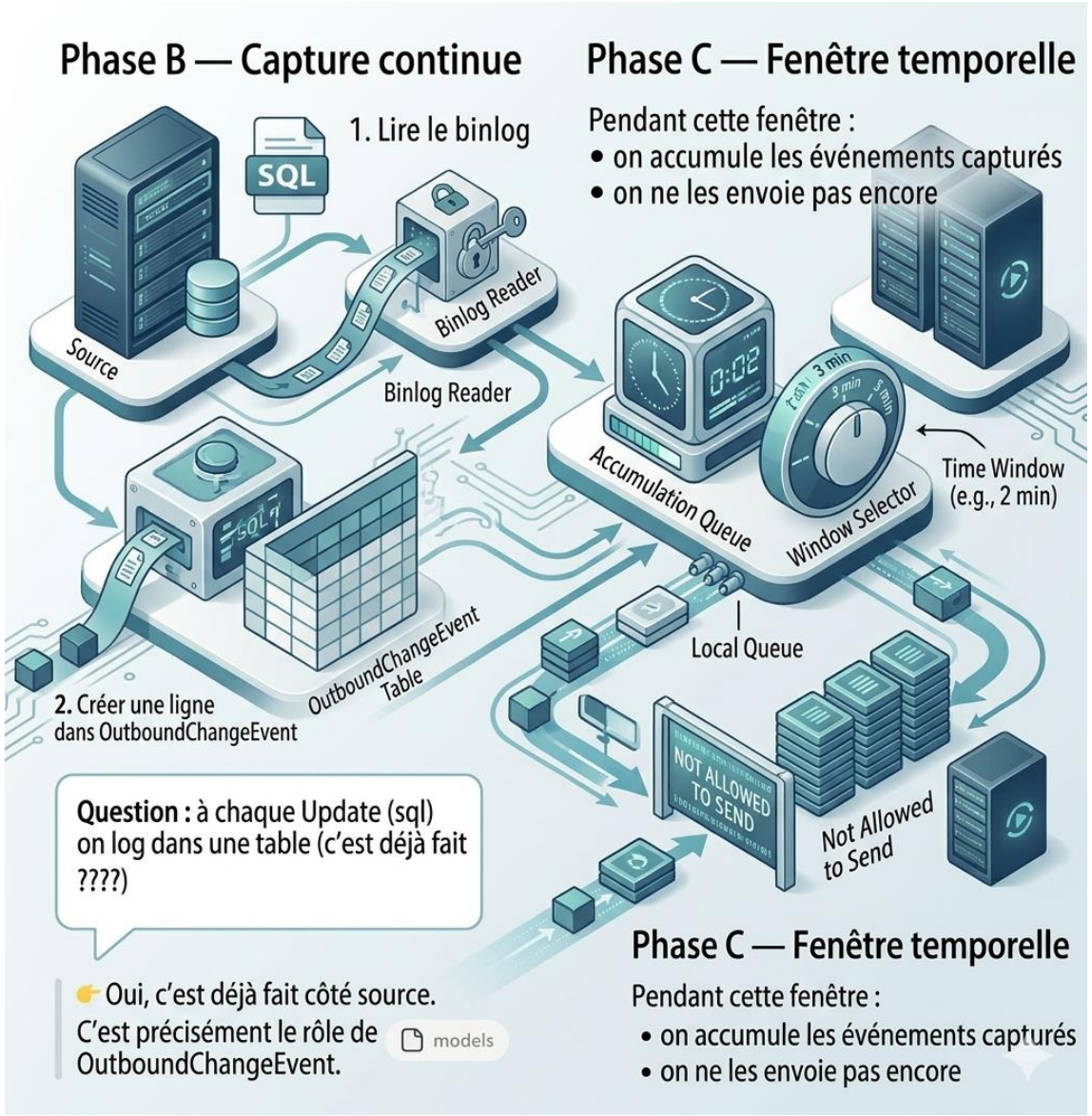
On définit une fenêtre, par exemple :

- 2 min
- ou 3 min
- ou 5 min

Pendant cette fenêtre :

