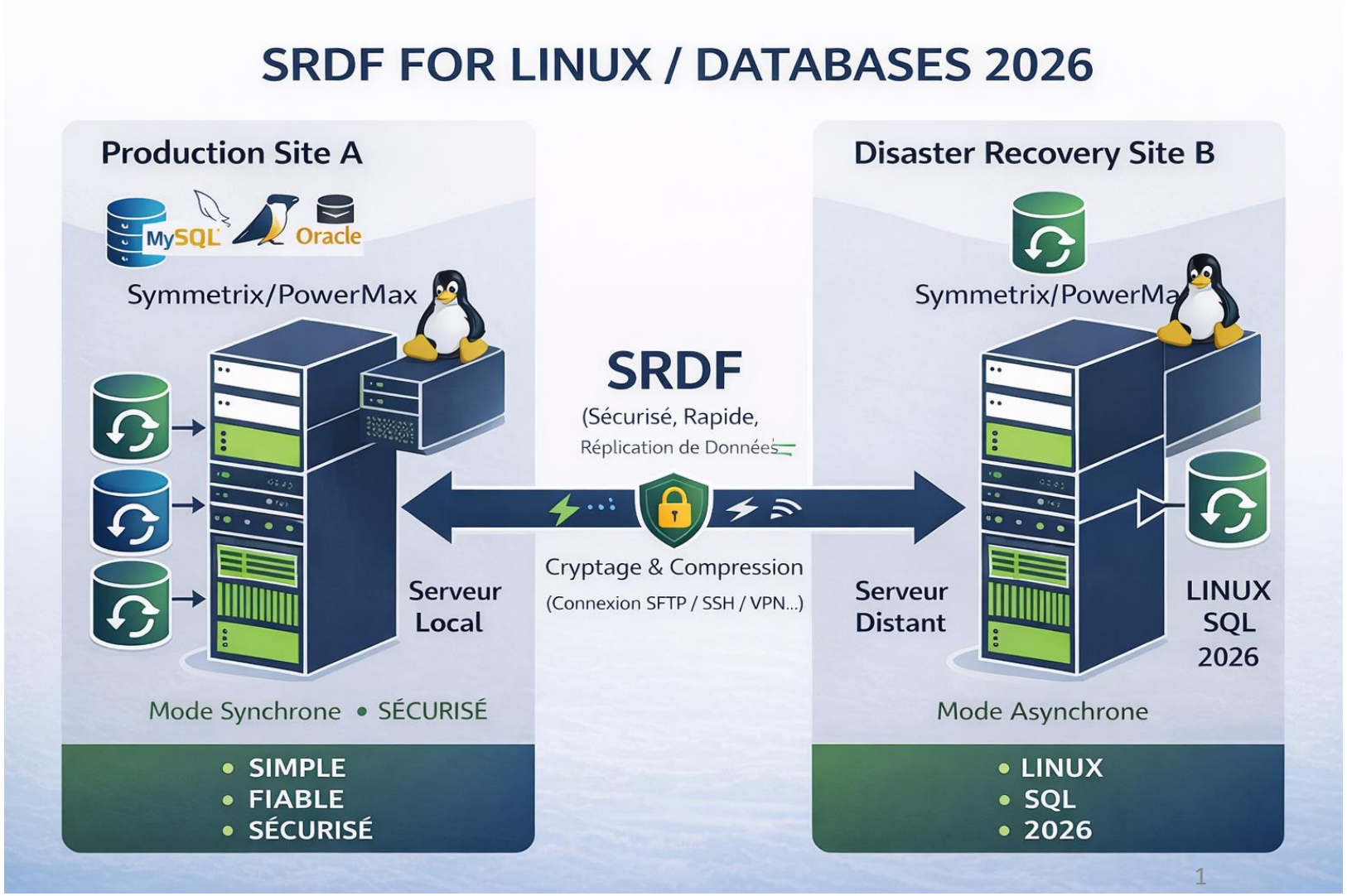
IDEO-LAB

April 2026

v 1.5

SRDF FOR STARTUPS



SRDF PAR IDEO-LAB

"10 fois plus accessible, 10 fois plus flexible, 10 à 20 fois moins cher."

- coût
- simplicité
- vitesse de déploiemen
- flexibilité infra
- indépendance vis-à-vis du hardware propriétaire

POSITIONNEMENT DU SRDF PAR IDEO-LAB

Promesse produit

- 10 à 20 fois moins cher
- beaucoup plus simple à déployer
- agnostique infra : AWS, GCP, Azure, OVH, bare metal, VM Linux, et même Windows si besoin
- pas lié à une baie propriétaire
- orienté reprise d'activité réelle, pas luxe datacenter réservé aux grands groupes

Là où nous sommesvéritablement meilleurs

- Installation légère, autonome, industrialisée
- Réplication multi-moteurs SQL standards : MySQL, MariaDB, puis PostgreSQL
- Compression + chiffrement natifs
- File d'attente / batching intelligent pour limiter le coût réseau
- Priorisation métier des données critiques
- Dashboard clair, logs lisibles, exploitabilité simple
- Fonctionnement sur machines ordinaires, pas sur matériel EMC hors de prix
- Mode "disaster recovery pragmatique" pour petites équipes techniques

SRDF IDEO-LAB VS SRDF Dell

La vraie rupture

Dell vend une solution de très haut de gamme, très robuste, mais lourde, chère, fermée et pensée pour de grosses entreprises.

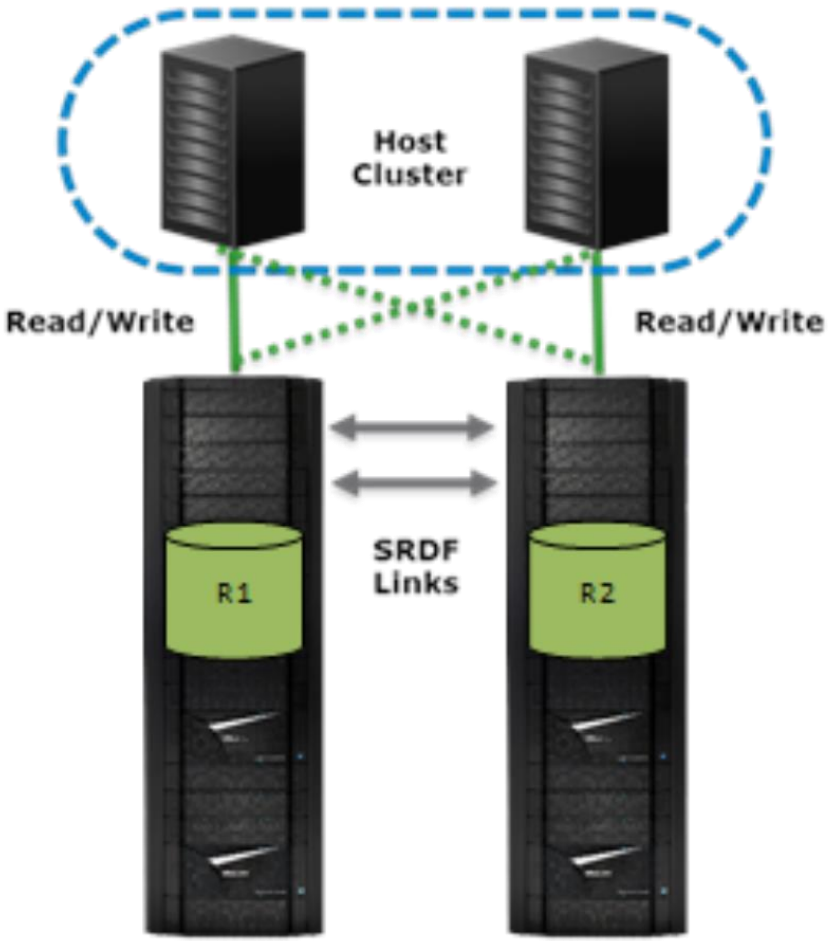
Toi, tu peux construire une solution :

- ouverte
- portable
- déployable partout
- beaucoup plus accessible économiquement
- adaptée aux infrastructures modernes hybrides et cloud

DELL SRDF EN QUELQUES MOTS ET CHIFFRES

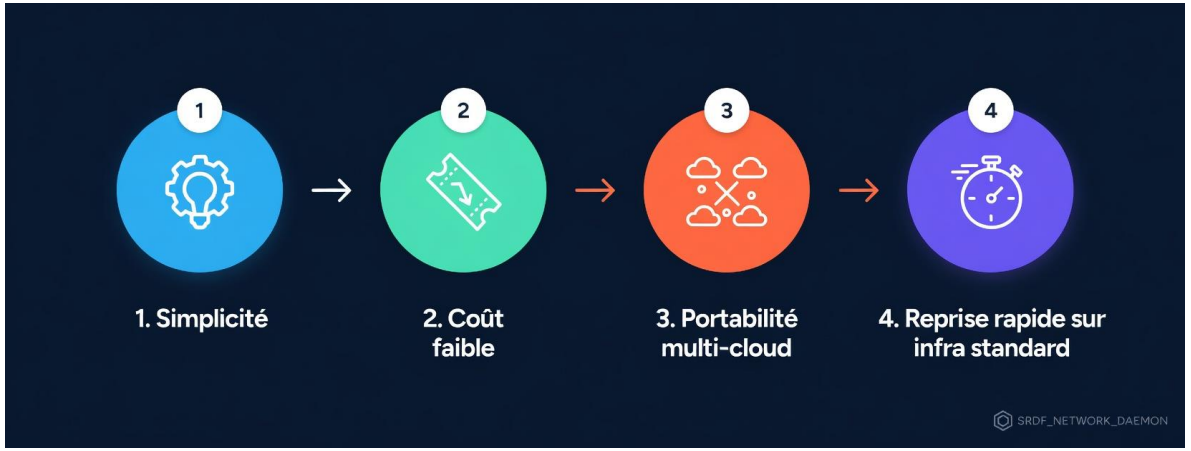
- **Petit périmètre** : quelques To protégés → on peut vite être dans une logique de **quelques milliers à quelques dizaines de milliers d'euros** rien que pour la couche SRDF/licences.
- **Projet entreprise réaliste** avec 2 baies, support, services, réseau, installation → on bascule souvent sur des **budgets de plusieurs dizaines de milliers à plusieurs centaines de milliers d'euros**, voire davantage selon la taille et la haute dispo visée. Cette estimation est une **inférence** à partir du fait que SRDF s'appuie sur des plateformes PowerMax mission-critical et sur une tarification à la capacité/licence, pas un prix public Dell affiché. Dell +2

SRDF / METRO DELL



SRDF IDEO-LAB

1. Simplicité
2. Coût faible
3. Portabilité multi-cloud
4. Reprise rapide sur infra standard



PLAN GENERAL DU DOCUMENT SRDF

Table des Matières

SRDF
PLAN GÈNERAL DU DOCUMENT

TABLE DES MATIERES

PROJET SRDF

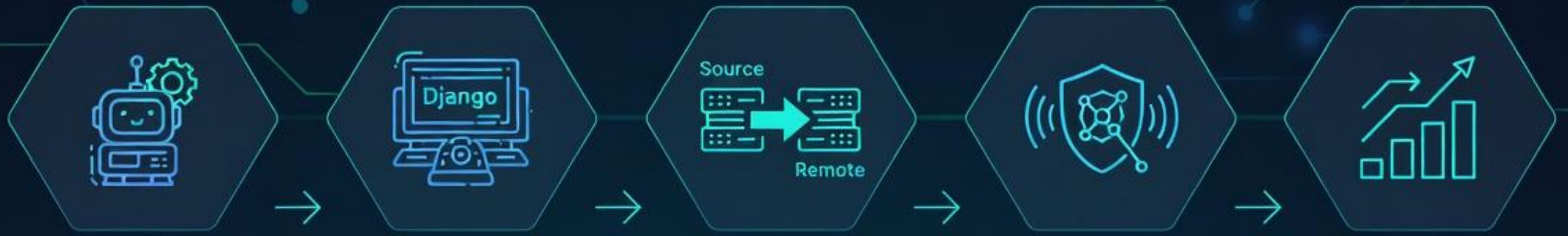
TABLE DES MATIERES - 1-3

TABLE DES MATIERES - 4 - 8

TABLE DES MATIERES - 9-10

SRDF LES OBJECTIFS POURSUIVIS

Architecture SRDF for Startups



Local Agent

Local agent deamon par moteur

Control Plane

Control plane Django central

Data Plane

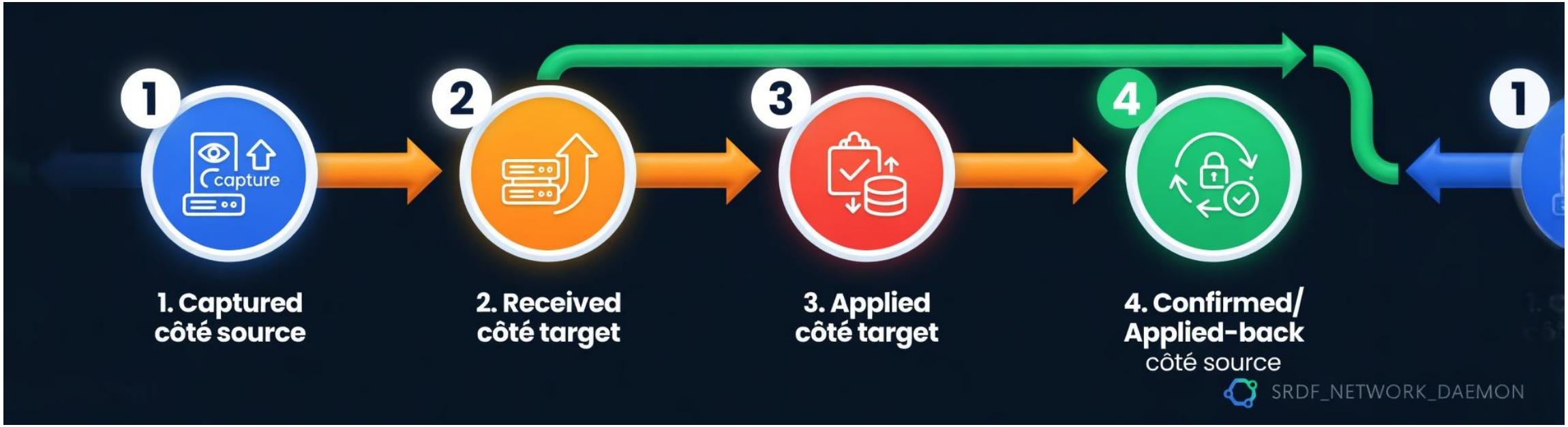
Data plane dédié source → remote

Meilleur Mode

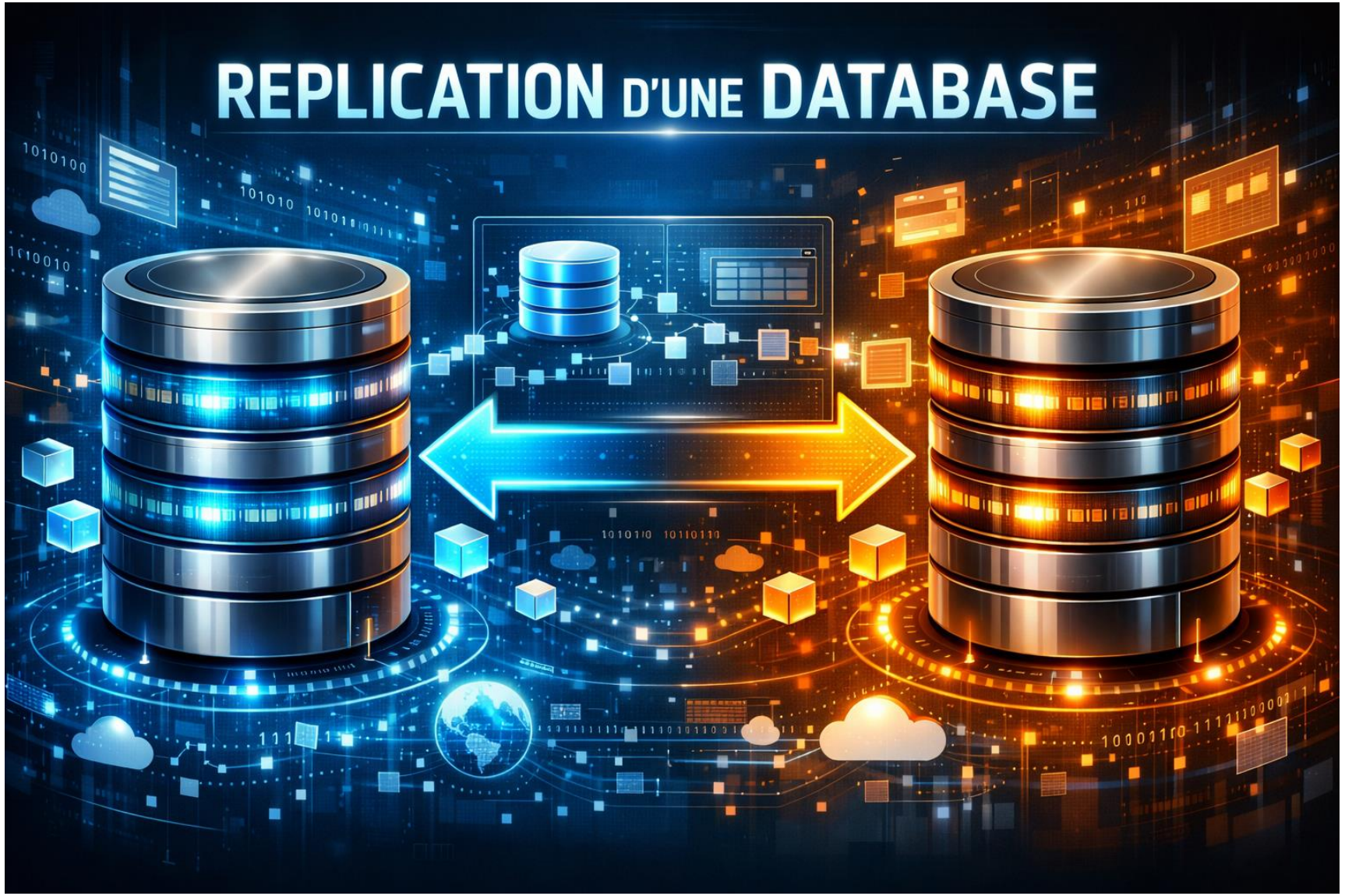
Meilleur mode de communication

Plan de Dev

Plan de dev étape par étape

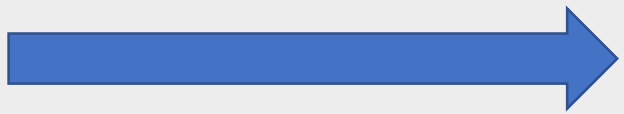

1
capture
2
3
4
1
1. Captured
côté source
2. Received
côté target
3. Applied
côté target
4. Confirmed/
Applied-back
côté source
SRDF_NETWORK_DAEMON

LA REPLICATION SRDF : PERIMETRE



OBJETS REPLIQUES SOUS MARIADB/MYSQL

- DATABASE
- TABLE
- INDEX
- COLONNE



- CREATE
- ALTER
- DROP

DDL

- UPDATE OF TABLE’S ROW
- INSERT VALUE INTO TABLE
- DELETE OF TABLE’S ROW

DML

OBJETS SQL CONCERNES

OBJETS REPLIQUES

OBJETS SOURCE

- DATABASE
- TABLE
- INDEX
- COLONNE

OBJETS REPLIQUES

- DATABASE
- TABLE
- INDEX
- COLONNE

COMMANDES DDL

- CREATE
- ALTER
- DROP

OPERATIONS SUR LES DONNÉES (DML)

UPDATE OF TABLE

Mettre à jour une table

INSERT VALUE INTO TABLE

Insérer des valeurs dans une table

DELETE OF TABLE'S ROW

Supprimer une ligne d'une table

Réplication synchronisée

REPLICATION D'UNE DATABASE

DATABASE SOURCE

PROCESSUS DE REPLICATION

CAPTURE
Lecture des modifications sur la source (Log, Binlog, ...)

TRANSFERT
Envoi des changements à la cible

APPLICATION
Rejeu des opérations sur la base cible

REPLICATION EN COURS...

DATABASE CIBLE (REPLIQUE)

TABLE: UTILISATEURS

id	nom	email
1	Alice	alice@mail.com
2	Bob	bob@mail.com

TABLE: PRODUITS

id	nom	prix
1	Clavier	50
2	Souris	30

TABLE: COMMANDES

id	user_id	total
1	1	80
2	2	30

OBJETS REPLIQUES

DDL (Structure)
- DATABASE
- TABLE
- INDEX
- COLONNE

COMMANDES (DML/DCL)
- CREATE
- ALTER
- DROP

OPERATIONS SUR LES DONNEES
- UPDATE OF TABLE
- INSERT VALUE INTO TABLE
- DELETE OF TABLE'S ROW

TABLE: UTILISATEURS

id	nom	email
1	Alice	alice@mail.com
2	Bob	bob@mail.com

TABLE: PRODUITS

id	nom	prix
1	Clavier	50
2	Souris	30

TABLE: COMMANDES

id	user_id	total
1	1	80
2	2	30

OBJECTIF : ASSURER LA CONSISTANCE ET LA DISPONIBILITE DES DONNEES SUR LA BASE CIBLE

SRDF : C’est quoi ?

Notre solution permet une **synchronisation temps réel** des bases de données (MySQL, MariaDB, PostgreSQL, Oracle) entre un serveur d’origine et un serveur distant, avec une latence minimale et une fiabilité maximale.

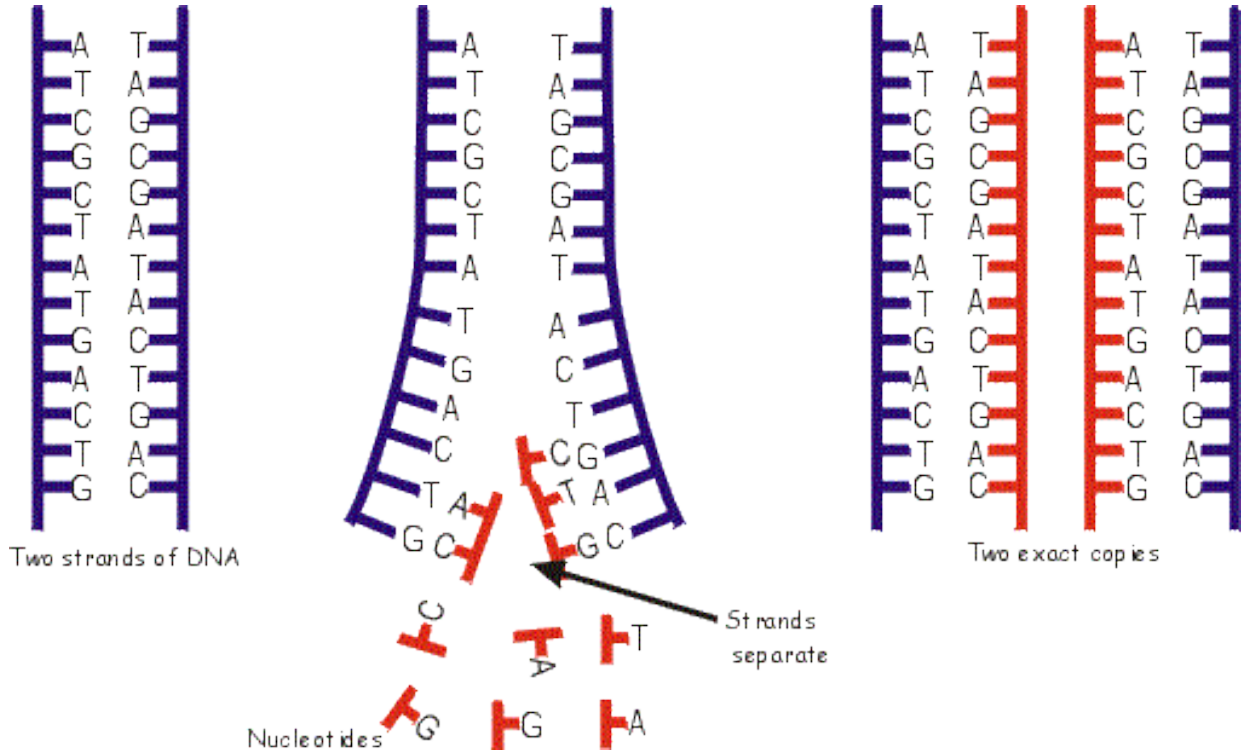
Le processus est le suivant :
Dès qu’une modification SQL (Insert, Update, Delete, Create…) est effectuée sur la base origine, elle est capturée, regroupée par paquets cohérents, dé-dupliquée, compressée et chiffrée.

Un batch s’exécute toutes les 30 secondes pour envoyer uniquement les mises à jour non encore synchronisées. Avant envoi, le système vérifie la disponibilité du serveur distant et gère les erreurs avec reprise automatique.

À réception, les données sont contrôlées pour éviter toute duplication, puis appliquées sur la base cible. Enfin, un accusé de réception est renvoyé pour confirmer la bonne synchronisation.

Grâce à ce mécanisme robuste, nous assurons une **réplication fiable, sécurisée et optimisée** des bases de données, idéale pour les startups qui ont besoin de haute disponibilité, de reprise après sinistre ou de bascule géographique sans perte de données.

LA REPLICATION



LA REPLICATION D'UN SERVEUR - SYSTEM

3. Delta services

État et définition :

- service activé/désactivé
- service ajouté
- override systemd modifié
- timer ajouté
- cron modifié

4. Delta data applicative hors DB

Exemples :

- uploads critiques
- fichiers générés
- cache persistant utile
- répertoires métiers

2. Delta packages / runtimes

Ce qui a été installé, mis à jour, supprimé :

- apt/dpkg
- pip/venv
- éventuellement npm, binaries maison, etc.

3. Delta services

État et définition :

- service activé/désactivé
- service ajouté
- override systemd modifié
- timer ajouté
- cron modifié

1. Delta filesystem

Tout ce qui change dans les zones utiles :

- `/etc/nginx/`
- `/etc/systemd/`
- `/etc/mysql/`
- `/opt/app/`
- `/var/www/...`
- scripts d'exploitation
- certificats
- templates
- fichiers de config métiers

Pas tout le filesystem, seulement les zones pilotées.

LA REPLICATION D'UN SERVEUR - CONFIG

5. Delta configuration système

- firewall
- users/groups techniques
- sudoers ciblés
- variables d'environnement de services
- montages ou points de montage logiques

6. Delta secrets / certificats

À traiter avec précaution extrême :

- renouvellement TLS
- clés privées
- tokens applicatifs
- credentials de réplication

LA REPLICATION : A NE JAMAIS FAIRE !

Ce qu'il ne faut surtout pas faire

Tu as aussi raison implicitement sur un point important : il ne faut pas tomber dans le piège du **"replicate everything"**.

Sinon on refait un mauvais clone de disque.

Il ne faut pas répliquer naïvement :

- `/proc`
- `/sys`
- `/dev`
- sockets
- pid files
- caches temporaires
- gros logs volatils
- machine-id tel quel
- clés host SSH si non voulu
- IP / identité réseau source
- artefacts temporaires du noyau

LA REPLICATION : CONCEPTS

A. Une baseline

On capture une photographie de référence du serveur :

- packages
- services
- arbres surveillés
- configs
- users techniques
- métadonnées critiques

B. Un journal de changements

Ensuite, on ne renvoie plus tout.

On envoie uniquement des événements ou mini-deltas :

- fichier X modifié
- fichier Y supprimé
- paquet Z installé
- service nginx restart demandé
- override systemd changé
- cert renouvelé
- venv hash modifié

C. Un apply engine sur la cible

La cible reçoit ces deltas et les applique :

- dans l'ordre,
- avec version,
- avec checksum,
- avec validation,
- avec rollback local si besoin.

LA REPLICATION : ENGINE

Option 2 — Scan incrémental très fréquent

Toutes les N secondes :

- checksum/mtime/size
- diff des services
- diff paquets
- diff configs
- diff users/groups ciblés

Avantages :

- plus robuste
- plus simple à développer
- plus facile à auditer

Inconvénients :

- moins instantané
- plus coûteux si mal conçu

LA REPLICATION : RPO

La notion clé : RPO quasi nul, pas “backup”

Tu décris un objectif de type :

- **RPO très faible**, potentiellement < 1 minute
- **RTO très faible**, quelques minutes
- sans refaire une restauration massive

Donc le produit doit viser :

- serveur cible déjà prêt,
- déjà provisionné,
- déjà compatible,
- déjà “chaud”,
- simplement maintenu en phase.

Le jour du sinistre, on ne “restaure” pas.
On fait plutôt :

- stop final des flux,
- validation dernier delta,
- promotion,
- remapping réseau/DNS,
- reprise de services.

C’est très différent d’un backup.

LA REPLICATION : LES REGLES

1. Ordre des opérations

Exemple :

- copier config avant restart service
- installer package avant déployer override
- restaurer cert avant reload nginx

Donc chaque delta doit être typé et ordonnançable.

2. Idempotence

Si un delta est rejoué deux fois, la cible ne doit pas casser.

3. Vérification

Après application :

- checksum fichier
- état service
- version paquet
- healthcheck applicatif

4. Reconciliation

Si un delta a été raté :

- rescan
- diff
- resync sélectif

5. Cohérence multi-objets

Exemple :

- config nginx + cert + app + socket + service doivent converger ensemble.

LA REPLICATION : BASE ARCHITECTURE

1. Host Agent

Installé sur le primaire :

- observe
- capture
- compacte
- signe/chiffre
- pousse les deltas

2. SRDF Control Plane

Dans Django :

- inventaire
- policies
- filesets
- snapshots
- delta logs
- état des nœuds
- supervision
- commandes de failover

3. Target Apply Agent

Sur le secondaire :

- reçoit
- met en file
- applique
- vérifie
- expose son état

4. Promotion Engine

En cas d'incident :

- gèle la file
- valide dernier état cohérent
- démarre les services dans l'ordre
- bascule IP / DNS / rôle

LA REPLICATION : Déploiement

V1 réaliste

Pour rester sérieux techniquement, la V1 pourrait se limiter à :

- réplication de fichiers ciblés
- réplication paquets apt/dpkg
- réplication services systemd
- réplication cron/timers
- réplication certificats/configs
- snapshots de cohérence
- commande de promotion manuelle

Sans toucher encore à :

- réseau complexe
- kernel tuning avancé
- containers
- block devices
- SELinux/AppArmor avancé
- identité machine complète

LA REPLICATION : Déploiement

V2

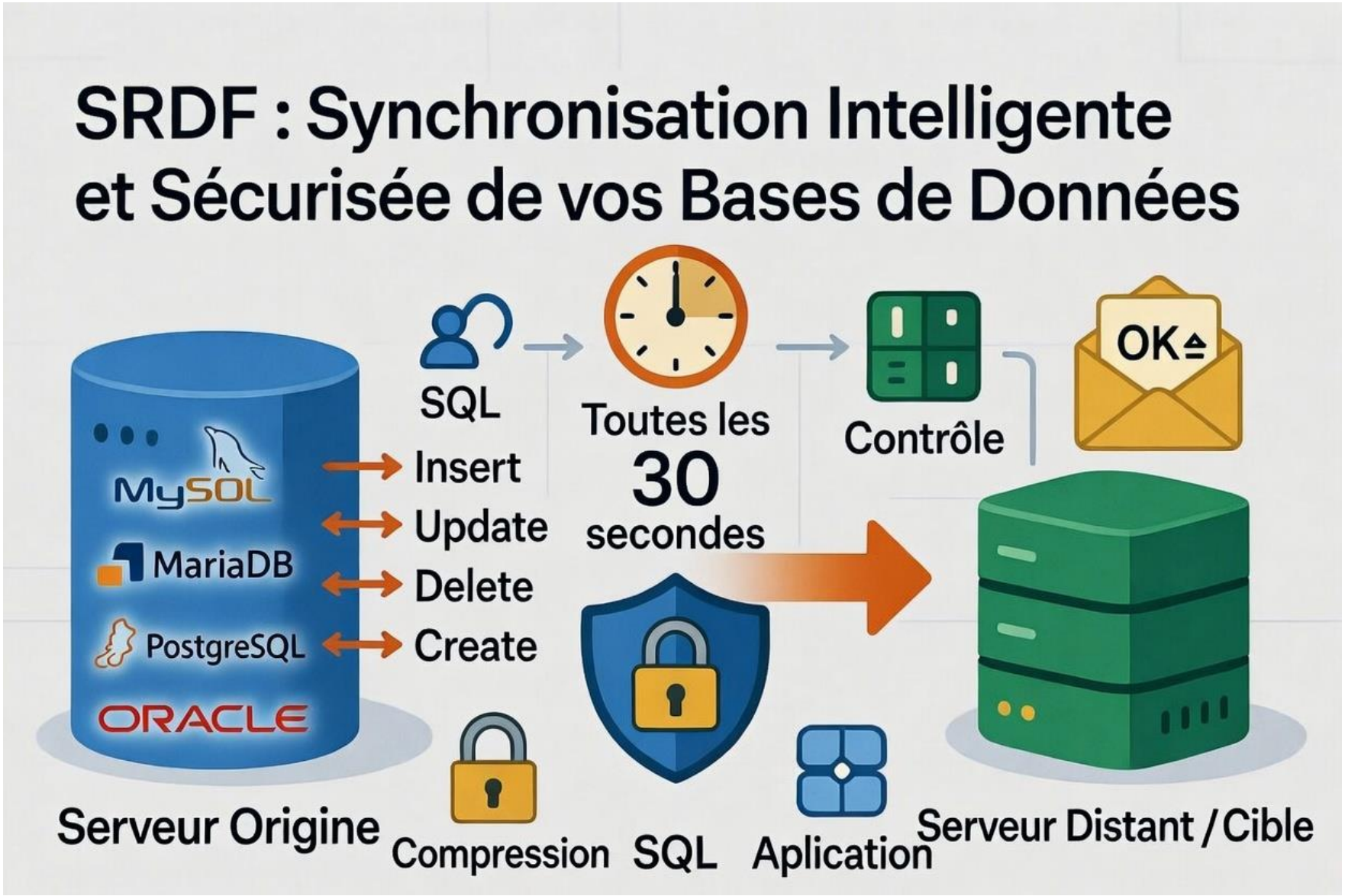
Ensuite :