- on accumule les événements capturés
- on ne les envoie pas encore




Phase B — Capture continue
1. Lire le binlog
SQL
Source
Binlog Reader
Binlog Reader
2. Créer une ligne dans OutboundChangeEvent
OutboundChangeEvent Table
Phase C — Fenêtre temporelle
Pendant cette fenêtre :
• on accumule les événements capturés
• on ne les envoie pas encore
Accumulation Queue
Window Selector
Time Window (e.g., 2 min)
Local Queue
NOT ALLOWED TO SEND
Not Allowed to Send
Question : à chaque Update (sql) on log dans une table (c'est déjà fait ????)
Oui, c'est déjà fait côté source.
C'est précisément le rôle de OutboundChangeEvent.
models
Phase C — Fenêtre temporelle
Pendant cette fenêtre :
• on accumule les événements capturés
• on ne les envoie pas encore

Phase D — Consistency Group

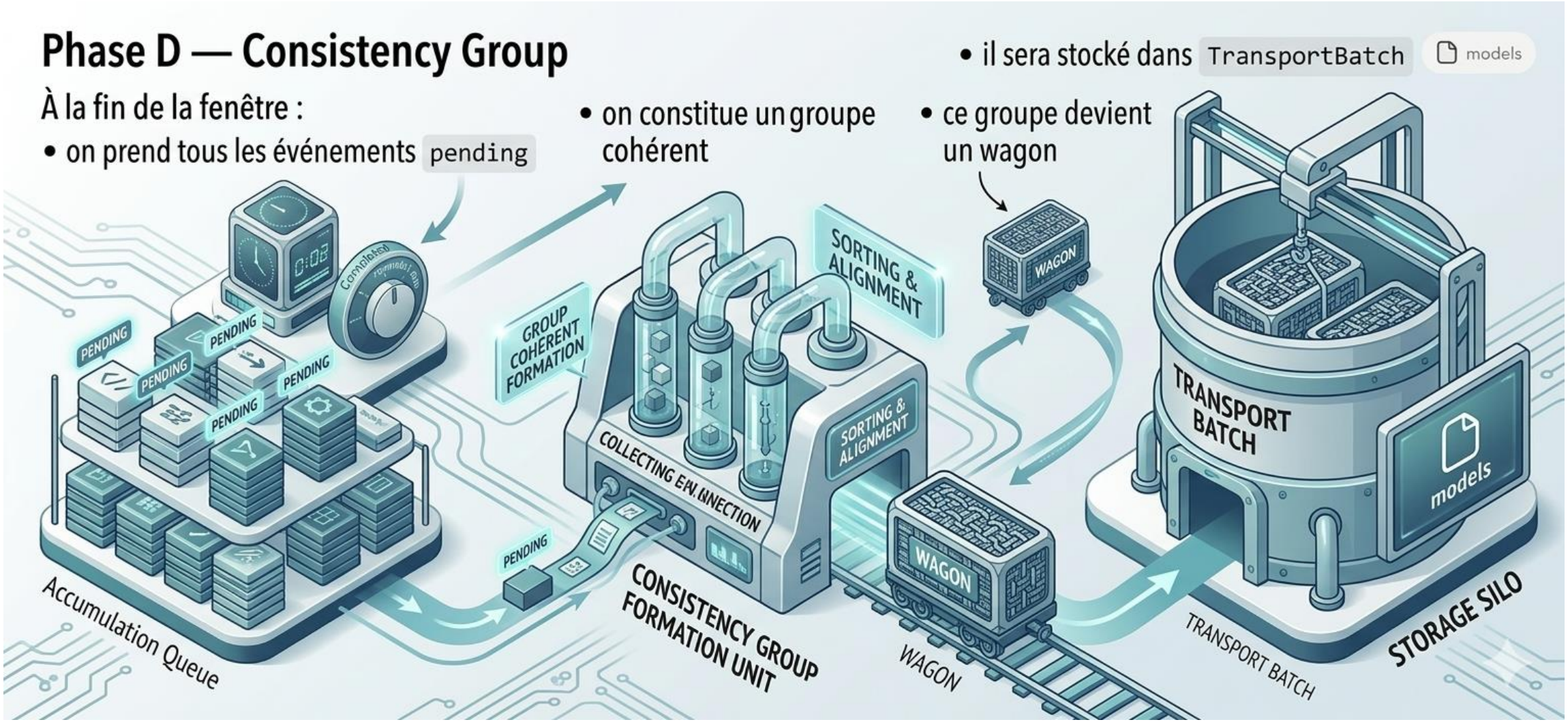
À la fin de la fenêtre :

- on prend tous les événements `pending`
- on constitue un **groupe cohérent**
- ce groupe devient un **wagon**
- il sera stocké dans `TransportBatch` models

Phase D — Consistency Group

À la fin de la fenêtre :

- on prend tous les événements `pending`
- on constitue un groupe cohérent
- ce groupe devient un wagon
- il sera stocké dans `TransportBatch` models



Phase E — Déduplication

Dans ce wagon, on ne doit pas envoyer toutes les versions successives d'un même objet.

Exemple :

- update client id=10
- update client id=10
- update client id=10

👉 On n'envoie que la **dernière image utile** de l'objet.

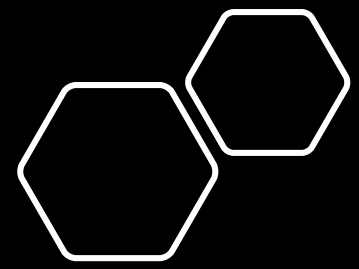
Donc la déduplication doit se faire par clé logique :

- `database_name`
- `table_name`
- `primary_key_data`

Et la règle sera :

- plusieurs `update` sur le même objet → garder le dernier
- `insert` puis `update` → garder un `insert` final enrichi
- `insert` puis `delete` dans la même fenêtre → ne rien envoyer
- `update` puis `delete` → envoyer seulement le `delete`

Ça, c'est le cœur de l'intelligence SRDF.



Phase E — Déduplication

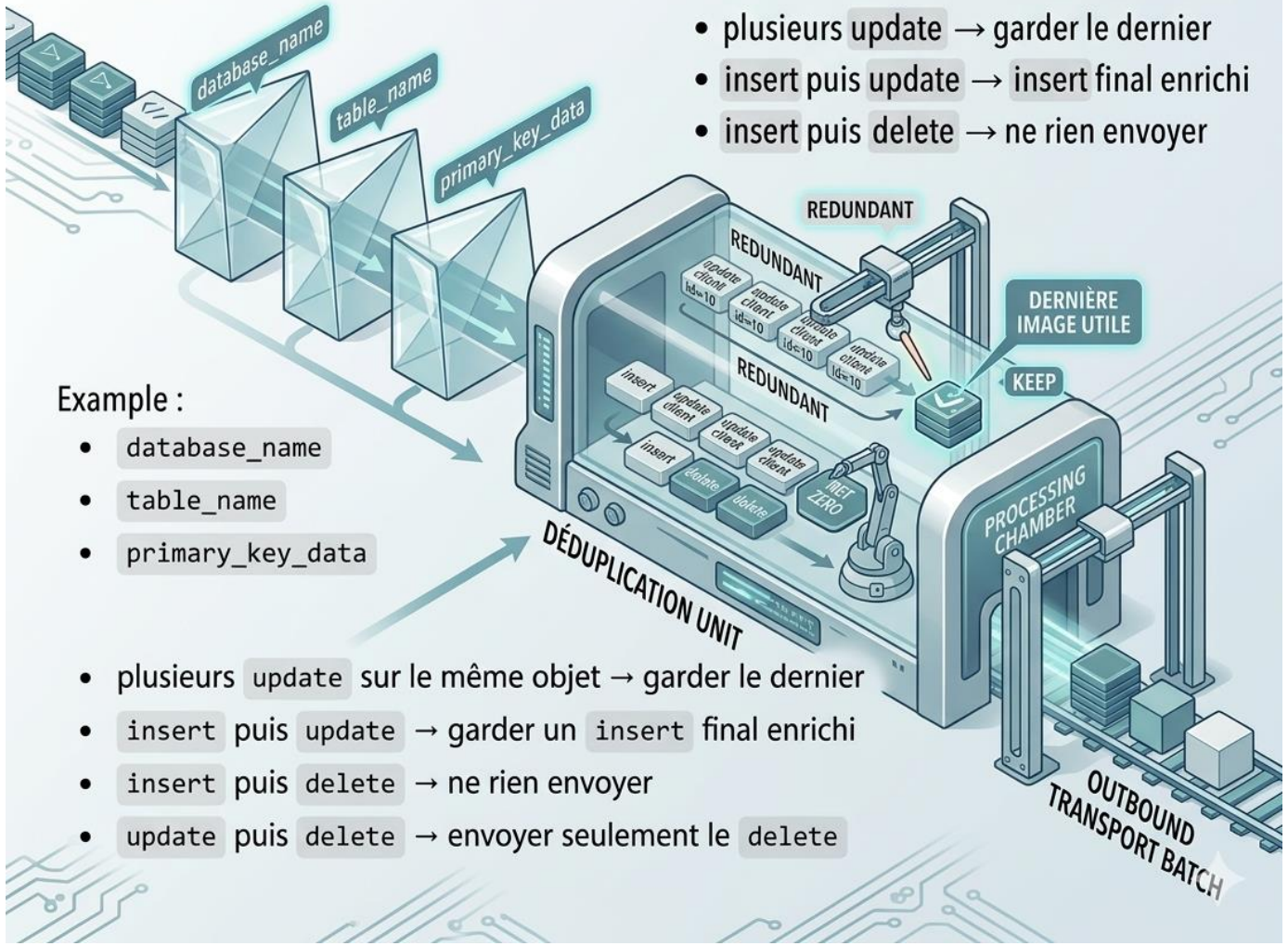
DÉDUPLICATION PAR CLÉ LOGIQUE :