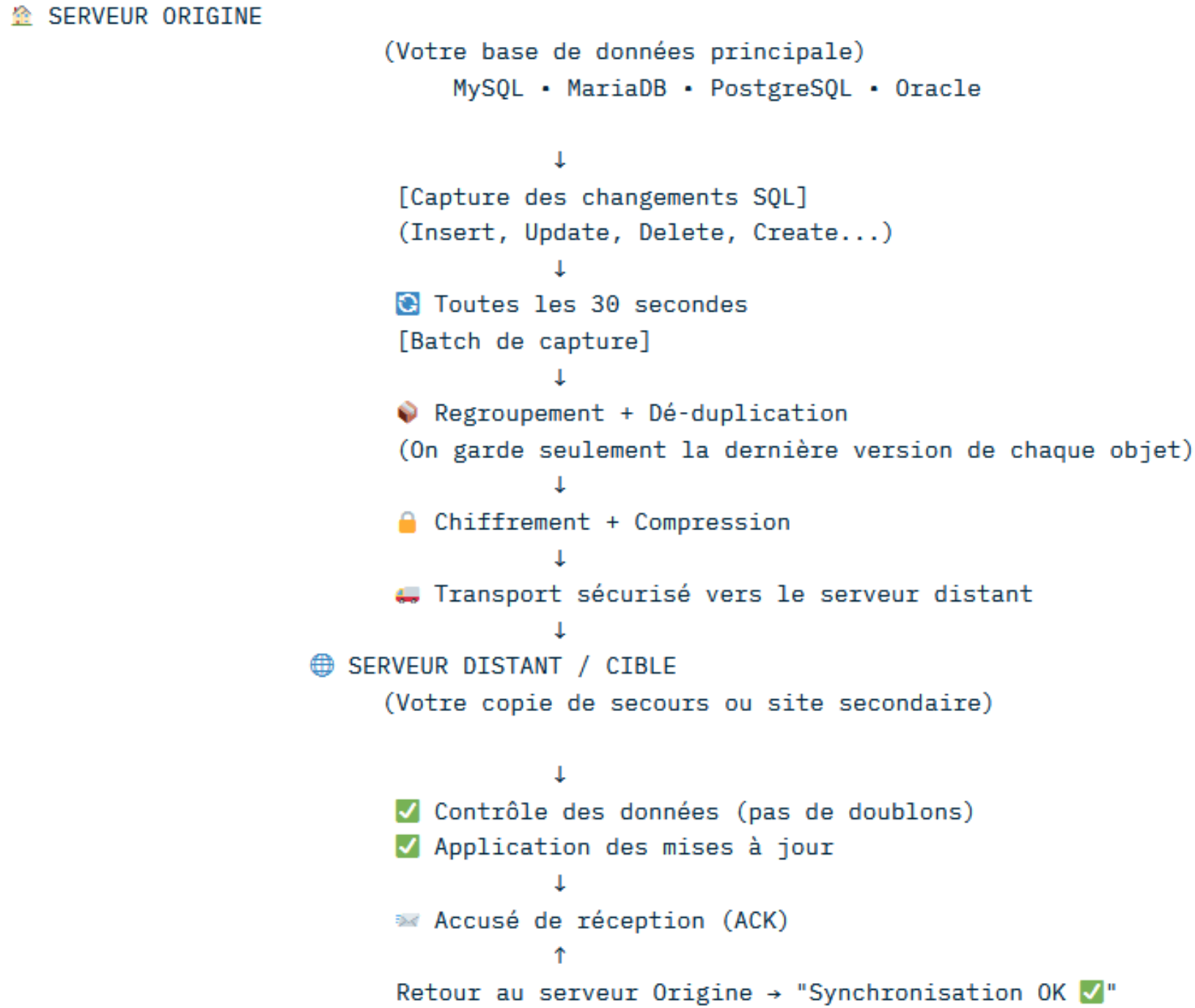
- users/groups techniques
- firewall
- Python venv / dépendances applicatives
- secrets via coffre
- replay plus rapide
- bascule semi-automatique
- tests de PRA

V3

Puis :

- multi-nodes
- dépendances inter-serveurs
- orchestration de rôle
- clusters applicatifs
- promotion coordonnée
- règles métier de bascule


SRDF : Synchronisation Intelligente
et Sécurisée de vos Bases de Données
SQL
Toutes les
30
secondes
Contrôle
OK
MySQL
MariaDB
PostgreSQL
ORACLE
Insert
Update
Delete
Create
Serveur Origine
Compression
SQL
Aplication
Serveur Distant / Cible


🏠 SERVEUR ORIGINE
(Votre base de données principale)
MySQL • MariaDB • PostgreSQL • Oracle
↓
[Capture des changements SQL]
(Insert, Update, Delete, Create...)
↓
🔁 Toutes les 30 secondes
[Batch de capture]
↓
📦 Regroupement + Dé-duplication
(On garde seulement la dernière version de chaque objet)
↓
🔒 Chiffrement + Compression
↓
🚚 Transport sécurisé vers le serveur distant
↓
🌐 SERVEUR DISTANT / CIBLE
(Votre copie de secours ou site secondaire)
↓
✅ Contrôle des données (pas de doublons)
✅ Application des mises à jour
↓
📨 Accusé de réception (ACK)
↑
Retour au serveur Origine → "Synchronisation OK ✅"

Un résumé ultra-simple

Imaginez que votre base de données principale (à gauche) est comme une **cuisine centrale** où toutes les commandes arrivent.

Dès qu'un plat (une modification) est préparé, notre système le **capture**, le met dans une **boîte bien organisée** (paquet), le rend plus léger et sécurisé, puis l'envoie rapidement à votre **deuxième cuisine** (serveur distant).

Une fois arrivé, on vérifie que tout est bon, on le sert, et on renvoie un petit mot : « Tout est bien reçu ! ».

Tout cela se fait **automatiquement toutes les 30 secondes**, sans doublons, sans erreurs, et en toute sécurité.

Pourquoi SRDF est largement supérieur aux solutions classiques ?

Contrairement aux outils traditionnels (réplication native MySQL/PostgreSQL ou solutions lourdes type Oracle GoldenGate), SRDF offre une **synchronisation intelligente toutes les 30 secondes** avec un contrôle total : groupement intelligent, dé-duplication automatique, chiffrement et compression intégrés.

Il consomme très peu de ressources, évite les doublons et les mises à jour inutiles, et gère les erreurs avec reprise automatique.

Résultat : une réplication **fiable, sécurisée, ultra-rapide et légère**, idéale pour les startups qui veulent une haute disponibilité et une reprise après sinistre sans complexité ni coût exorbitant.

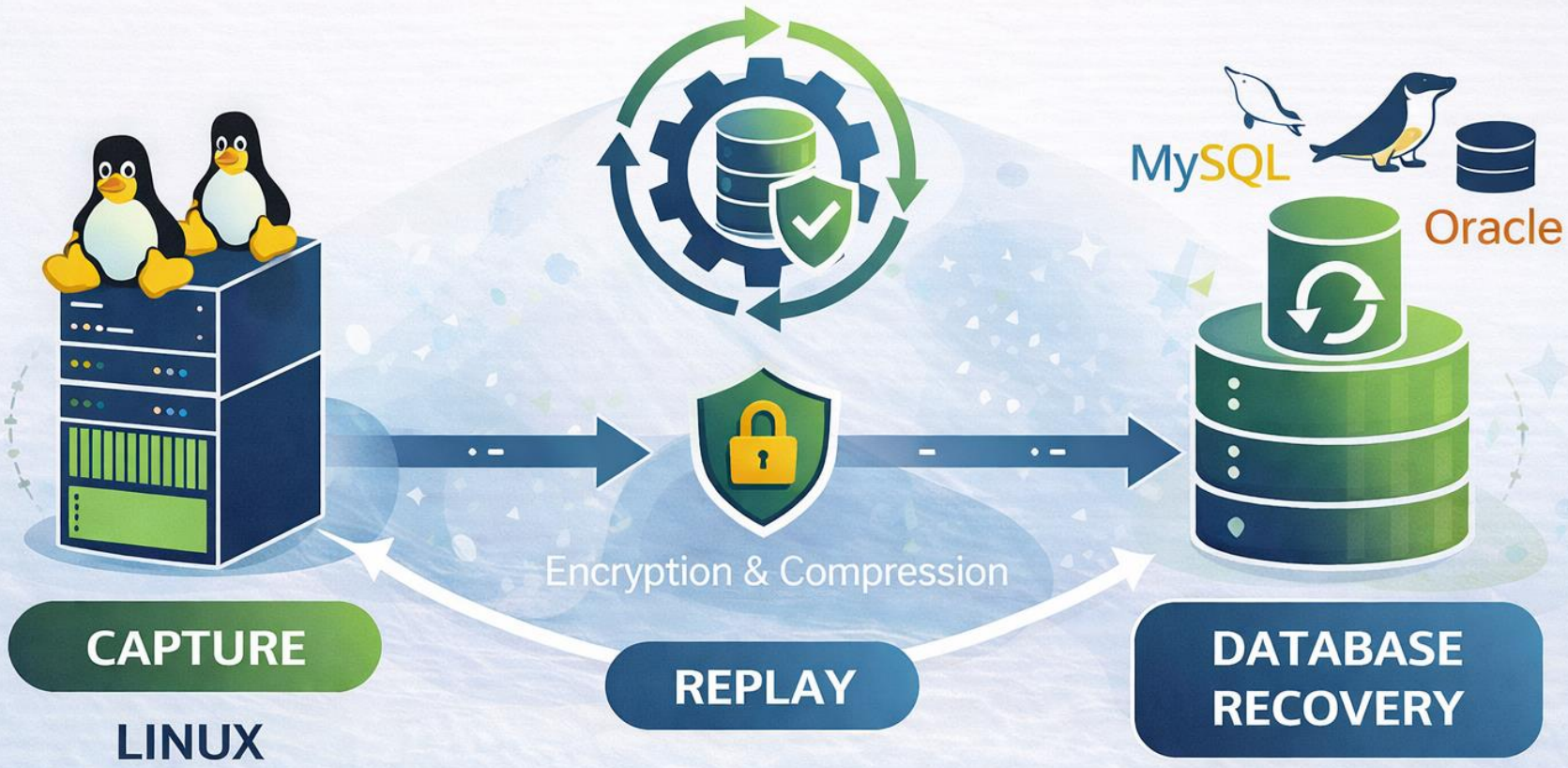
SRDF transforme une contrainte technique en **avantage compétitif** : vos données sont toujours à jour, partout, en toute sécurité, sans ralentir vos applications.

Solution	⚡ Points forts	⚠️ Limites	🎯 Cible
SRDF	• Batch optimisé + dédup • Compression + chiffrement • Faible consommation • Reprise automatique	• Moins mature	Startups / scale-ups solution moderne & optimisée
Native	• Gratuit / intégré • Latence temps réel • Setup simple	• Pas de dédup • Pas de compression native • Consomme bande passante • Gestion manuelle erreurs	Environnements simples budget limité
GoldenGate	• Très mature • Latence très faible • Multi-DB robuste	• Très cher • Complexe • Ressources élevées	Grandes entreprises Oracle-centric
AWS DMS	• Managed AWS • Facile à déployer • Multi-sources	• Latence variable • Coûts évolutifs • Lock-in AWS	Projets AWS migration simple

SRDF 2026 DOIT ETRE

- SIMPLE
- EFFICACE
- RAPIDE
- FIABLE
- SECURE
- EVOLUTIF
- ABORDABLE

SRDF IDEO-LAB : GENERAL FLOW



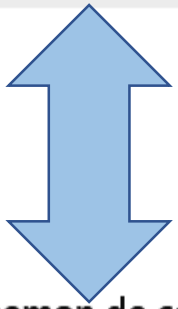
SRDF COMMUNICATIONS

SRDF C'EST 4 BRIQUES DISTINCTES

1. Orchestrator Local

Sur le serveur source, il pilote :

- capture
- dedup
- build wagon / batch
- envoi du wagon vers le remote
- attente ACK / confirmation d'application
- mise à jour des `OutboundChangeEvent`



3. Mini serveur / daemon de communication Local

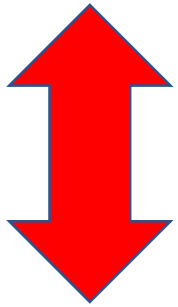
Côté source, exposé en API, pour recevoir les callbacks du remote :

- ACK batch reçu
- APPLY batch terminé
- APPLY batch failed

2. Orchestrator Remote

Sur le serveur target, il pilote :

- réception du wagon
- enregistrement dans `InboundChangeEvent`
- apply SQL sur la database target
- marquage des inbound en `applied` / `failed`
- retour d'état vers le source