Example :

- `database_name`
- `table_name`
- `primary_key_data`

RÈGLES DE DÉDUPLICATION (COEUR SRDF) :

- plusieurs `update` → garder le dernier
- `insert` puis `update` → `insert` final enrichi
- `insert` puis `delete` → ne rien envoyer



- plusieurs `update` sur le même objet → garder le dernier
- `insert` puis `update` → garder un `insert` final enrichi
- `insert` puis `delete` → ne rien envoyer
- `update` puis `delete` → envoyer seulement le `delete`

Phase F — Envoi

Une fois le consistency group prêt :

- sérialisation JSON
- checksum
- éventuellement compression
- envoi HTTP vers la cible

Phase G — Inbox côté cible

La cible ne doit pas appliquer directement.

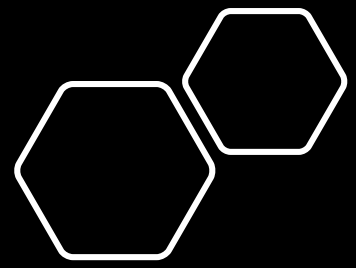
Elle doit d'abord recevoir dans une queue cible :

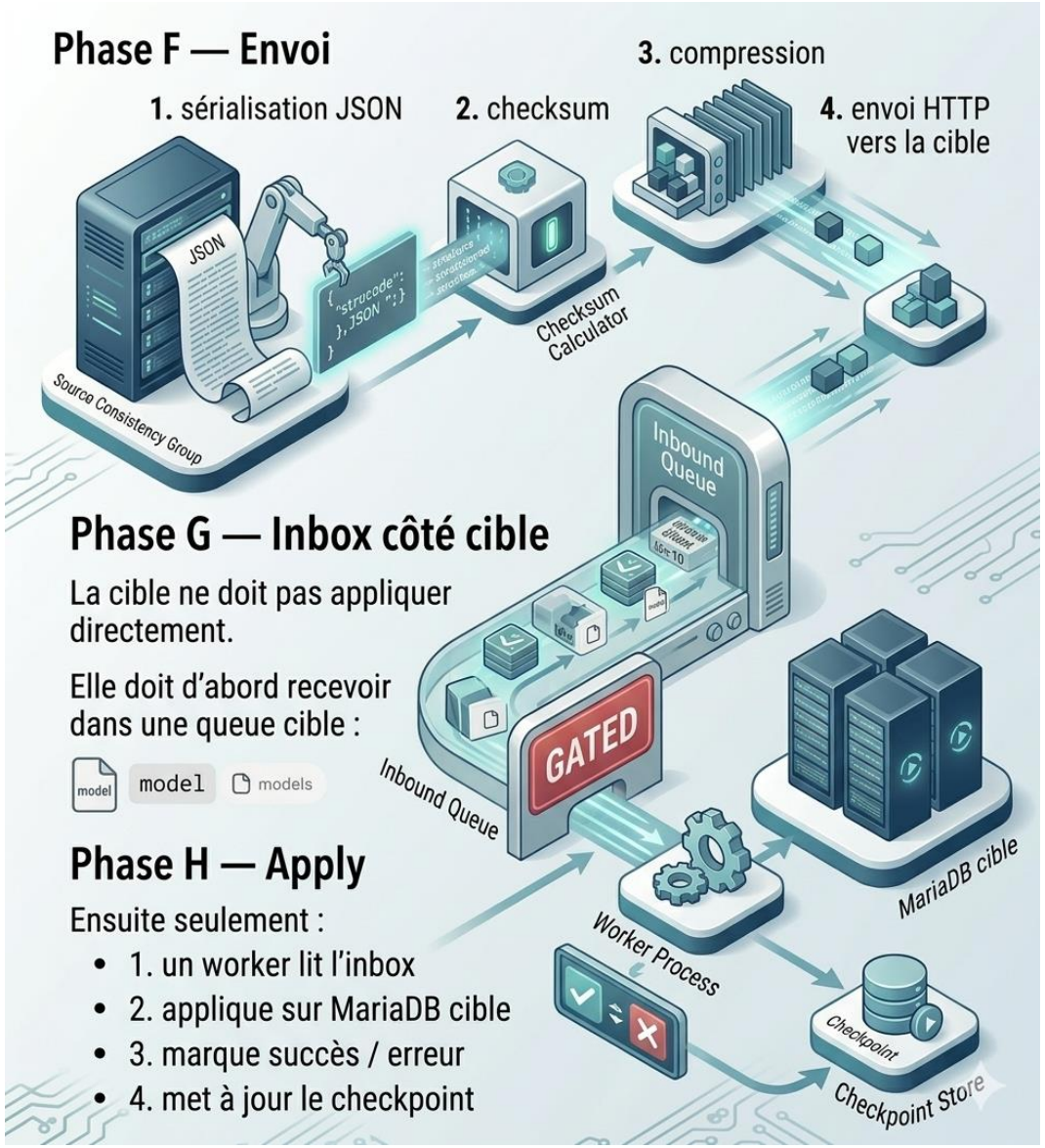
- `InboundChangeEvent` models

Phase H — Apply

Ensuite seulement :

- un worker lit l'inbox
- applique sur MariaDB cible
- marque succès / erreur
- met à jour le checkpoint




Phase F — Envoi
1. sérialisation JSON
2. checksum
3. compression
4. envoi HTTP vers la cible
JSON
Source Consistency Group
Checksum Calculator
Inbound Queue
Phase G — Inbox côté cible
La cible ne doit pas appliquer directement.
Elle doit d'abord recevoir dans une queue cible :
model
model
models
GATED
Inbound Queue
MariaDB cible
Phase H — Apply
Ensuite seulement :
1. un worker lit l'inbox
2. applique sur MariaDB cible
3. marque succès / erreur
4. met à jour le checkpoint
Worker Process
Checkpoint
Checkpoint Store

Phase I — Contrôle comparatif

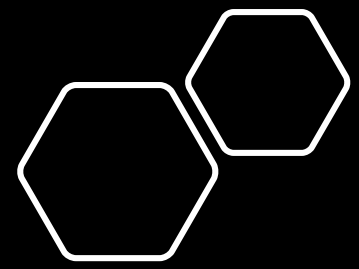
Et tu as raison : sans comparatif source/cible, on pilote à l'aveugle.

Il faut donc un mécanisme de contrôle :

- par table
- par plage d'IDs
- par checksum logique
- ou par timestamp + cardinalité

Pour savoir :

- est-ce que la cible a bien convergé
- est-ce qu'il manque des objets
- est-ce qu'on doit rejouer



PLAN D’ATTAQUE CONCRET POUR V 1.1

- 1. Local agent daemon par moteur
- 2. Control plane Django central
- 3. Data plane dédié source → remote
- 4. Le meilleur mode de communication
- 5. Plan de dev étape par étape

1. Local agent daemon par moteur

- un process Linux/systemd par serveur
- boucle continue
- charge un `engine_adapter` selon le type : mariadb, mysql, postgresql, oracle
- publie une API locale sécurisée
- remonte heartbeat + health + backlog + erreurs

2. Control plane Django central

- inventaire, services, audits, dashboards, commandes
- pas le chemin critique data plane
- il pilote, il n'achemine pas le flux lourd

3. Data plane dédié source → remote

3. Data plane dédié source → remote

- pas via Django synchrone pour les gros batchs
- batch binaire/JSON compact
- compression puis chiffrement
- envoi HTTP API entre agents dans un premier temps
- retry/backoff/idempotence côté agent

4. Le meilleur mode de communication

Pour ton point 6, mon choix pour la V1 est :

HTTP(s) API agent-to-agent, avec POST de batchs, accusés de réception JSON, et éventuellement polling de health.

Pourquoi :

- simple à opérer
- compatible systemd, nginx, logs, retry
- facile à sécuriser par token + mTLS ensuite
- très bon pour un modèle batch/outbox
- beaucoup plus simple que websocket pour ce cas
- microservice oui, mais en pratique : **agent Python exposant une petite API REST interne**

Ce qu’il faut faire maintenant, dans le bon ordre

- Étape 1 — finaliser la capture MariaDB
- Étape 2 — construire le premier consistency group local
- Étape 3 — ajouter un vrai daemon shipper
- Étape 4 — construire le remote receiver

Étape 1 — finaliser la capture MariaDB

Étape 1 — finaliser la capture MariaDB

La capture actuelle lit `WriteRowsEvent / UpdateRowsEvent / DeleteRowsEvent`, stocke les événements, et maintient le checkpoint. C'est déjà bien. binlog_capture

Mais pour qu'elle soit "production-grade", il faut encore renforcer 5 points :

- stocker proprement le **log_file réel** sur chaque event, pas vide
- mieux renseigner `transaction_id` quand possible
- gérer la notion de **commit boundary / transaction boundary**
- fiabiliser la détection de PK, car le fallback actuel "first scalar column" est trop faible pour une vraie réplication universelle
- prévoir la capture DDL à part, car le binlog row-based couvre surtout DML

Aujourd'hui, ton code met encore `transaction_id=""` et `log_file=""` dans les events normalisés ; c'est acceptable pour un POC, mais pas pour la suite SRDF. binlog_capture

binlog_capture

Étape 2 — construire le consistency group local

Étape 2 — construire le premier consistency group local

C'est la prochaine étape logique, et franchement la plus importante maintenant.

Le principe :

- lire les `OutboundChangeEvent` en statut `pending`
- limiter par `replication_service`
- prendre une fenêtre simple : ex. 100 events max
- les ordonner par `id`
- les dédupliquer simplement
- construire un `TransportBatch`
- marquer les events comme `batched`

Étape 3 — ajouter un vrai daemon shipper

Étape 3 — ajouter un vrai daemon shipper

Après le batch builder :

- loop systemd
- si pending > 0, tenter de bâtir un batch
- si batch ready > 0 et remote dispo, envoyer
- sinon backoff
- journaliser backlog, rythme, erreurs

Étape 4 — construire le remote receiver

- endpoint `/api/srdf/v1/batches/receive`
- validation checksum / format / auth
- création des `InboundChangeEvent`
- réponse `accepted / duplicate / invalid_payload / partially_accepted / rejected`

Le PDF insiste explicitement sur ces statuts d'ack et sur l'idempotence. SRDF_CONCEPTS

ORGANISATION DES TACHES DE CONCEPTION

A. Consistency Group logique

B. TransportBatch physique

Déduplication : ce qu’on fait maintenant

Compression / chiffrement

Retry / erreurs

Le vrai serveur universel évolutif

Ma recommandation très concrète sur le “wagon”

A. Consistency Group logique

Objet de regroupement métier/temps/transaction :

- `consistency_group_uid`
- `replication_service_id`
- `opened_at`
- `closed_at`
- `seal_reason` : `max_events`, `max_bytes`, `time_window`, `manual`
- `first_event_id`
- `last_event_id`
- `event_count`
- `group_status`