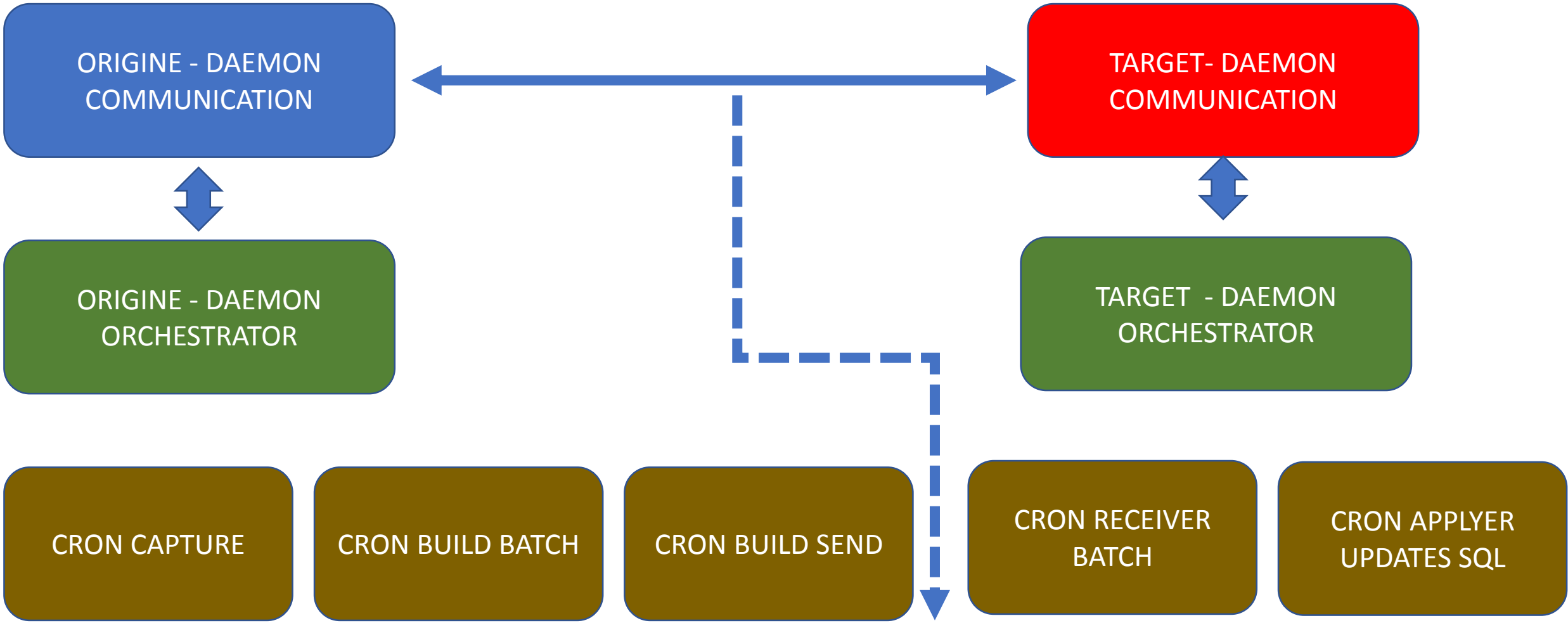


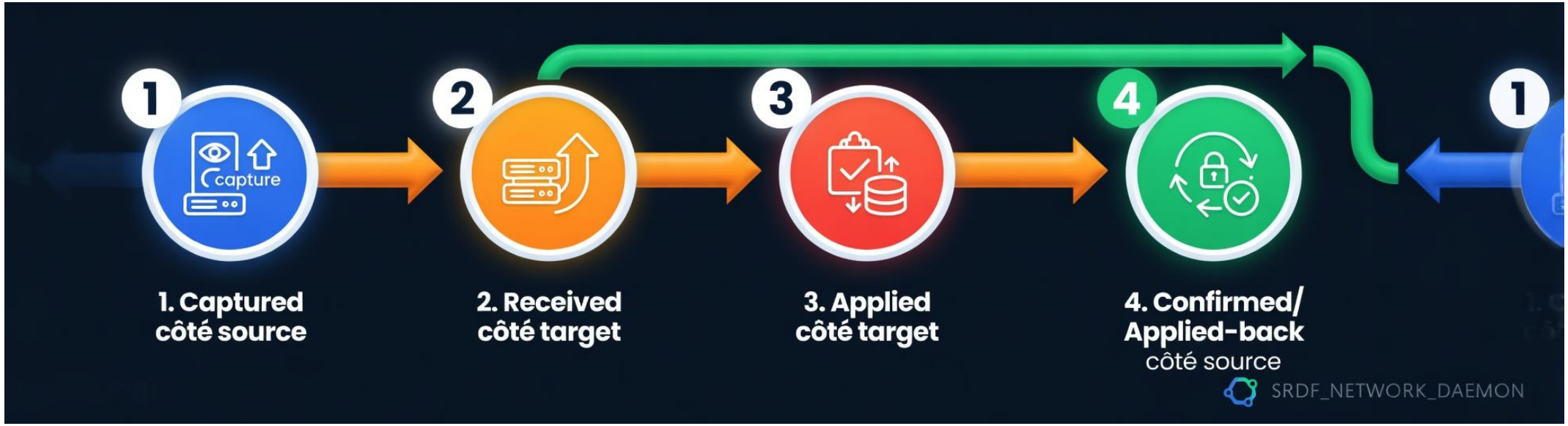
4. Mini serveur / daemon de communication Remote

Côté target, exposé en API, pour recevoir les demandes du source :

- `receive_wagon`
- éventuellement `apply_batch`
- éventuellement `health/ping`

ARCHITECTURE DE COMMUNICATION




1
capture
2
3
4
1
1. Captured
côté source
2. Received
côté target
3. Applied
côté target
4. Confirmed/
Applied-back
côté source
SRDF_NETWORK_DAEMON

RAPPEL
PRINCIPE
ARCHITECTURE
GLOBAL

Les 4 états métier à distinguer

Tu veux implicitement 4 niveaux, et c'est très bien :

1. **captured** côté source
2. **received** côté target
3. **applied** côté target
4. **confirmed/applied-back** côté source

C'est seulement au niveau 4 qu'on peut dire :

- "ce wagon est vraiment répliqué"

Le vrai flux correct

Côté source

1. capture
2. dedup
3. build d'un wagon
4. appel API vers le **daemon remote**
5. le wagon passe en `sent`
6. attente du retour du remote
7. quand le remote dit "applied", alors seulement :
   - batch `acked/applied`
   - outbound `acked/applied`

Côté target

1. le daemon remote reçoit le wagon
2. il l'enregistre dans la table `Inbound`
3. il déclenche l'**orchestrator remote**
4. l'orchestrator remote applique les updates SQL
5. il met à jour `InboundChangeEvent`
6. il appelle le **daemon local** pour dire :
   - batch reçu
   - batch appliqué
   - ou batch en erreur

Donc le modèle cible devient

Serveur origine

- `OutboundChangeEvent`
- `TransportBatch`
- `SRDFOrchestrator` local
- `CommunicationDaemon` local

Serveur target

- `InboundChangeEvent`
- `SRDFRemoteOrchestrator`
- `CommunicationDaemon` remote
- applier SQL local

Donc oui, je reformule

- un Orchestrator Local
- un Orchestrator Remote
- un mini serveur de communication Local
- un mini serveur de communication Remote
- un protocole bidirectionnel
- et des statuts cohérents source/target

La cible d'architecture

Orchestrator Local

Responsable de :

1. capture
2. dedup
3. build batch
4. send wagon vers target
5. attendre retour remote
6. mettre à jour `OutboundChangeEvent`

Orchestrator Remote

Responsable de :

1. recevoir wagon
2. créer `InboundChangeEvent`
3. appliquer les updates SQL sur la DB target
4. marquer les inbound `applied` / `failed`
5. notifier le source

Daemon/API Local

Endpoints pour recevoir du remote :

- `batch_received`
- `batch_applied`
- `batch_failed`

Daemon/API Remote

Endpoints pour recevoir du source :

- `receive_batch`
- éventuellement `trigger_apply`
- `health`

Le flux complet voulu

Sens Local → Remote

1. Local orchestrator lance capture
2. Local orchestrator lance dedup
3. Local orchestrator lance build batch
4. Local sender appelle le daemon remote avec :
   - `batch_uid`
   - payload
   - checksum

Sur le target

5. Remote daemon reçoit le wagon
6. appelle `transport_receiver.py`
7. crée les `InboundChangeEvent`
8. déclenche l'orchestrateur remote
9. l'orchestrateur remote lance `transport_applier.py`
10. les inbound passent en `applied` ou `failed`

Sens Remote → Local

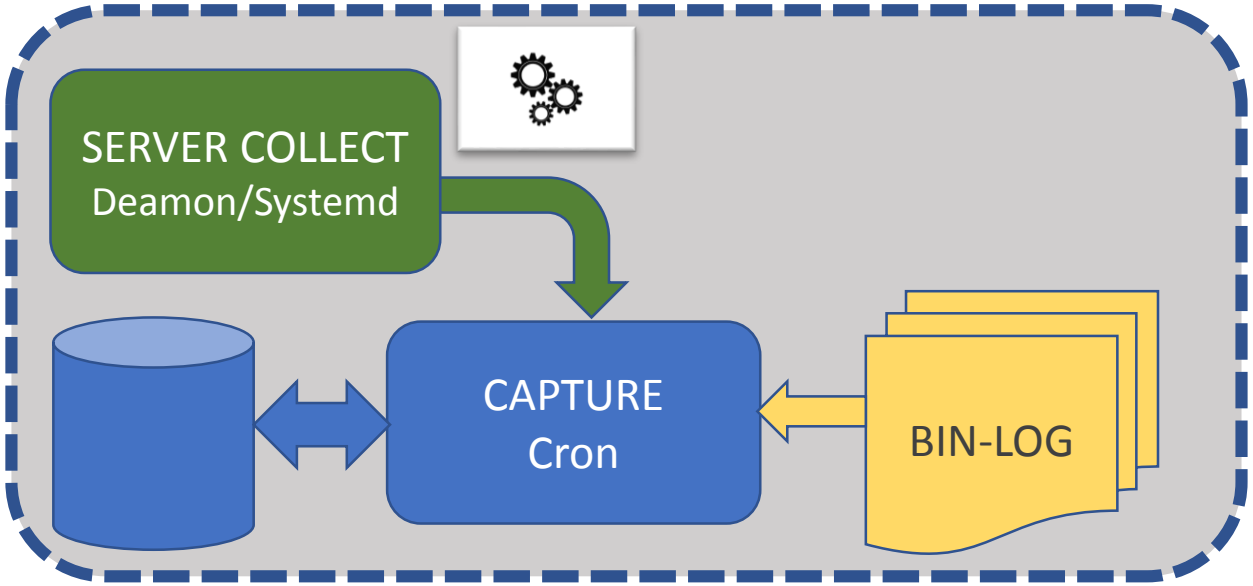
11. Remote daemon appelle le daemon local
12. local met à jour :
    - `TransportBatch`
    - `OutboundChangeEvent`
    - statuts finaux source

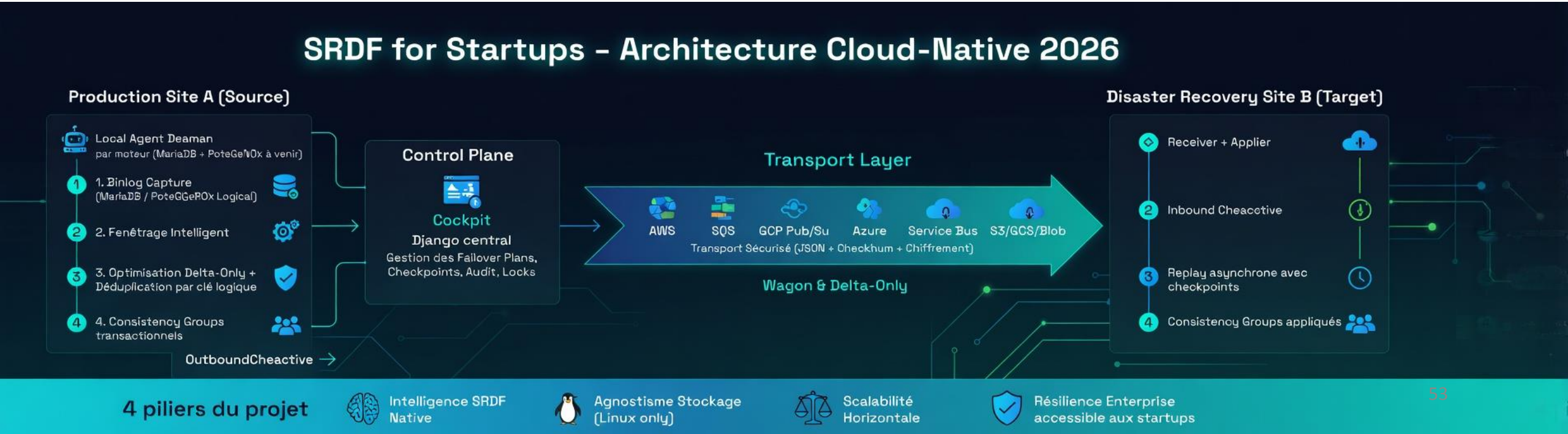
SRDF OBJECTIFS & FLOW

OBJECTIFS POURSUIVIS

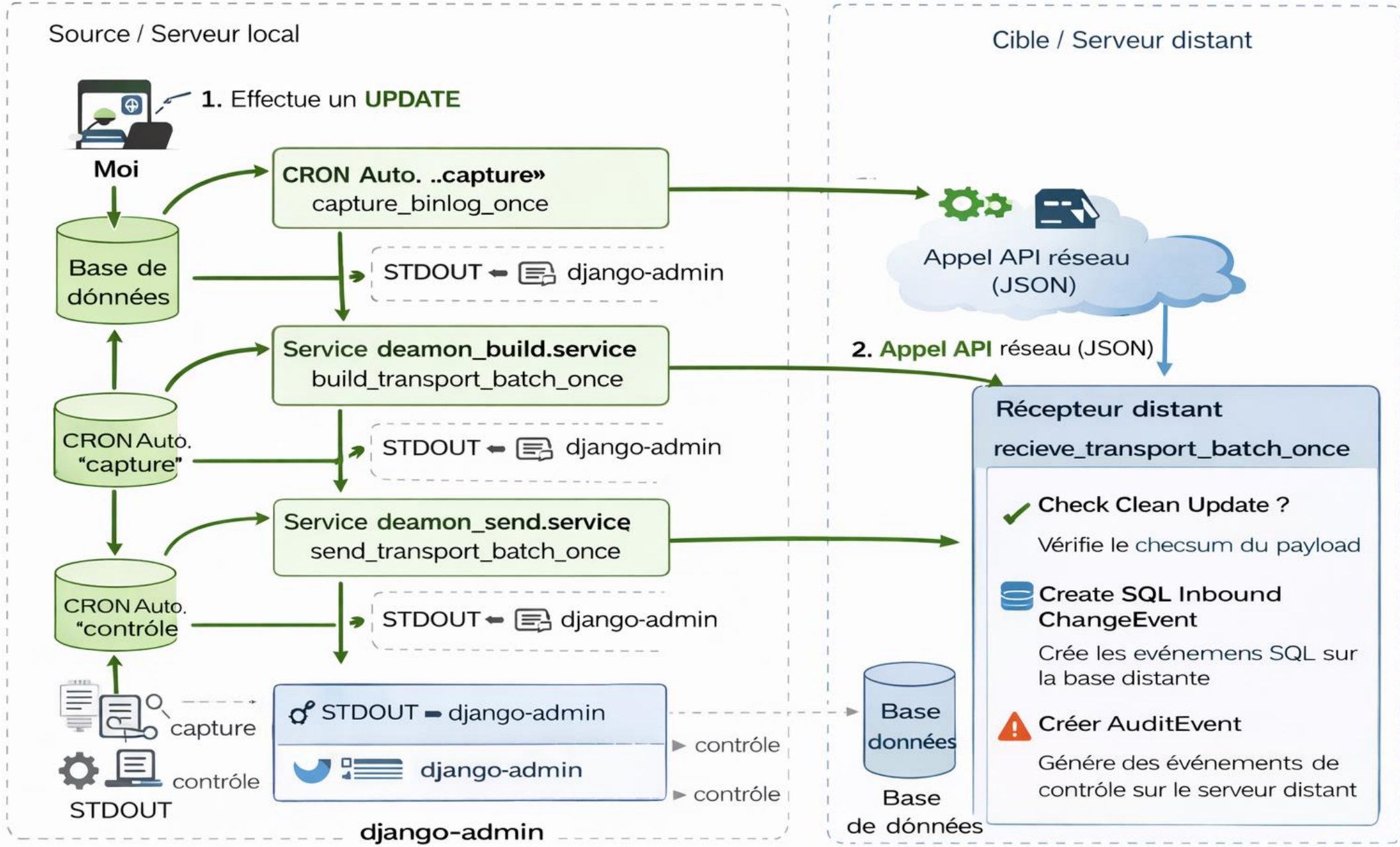
- Mise à jour « SQL » sur la Base de donnée (Origine) , *Insert, Update, Delete, Create, ..*
- Réveil du Daemon toutes les 30 secondes afin de lancer le « Batch de capture »
- Batch de capture des dernières mises à jour « non synchronisées» sur Cible
- Groupement de ces mises à jour par « paquets consistents » seuil à définir..
- De-duplication de ces Maj « On ne garde que la dernière » sur le même objet.
- Vérification si un transport est disponible (Check du serveur distant), Busy ??
- Encryptage des données - Compression des données
- Transport du paquet de « mises à jour » vers le serveur distant.
- Réception du paquet de mises à jour sur le serveur distant
- Contrôle des données à mettre à jour (***ne pas refaire un update déjà réalisé***)
- **Controle des erreurs** et retour à l'envoyeur !!
- Préparation du message de retour et envoie « accusé réception » vers le serveur Origine
- Le serveur origine, met à jour la table des captures « **synchro » OK....** »

SRDF : GENERAL FLOW

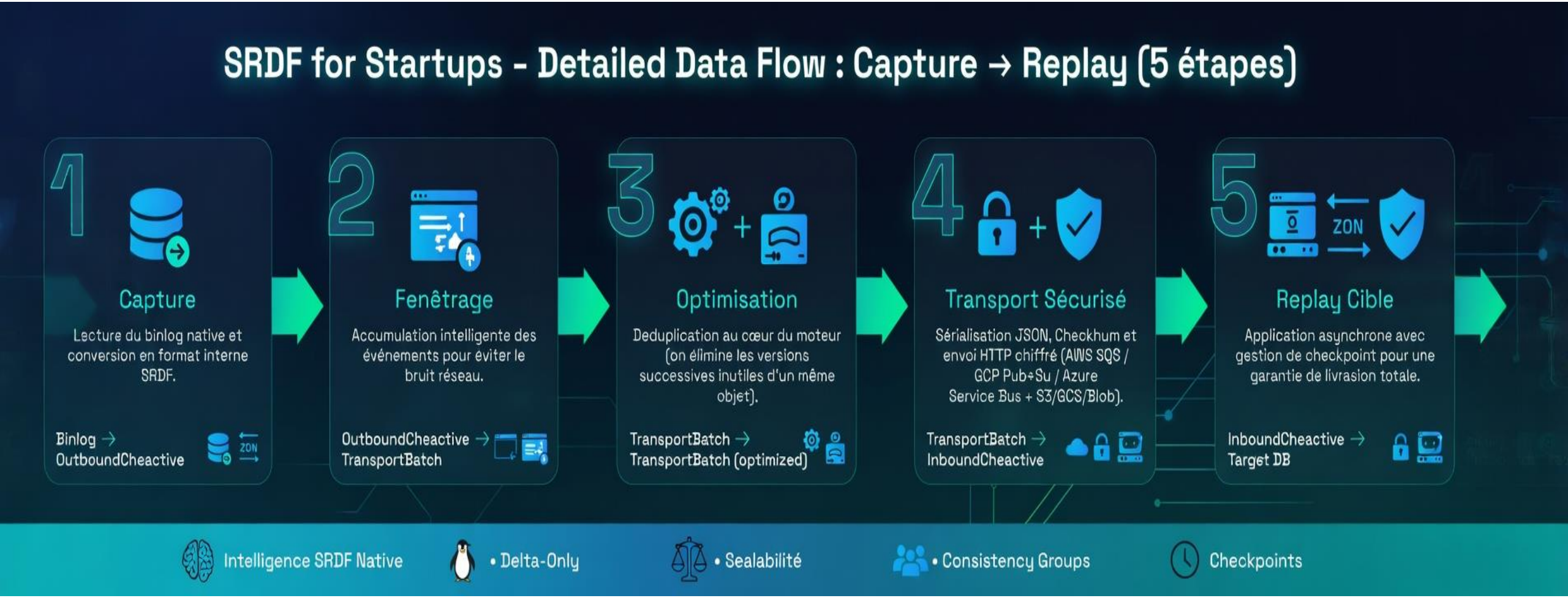




Flux complet du traitement d'un UPDATE local jusque sur le serveur distant



DETAILED DATA FLOW



BRIQUES LOGICIELLES ENVISAGEES

Niveau 1 — briques unitaires

Déjà faites :

- capture
- build batch
- send batch
- receive batch

Niveau 2 — orchestrateur permanent

Pas encore fait :

- daemon
- boucle continue
- systemd
- wakeup interval
- scheduling interne
- supervision complète

AGENT / DAEMON

- gestion des erreurs
- logs
- lock d'exécution
- backoff
- heartbeat
- config par service
- mode dry-run
- mode debug

BOUCLE INTERNE – SERVICE SYSTEMD

Sur le source

service systemd

```
srdf-source-agent.service
```

boucle interne

- wake up toutes les N secondes
- capture binlog
- construit un ou plusieurs batchs
- envoie les batchs
- loggue
- dort

Sur la cible

service systemd

```
srdf-target-agent.service
```

ou plus simplement :

- Django + API déjà exposée
- receiver appelé via `/api/action/`
- plus tard applier SQL en fond

SRDF : CONCEPT ET PRÉSENTATION

CONCEPT

PRÉSENTATION

Le Pitch : La résilience “Enterprise” enfin accessible

Il y a 20 ans – EMC SRDF
(propriétaire & hors de prix)

Aujourd’hui – SRDF for Startups
(scalable, abordable & simple)

L’héritage de la résilience Enterprise, désormais accessible aux startups et infrastructures modernes.

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**.

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.

Agnostisme 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.

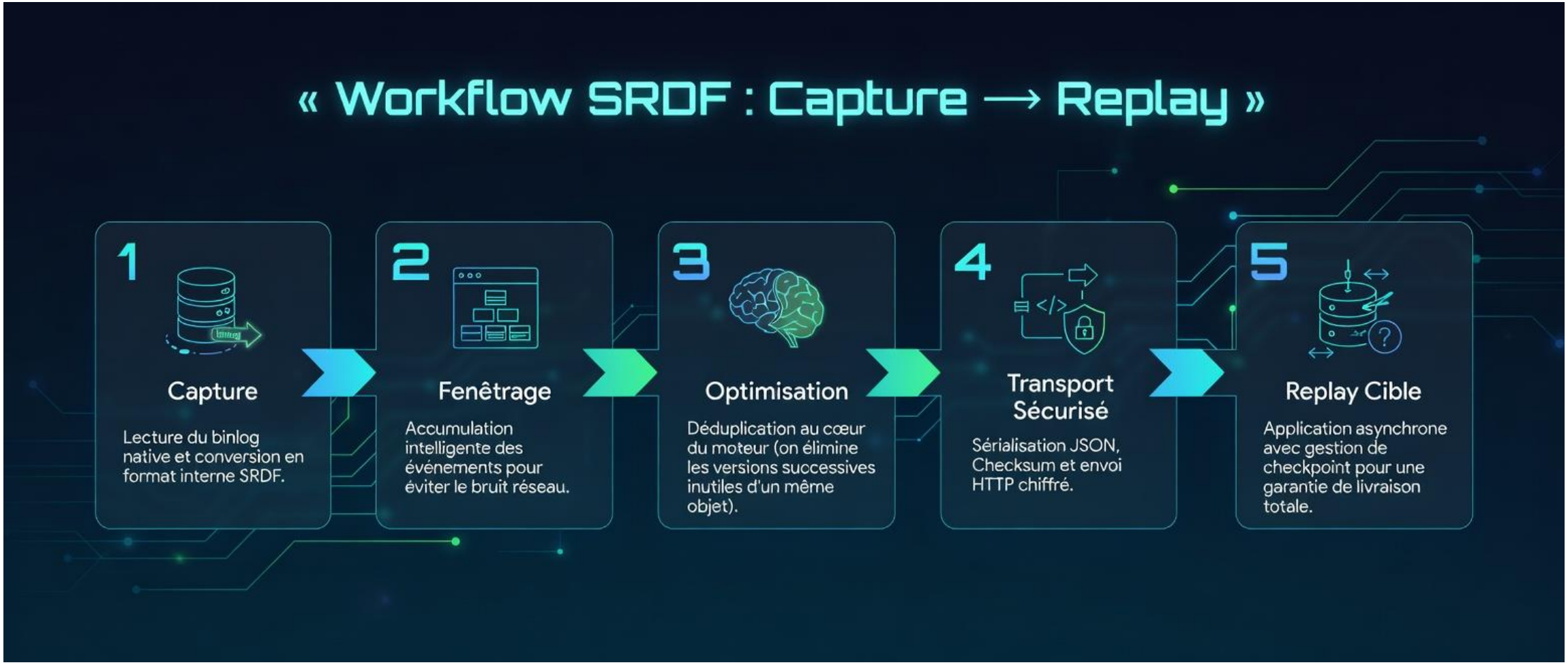
Scalabilité Horizontale

Conçu pour encaisser les charges des startups en hyper-croissance, avec une gestion transparente des files d'attente (Inbound/Outbound).

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).


« Workflow SRDF : Capture ⟶ Replay »
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.

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
Cloûts 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"

📈 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

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

INTELLIGENCE

DONNÉE

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

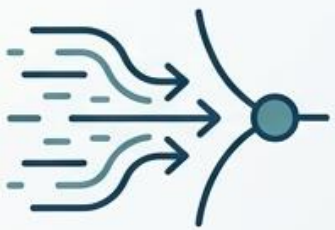
Le projet repose sur une architecture de **Data Stream Management** qui priorise la cohérence métier et l'économie de ressources. Contrairement aux solutions de réplication classiques qui saturent le réseau, notre approche est **sélective et optimisée**.

Principes de Fonctionnement - SRDF for Startups

1

CAPTURE

Lecture Native du Binlog
(Zero Impact Source)



Capture Asynchrone
(Binlog Reader)

2

FENÊTRAGE
(CONSISTENCY GROUP)

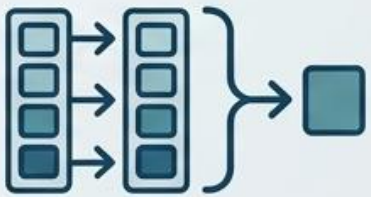
Accumulation Intelligente
dans des Wagons

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

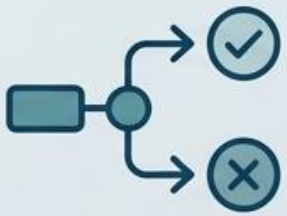
Compression, Chiffrement,
Transport Optimisé

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".

SRDF : CAPTURE DES UPDATES À PARTIR DE LA LOG DU MOTEUR SQL

TRANSACTION LOG

CAPTURE SRDF

LA PHASE DE « CAPTURE »

La phase de **Capture** est le point d'entrée critique du système. Son rôle est d'extraire les modifications de données de la source sans jamais perturber les performances de l'application de production.

Voici le détail technique de cette étape, conçue pour garantir un impact quasi nul sur la base de données principale (Zero Impact) :

SRDF CAPTURE

1. Le principe du "Log-Based CDC«

2. Le Binlog Reader (Lecture Asynchrone)

3. Parsing et Abstraction

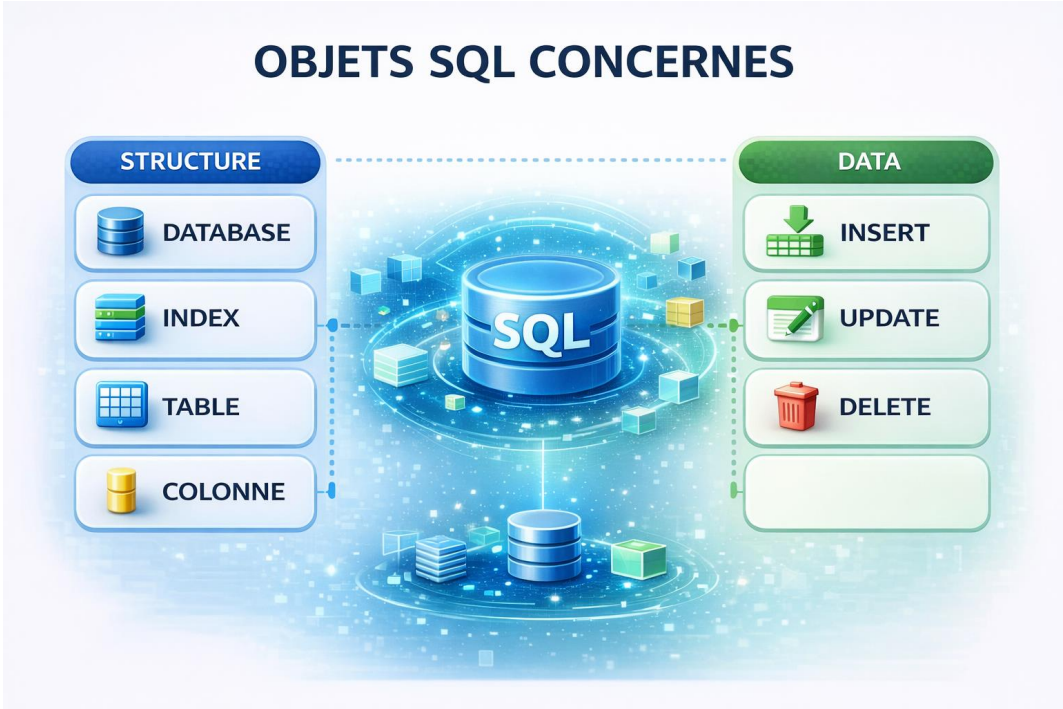
4. Conversion au Format Interne SRDF

5. Marquage Temporel et Séquençage

0 – OBJETS SQL CONCERNES"

- STRUCTURE :
  - DATABASE
  - INDEX
  - TABLE
  - COLONNE

- DATA :
  - INSERT
  - UPDATE
  - DELETE



0 – OBJETS SQL CONCERNES

- STRUCTURE :
  - DATABASE
  - INDEX
  - TABLE
  - COLONNE

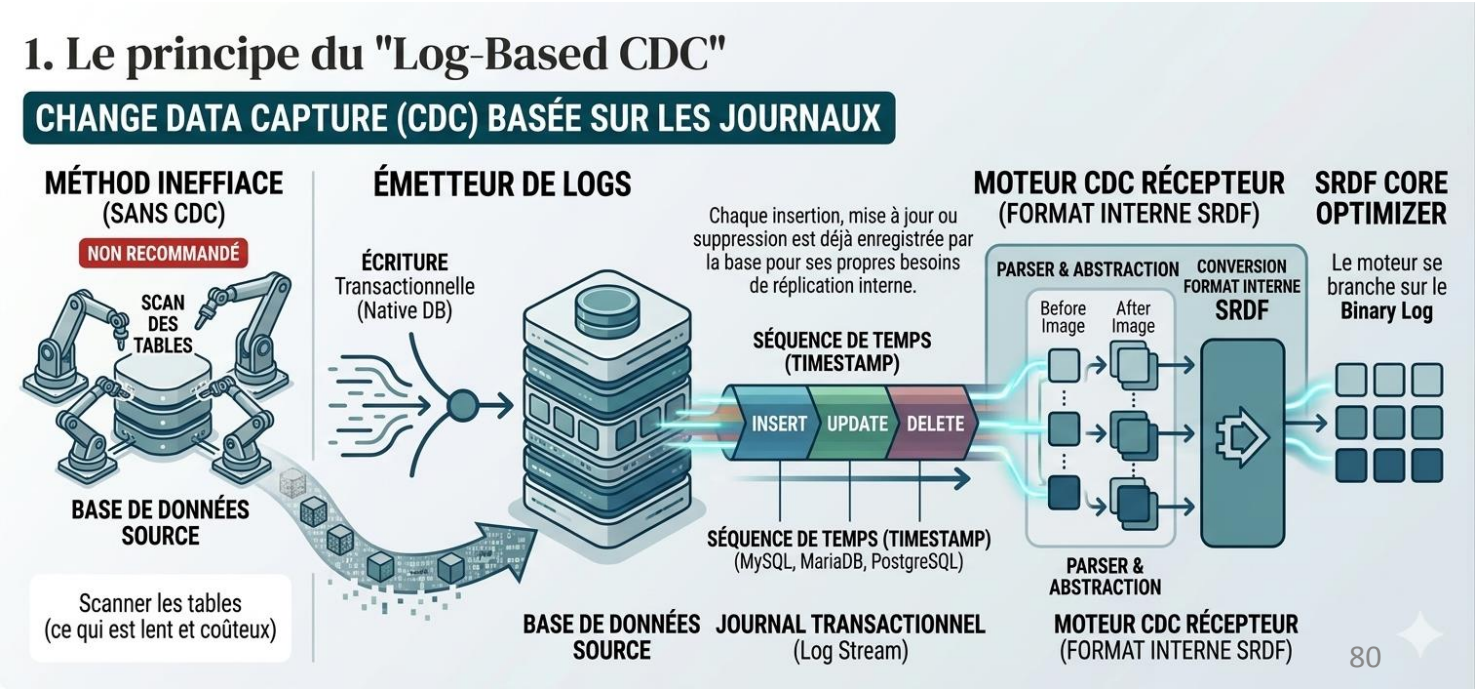
- DATA :
  - INSERT
  - UPDATE
  - DELETE

1. Le principe du "Log-Based CDC"

1. Le principe du "Log-Based CDC"

Au lieu de scanner les tables (ce qui est lent et coûteux), nous utilisons la technique du **Change Data Capture (CDC)** basée sur les journaux.