B. TransportBatch physique

Objet de transport réseau :

- `batch_uid`
- `consistency_group_uid`
- `chunk_no`
- `chunk_count`
- `payload`
- `raw_size`
- `compressed_size`
- `compression`
- `encryption_mode`
- `checksum`
- `status`

Déduplication : ce qu'on fait maintenant

Je recommande une **déduplication simple intra-batch** :

clé de dédup :

`(database_name, table_name, primary_key_data)`

règles :

- plusieurs `update` successifs sur la même ligne : on garde le dernier `after_data`
- `insert` puis `update` : on fusionne en `insert` avec le dernier état
- `update` puis `delete` : on garde `delete`
- `insert` puis `delete` dans la même fenêtre : on supprime entièrement du batch
- delete puis insert sur même PK dans la même fenêtre : cas ambigu, à traiter comme deux events séparés en V1 ou comme `upsert_recreate` plus tard

C'est cohérent avec ton objectif "déduplication simple" avant réseau.

Compression / chiffrement

Là aussi, ton PDF est bon : **serialize → compress → encrypt → checksum/signature**, jamais l'inverse. Il recommande zstd/gzip, AES-GCM, SHA-256. SRDF_CONCEPTS

Ma reco pratique V1 :

- sérialisation JSON compacte
- checksum SHA-256
- **pas de chiffrement dans le tout premier batch local sans réseau**
- ensuite :
  - compression `zstd`
  - chiffrement `AES-256-GCM`
  - version de trame obligatoire

Donc pour maintenant :

- on calcule déjà `checksum`
- on stocke `compression=""` ou `"none"`
- on remet le chiffrement à l'étape réseau

Retry / erreurs

Je propose :

- `retry_count` sur batch et event déjà présent, donc on l'exploite
- barème batch :
  - 1 : +10 sec
  - 2 : +30 sec
  - 3 : +2 min
  - 4 : +10 min
  - 5 : +1 h
  - puis failed/dead
- `last_error` toujours rempli
- aucun retry infini

Le vrai serveur universel évolutif

Core

- config loader
- scheduler loop
- health state
- local API
- auth/token/mTLS
- queue manager
- batch builder
- shipper
- receiver
- applier dispatcher
- retry/backoff
- audit/logger
- checkpoint manager

Adapters SQL

- `engine_mariadb.py`
- `engine_mysql.py`
- `engine_postgresql.py`
- `engine_oracle.py`

Chaque adapter implémente une interface du genre :

- `capture_changes_once()`
- `build_identity_key(event)`
- `normalize_event(raw_event)`
- `apply_event(inbound_event)`
- `fetch_checkpoint()`
- `save_checkpoint()`
- `validate_target_compatibility()`

SRDF : DECOUPAGE PAR PHASES

Phase 1 : Finaliser MariaDB capture

Phase 2 : Créer le service

Phase 3 : Brancher une action

Phase 4 : Créer le daemon systemd local

Phase 5 : Créer le remote intake API

Phases 1 & 2

Phase 1

Finaliser MariaDB capture

- enrichir `OutboundChangeEvent`
- fiabiliser PK
- préparer boundary transaction
- ajouter action dédiée `transport.build_batch`

Phase 2

Créer le service :

- `services/transport_batch_builder.py`

Fonctions principales :

- `build_transport_batch_once(replication_service_id, max_events=100, max_batch_bytes=...)`
- `_fetch_pending_events(...)`
- `_dedupe_events_v1(...)`
- `_serialize_batch_payload(...)`
- `_compute_checksum(...)`
- `_mark_events_batched(...)`

Phases 3 & 4

Phase 3

Brancher une action

Dans `action_runner.py`, ajouter :

- `transport.build_batch`
- `transport.build_batch_all`
- plus tard `transport.send_batch`, `transport.retry_batch`

Phase 4

Créer le daemon systemd local

- `srdf_transportd.py`
- boucle infinie
- scan des services actifs
- build batches
- logs
- sleep dynamique selon backlog

Phase 5

Créer le remote intake API

Dans `views.py` ou mieux dans une API dédiée :

- `api_receive_batch`
- validation token
- validation payload/checksum
- création inbound
- réponse ack