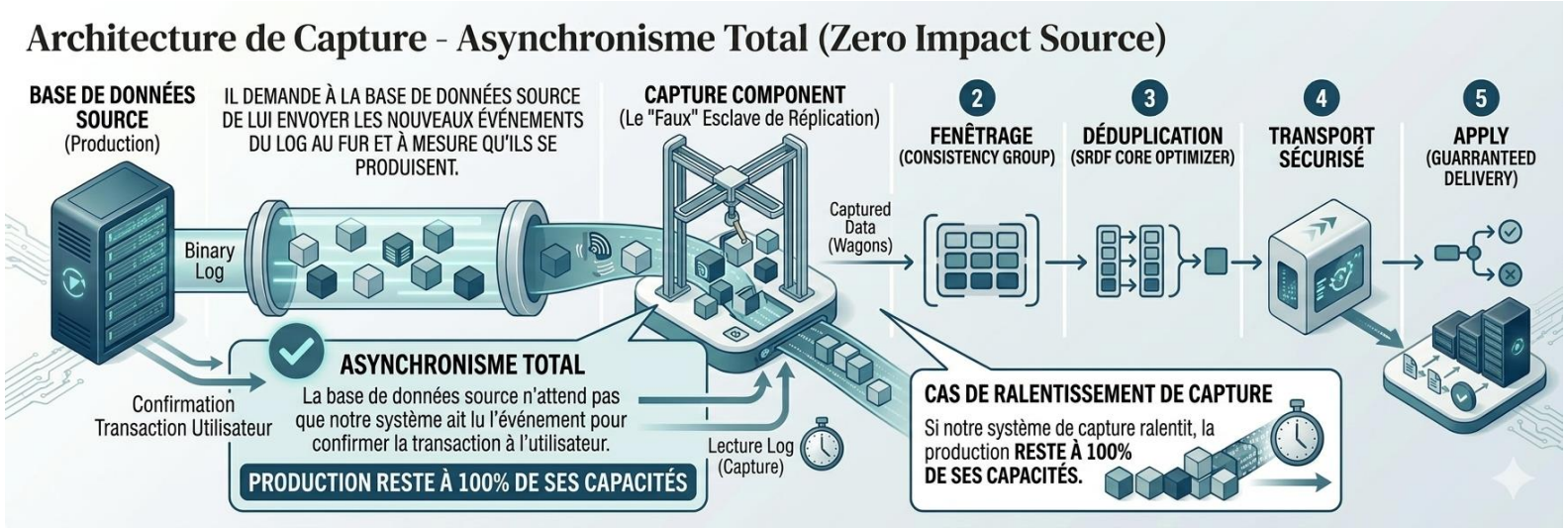
- **Source :** Le moteur se branche sur le **Binary Log** (binlog) de la base de données (MySQL, MariaDB, PostgreSQL).
- **Nature :** C'est un flux séquentiel où chaque insertion, mise à jour ou suppression est déjà enregistrée par la base pour ses propres besoins de réplication interne.



2. Le Binlog Reader (Lecture Asynchrone)

Le composant de capture agit comme un "faux" esclave de réplication.

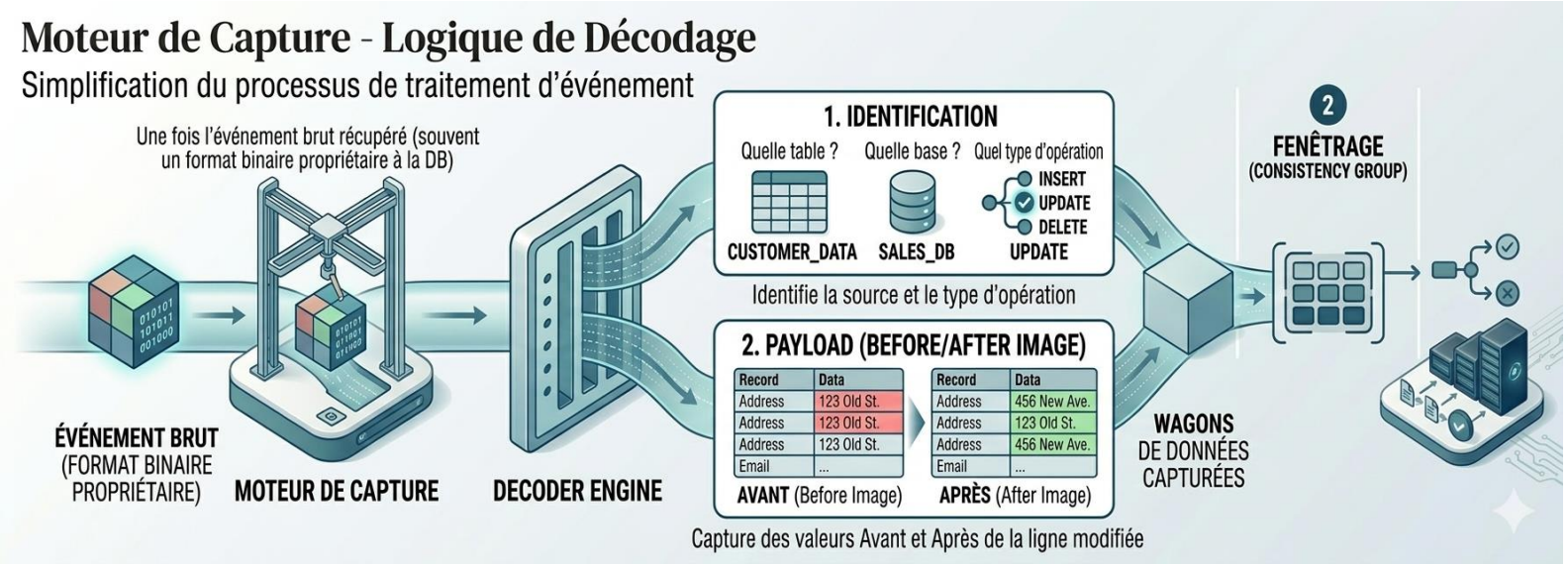
- Il demande à la base de données source de lui envoyer les nouveaux événements du log au fur et à mesure qu'ils se produisent.
- **Asynchronisme total :** La base de données source n'attend pas que notre système ait lu l'événement pour confirmer la transaction à l'utilisateur. Si notre système de capture ralentit, la production reste à 100% de ses capacités.



3. Parsing et Abstraction

Une fois l'événement brut récupéré (souvent un format binaire propriétaire à la DB), le moteur de capture le décode :

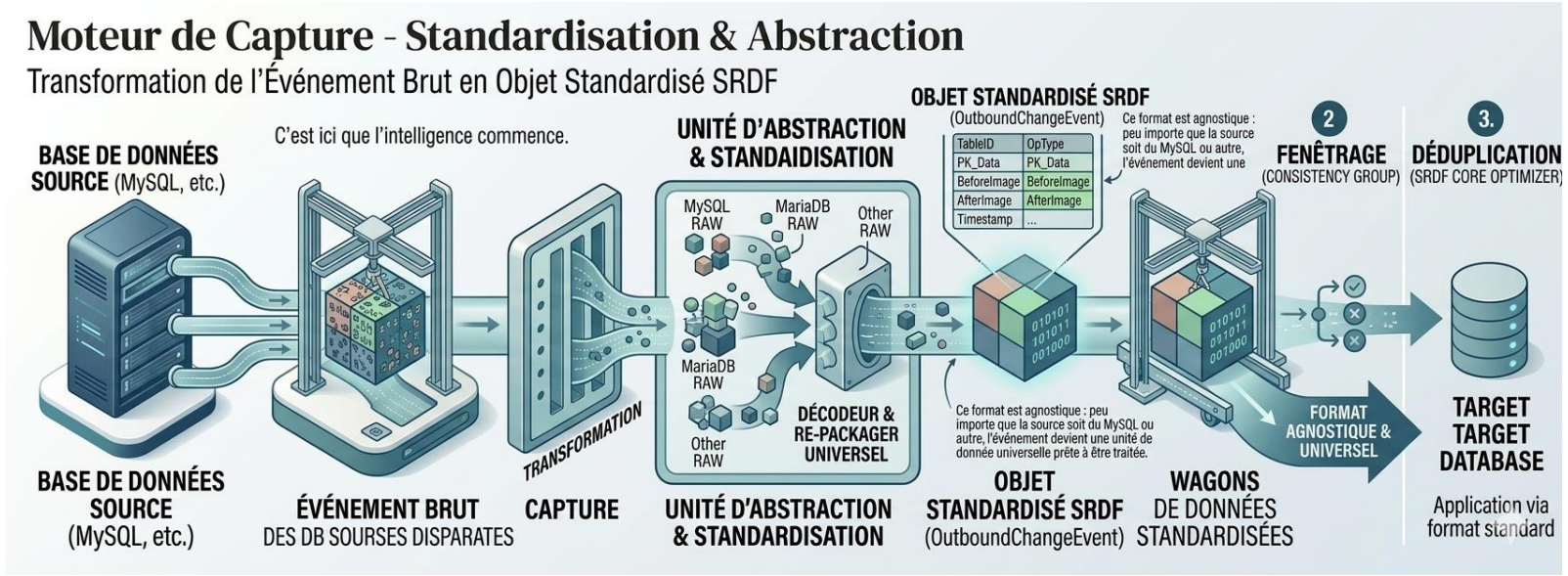
- **Identification :** Quelle table ? Quelle base ? Quel type d'opération (INSERT/UPDATE/DELETE) ?
- **Payload :** Capture des valeurs "Avant" (Before Image) et "Après" (After Image) de la ligne modifiée.



4. Conversion au Format Interne SRDF

C'est ici que l'intelligence commence. L'événement brut est transformé en un **objet standardisé SRDF** (OutboundChangeEvent).

- Ce format est agnostique : peu importe que la source soit du MySQL ou autre, l'événement devient une unité de donnée universelle prête à être traitée par les phases suivantes (Fenêtrage et Déduplication).

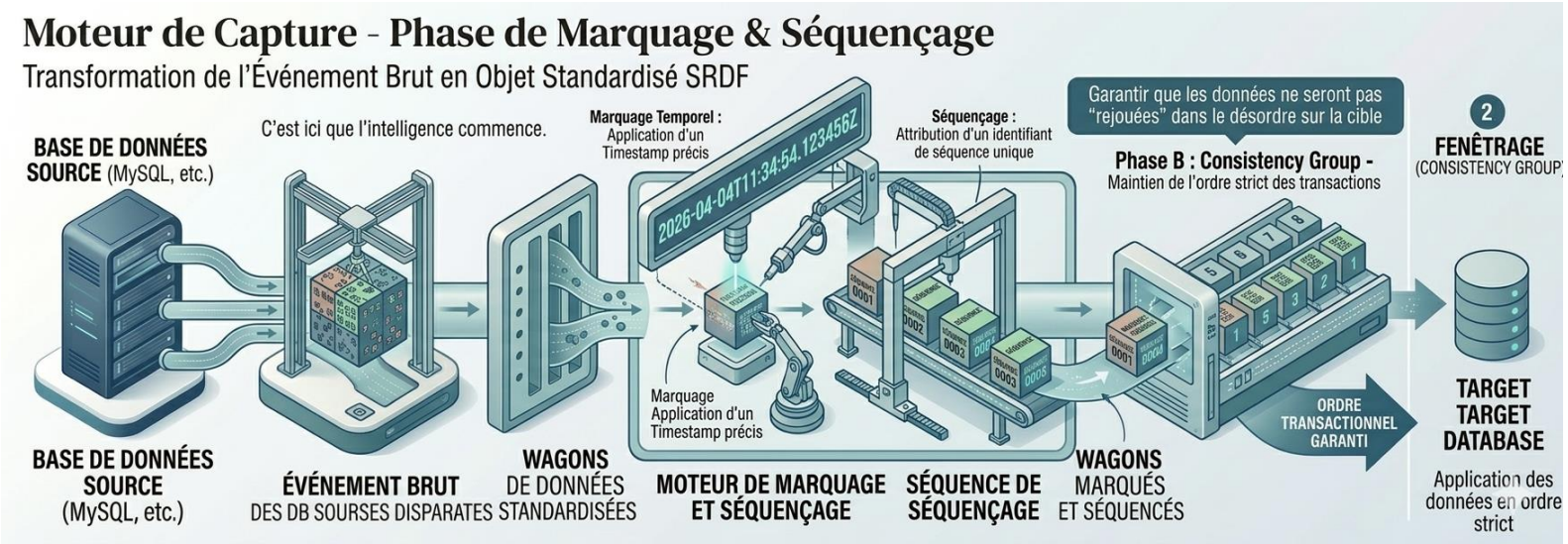


5. Marquage Temporel et Séquençage

5. Marquage Temporel et Séquençage

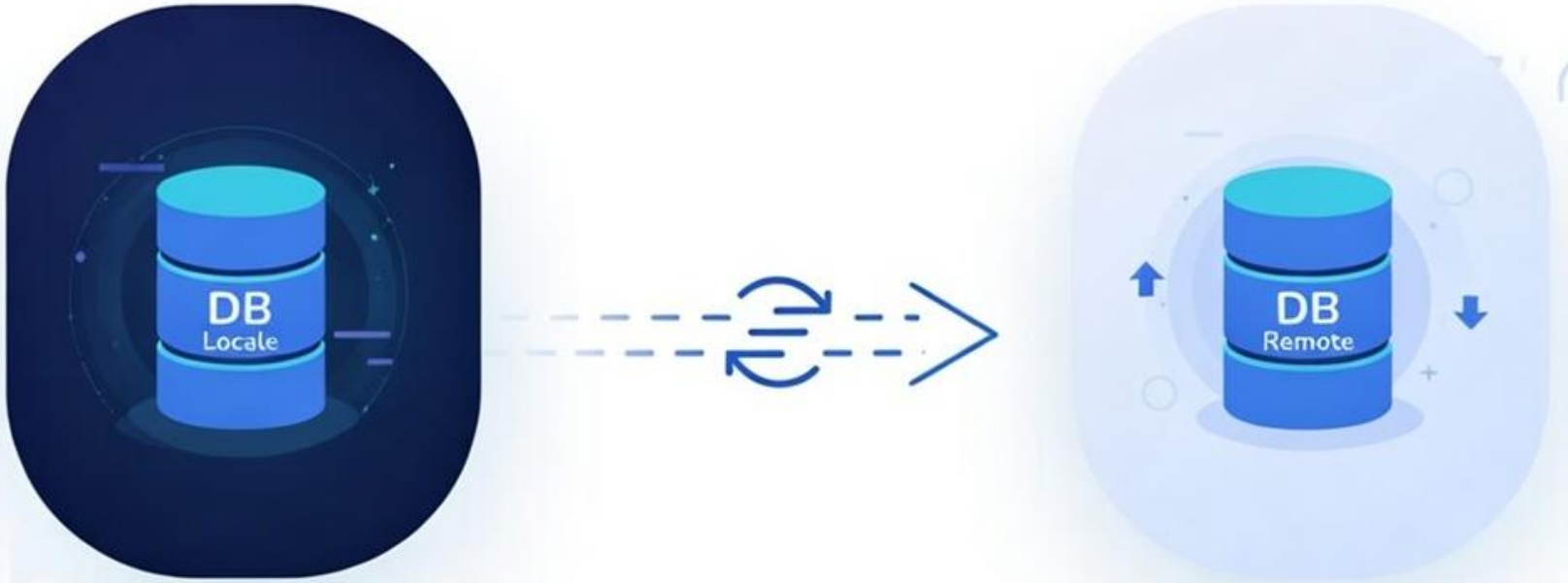
Chaque événement capturé reçoit un **Timestamp précis** et un identifiant de séquence.

- Cela permet de maintenir l'ordre strict des transactions, ce qui est indispensable pour la phase de **Consistency Group** (Phase B) afin de garantir que les données ne seront pas "rejouées" dans le désordre sur la cible.



PRINCIPES DE BASE DE SRDF-DB

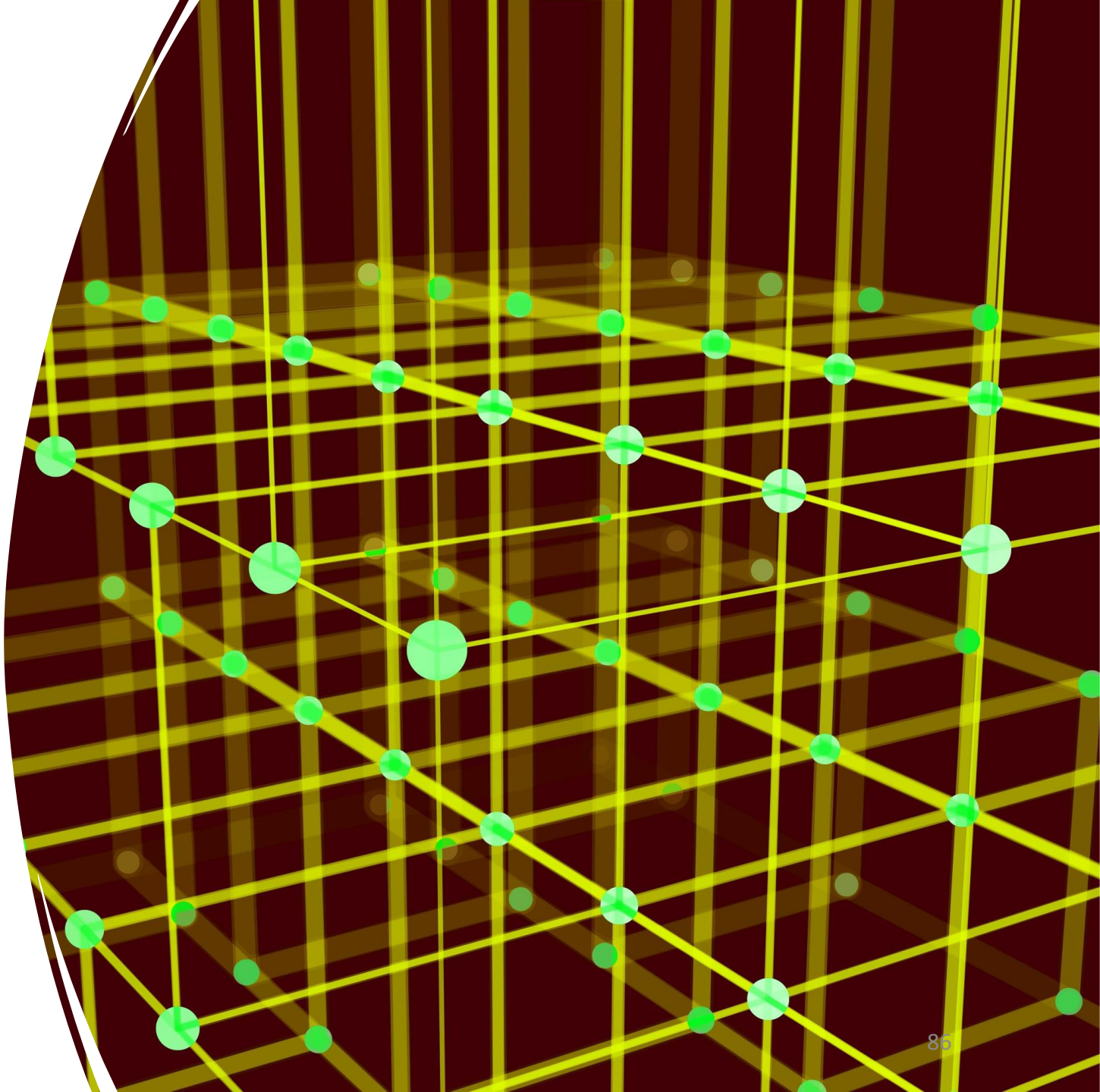
(Replica DB Locale → DB Remote)



DB LOCALE

DB REMOTE

PRINCIPES DE BASE DE SRDF-DB



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*)

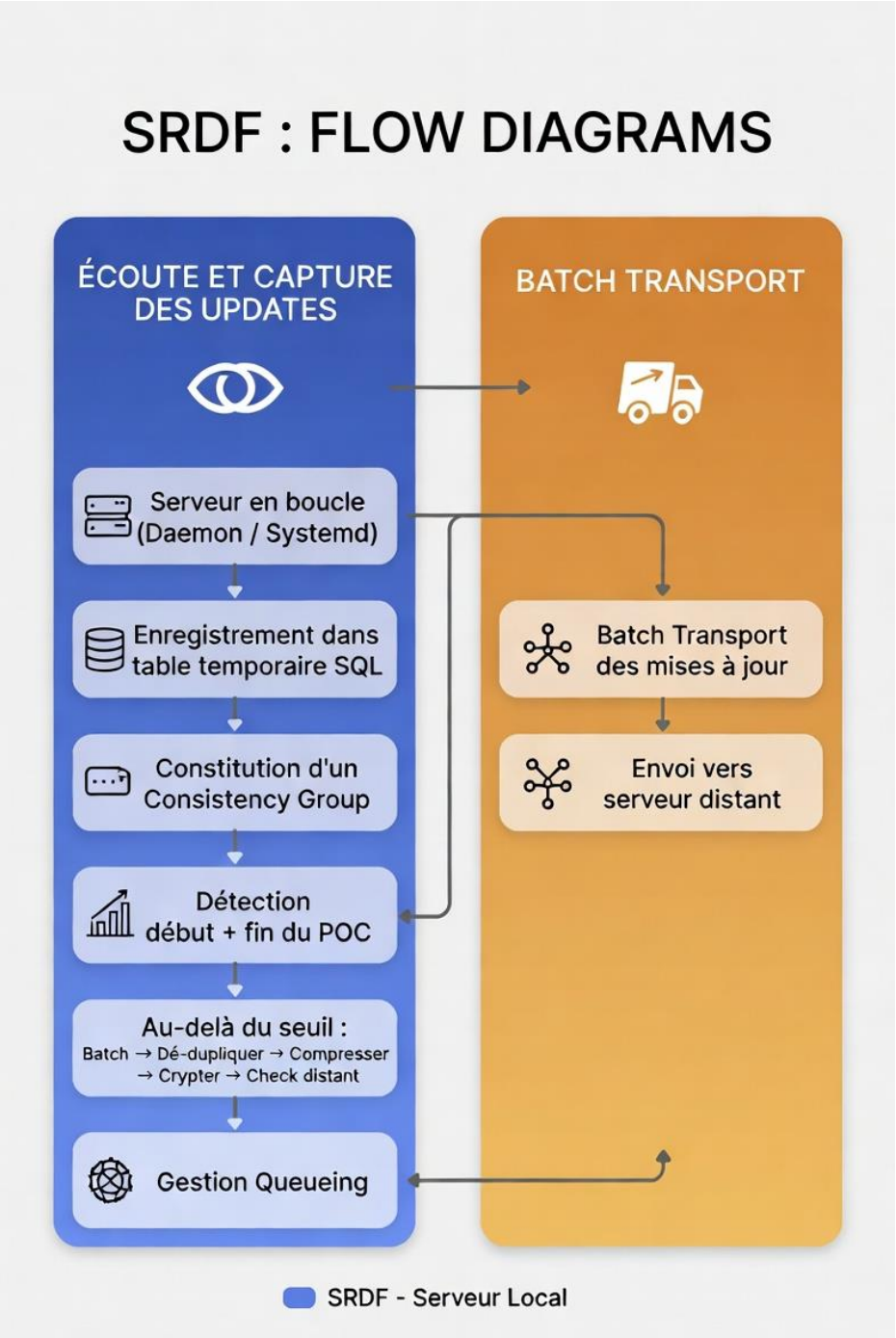

SYNCHRONISATION
DB REMOTE
1. Agent Remote
• Écoute du canal
• Vérification intégrité
2. Réception & Stockage
• Stockage dans table SQL réception
• ID unique
Retry
3. File d'attente & Exécution
• Mise en queue
Celer y
• Exécution sur DB Target
Logging
4. Accusé & Retour
• Stockage résultat
• Renvoi accusé de réception
Remote → Origine (MariaDB Cible)

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

PLANNING

DÉVELOPPEMENT

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

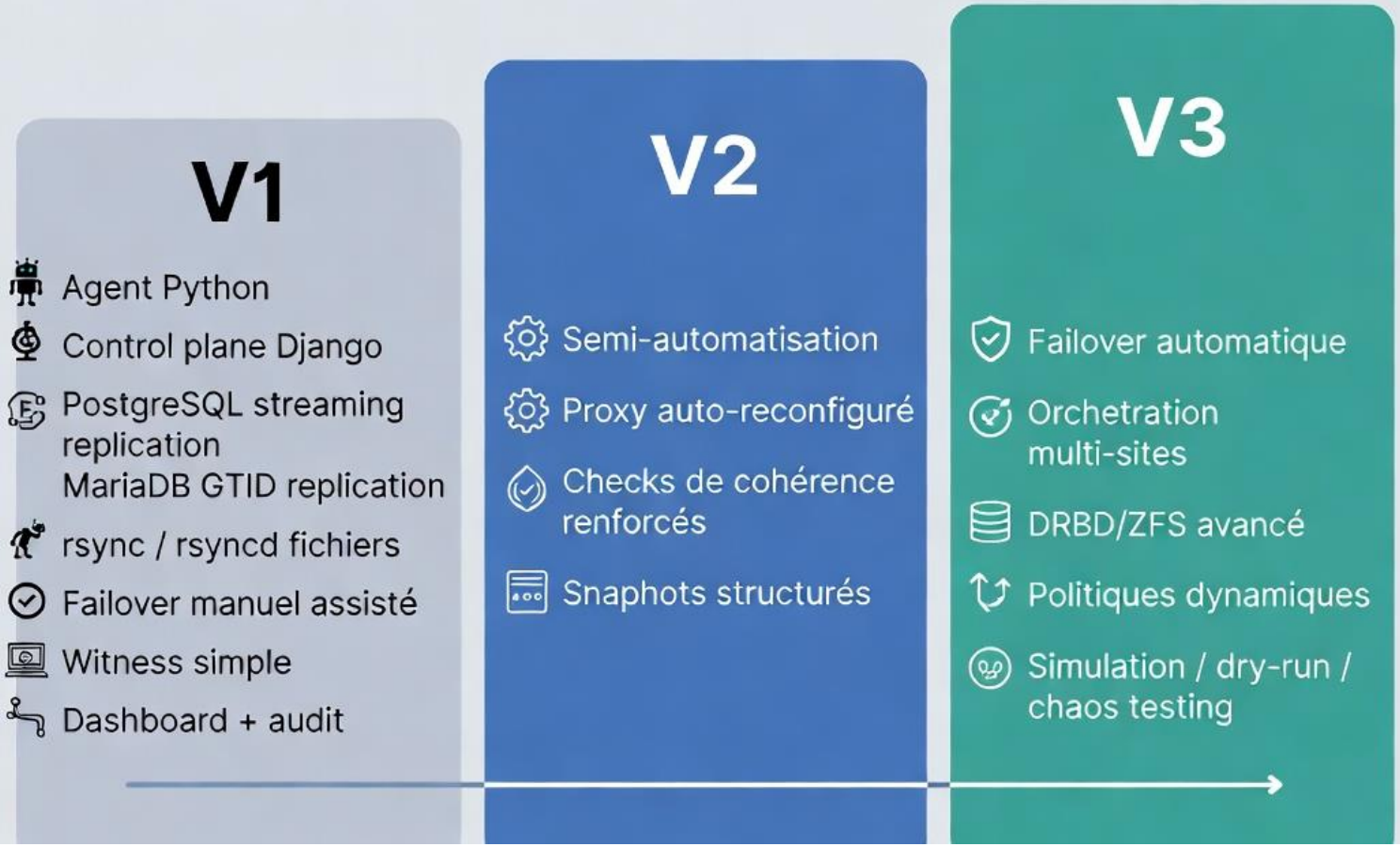
V2

- 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 :

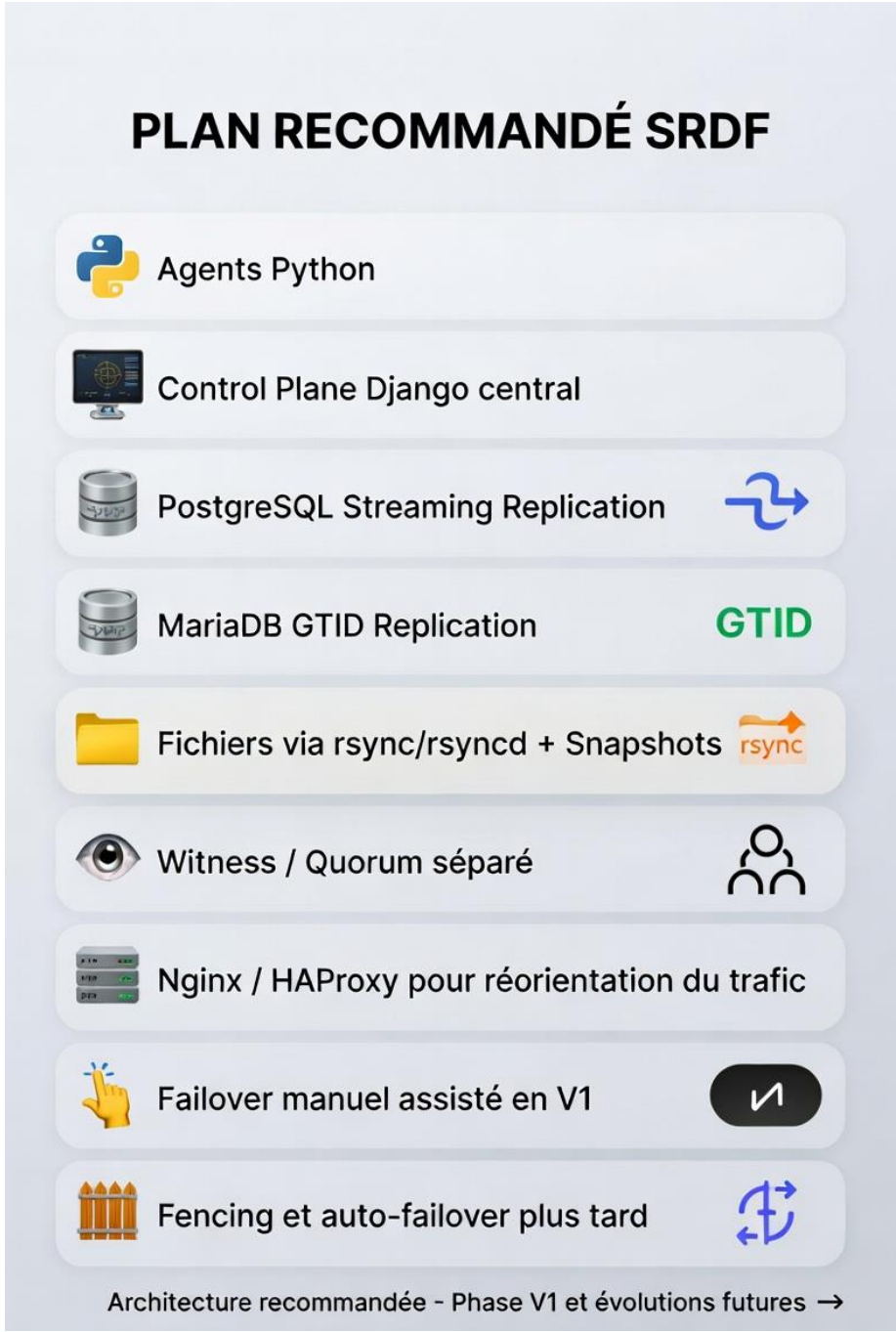
> en V1, **on ne réécrit pas PostgreSQL replication ni MariaDB replication.**
> On construit la **couche d’orchestration, de supervision, d’audit et d’exécution contrôlée.**

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

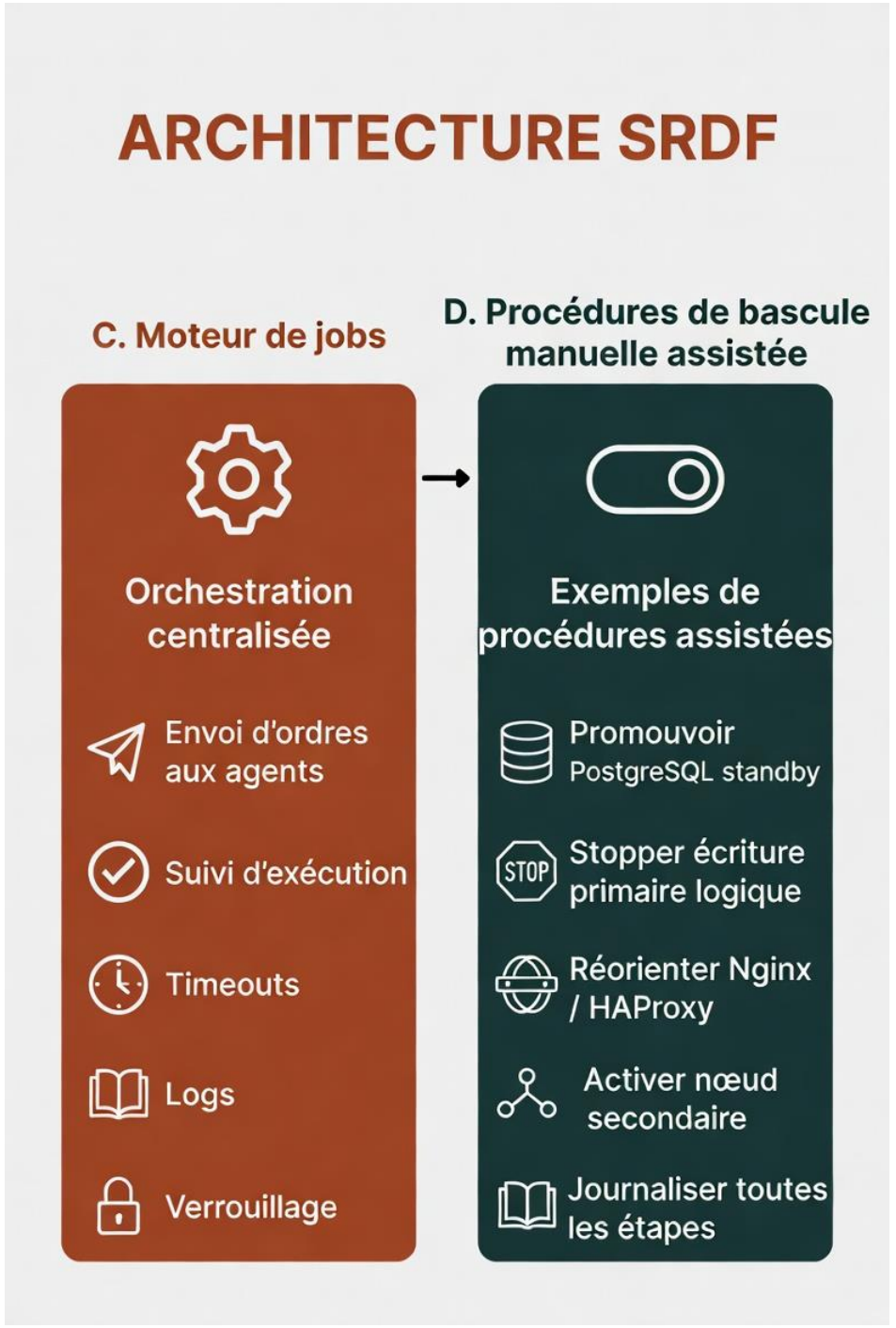
- envoi d'ordres aux agents
- suivi d'exécution
- timeouts
- logs
- verrouillage

D. Des procédures de bascule manuelle assistée

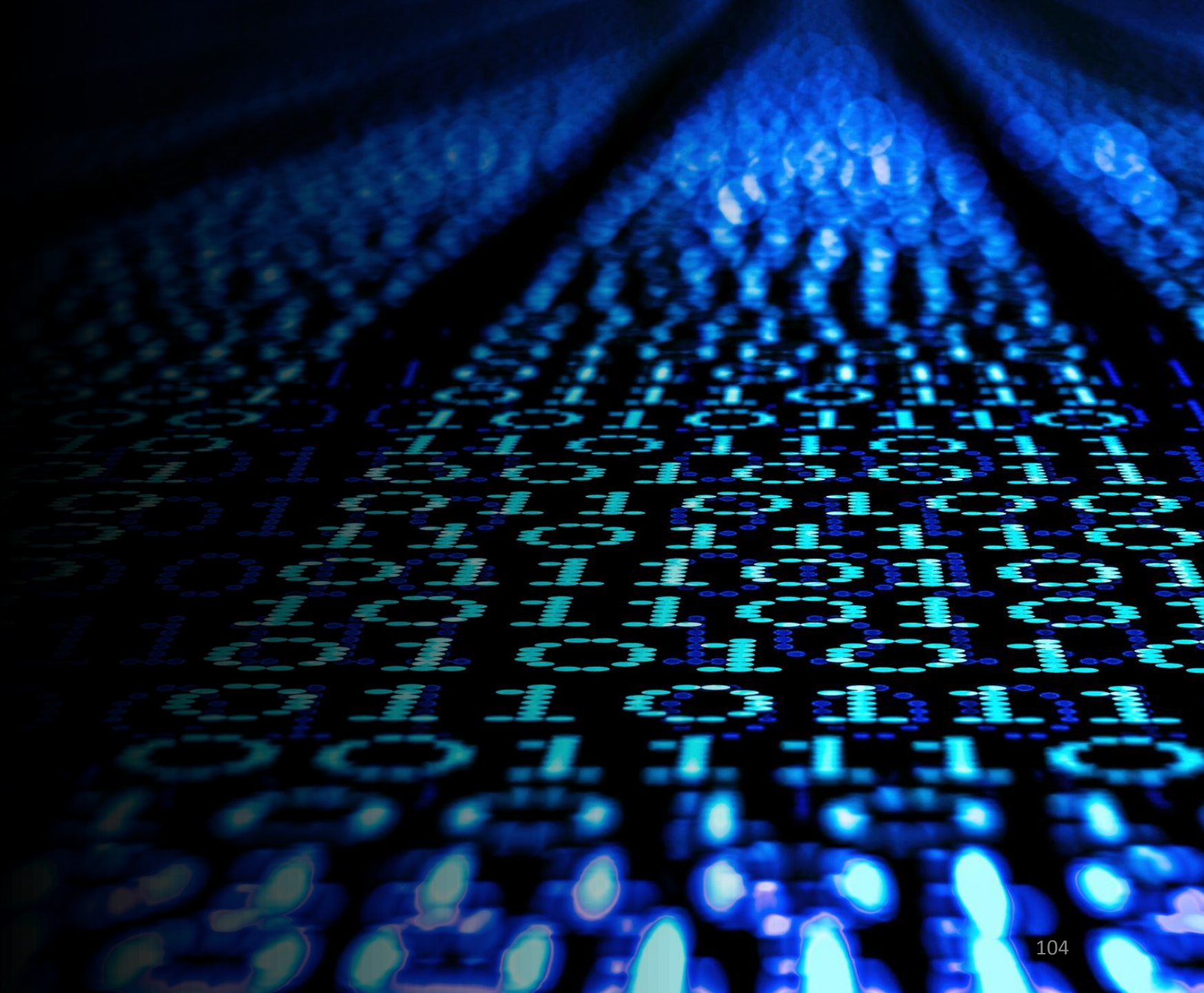
Exemples :

- 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

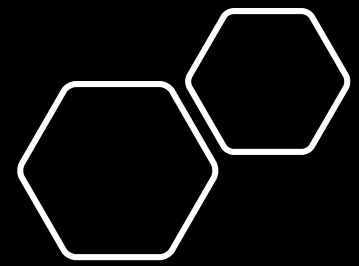
- `mariadb.status`
- `mariadb.set_read_only`
- `mariadb.set_read_write`
- `mariadb.stop_replica`
- `mariadb.start_replica`
- `mariadb.reset_replica`

7.4 Nginx / proxy

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

7.5 File sync

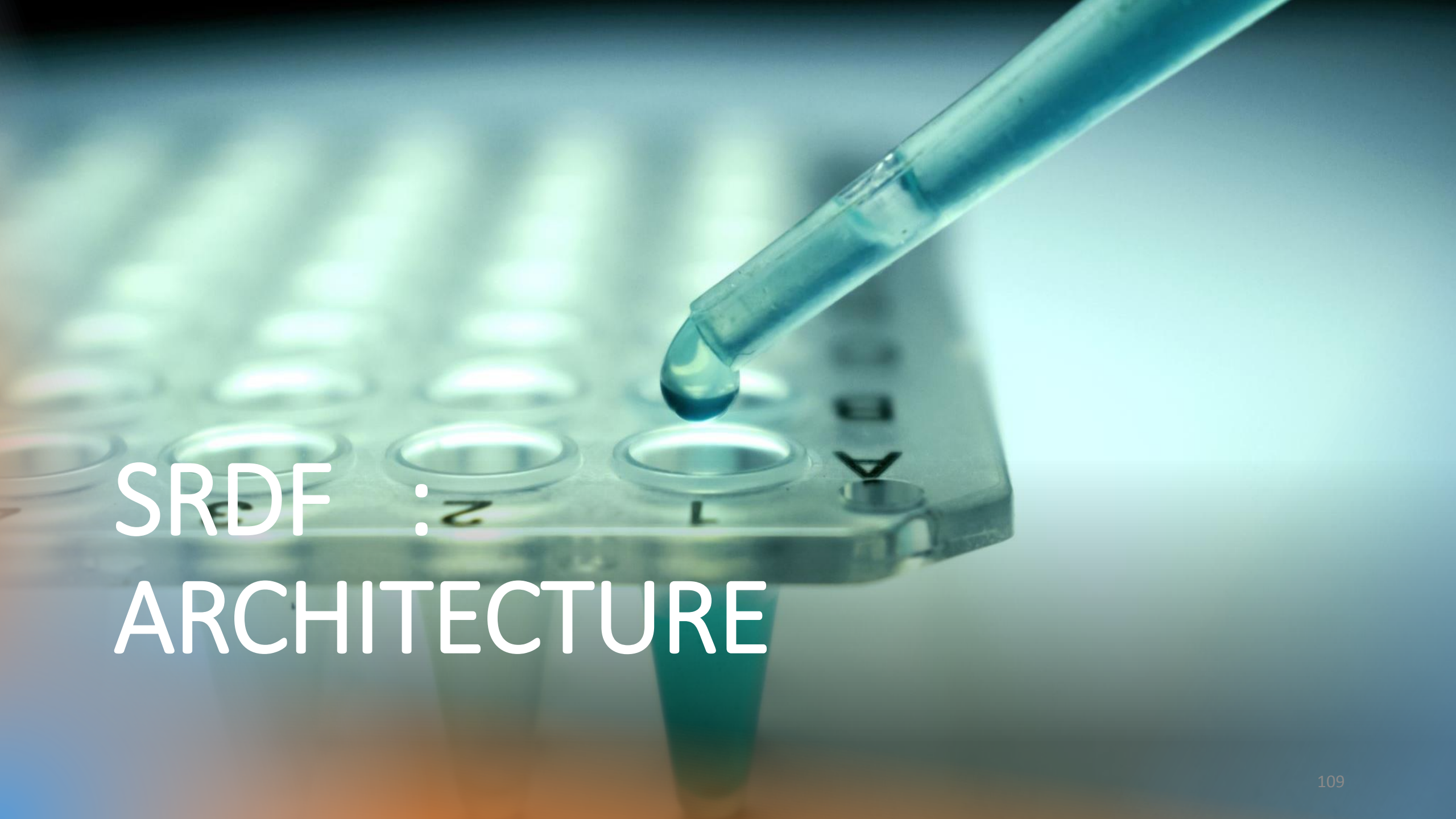
- `filesync.run_rsync`
- `filesync.snapshot_create`
- `filesync.verify_checksum`



SRDF ARCHITECTURE

COMPOSANTS

FLUX & RÉPLICATION



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

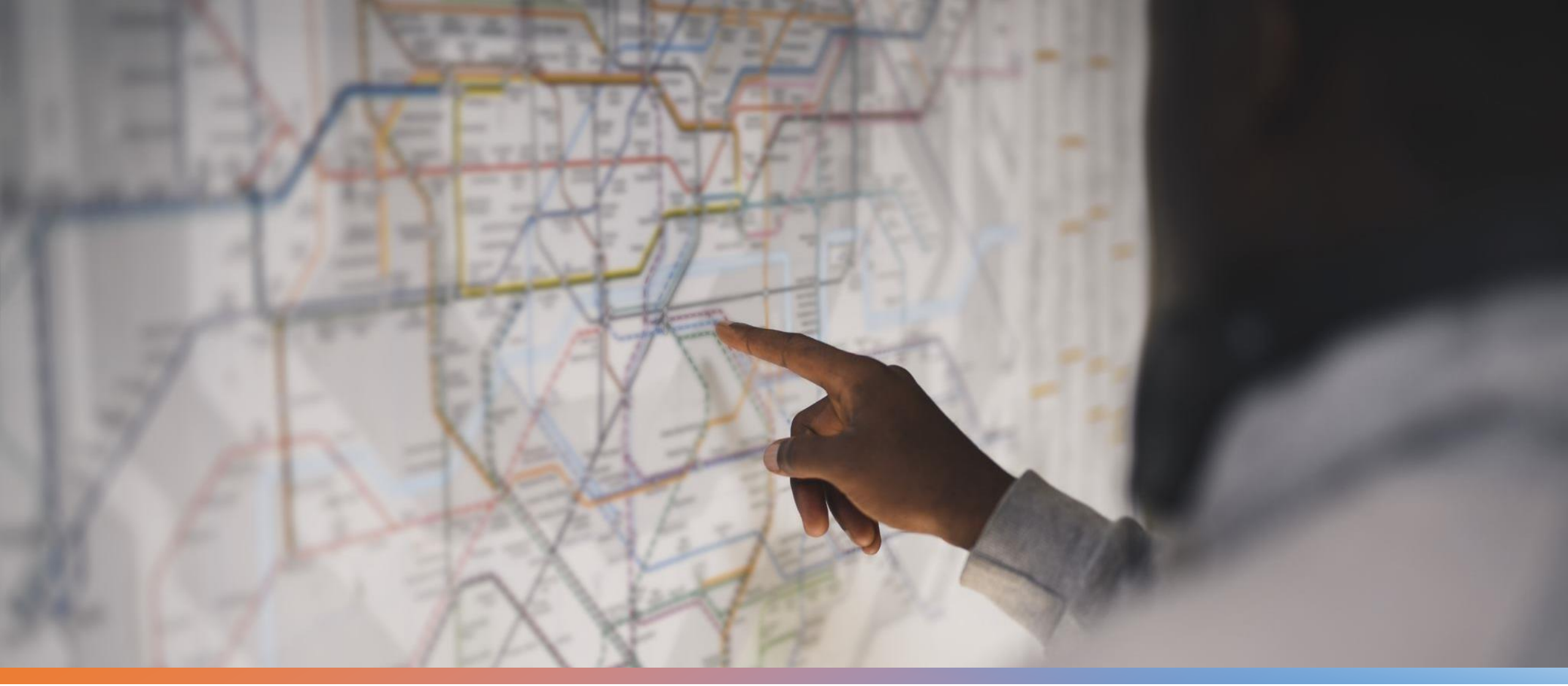
Dans `my.ini` MySQL :

```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

SRDF : CONCEPTS DE BASE EN PROFONDEUR

- 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

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

PIPELINE ACTUEL

```
[MySQL binlog]
      ↓
capture_binlog_once
      ↓
OutboundChangeEvent(engine=mysql)
      ↓
TransportBatch
      ↓
Receiver
      ↓
InboundChangeEvent(engine=mysql)
      ↓
Applier → MariaDB
```

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`

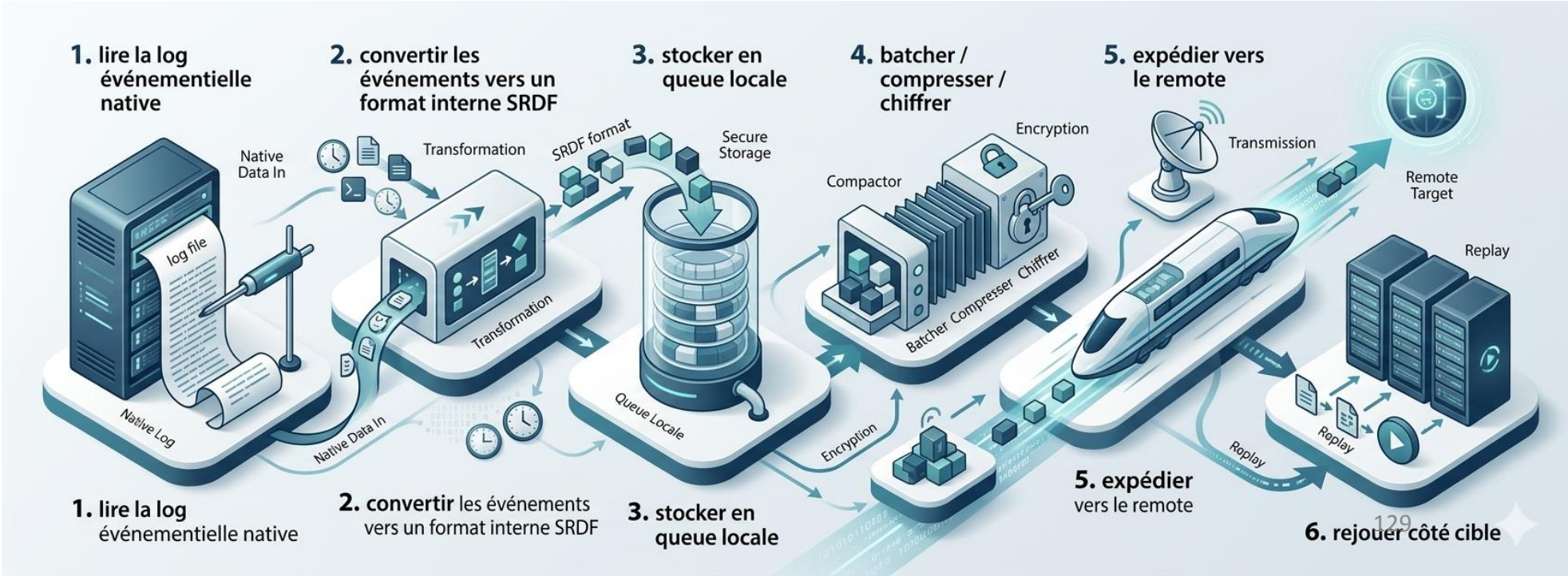
Le vrai coeur du projet

- 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

AGENDA

PLAN INITIAL

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 + Compression + Encryption
- 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**.

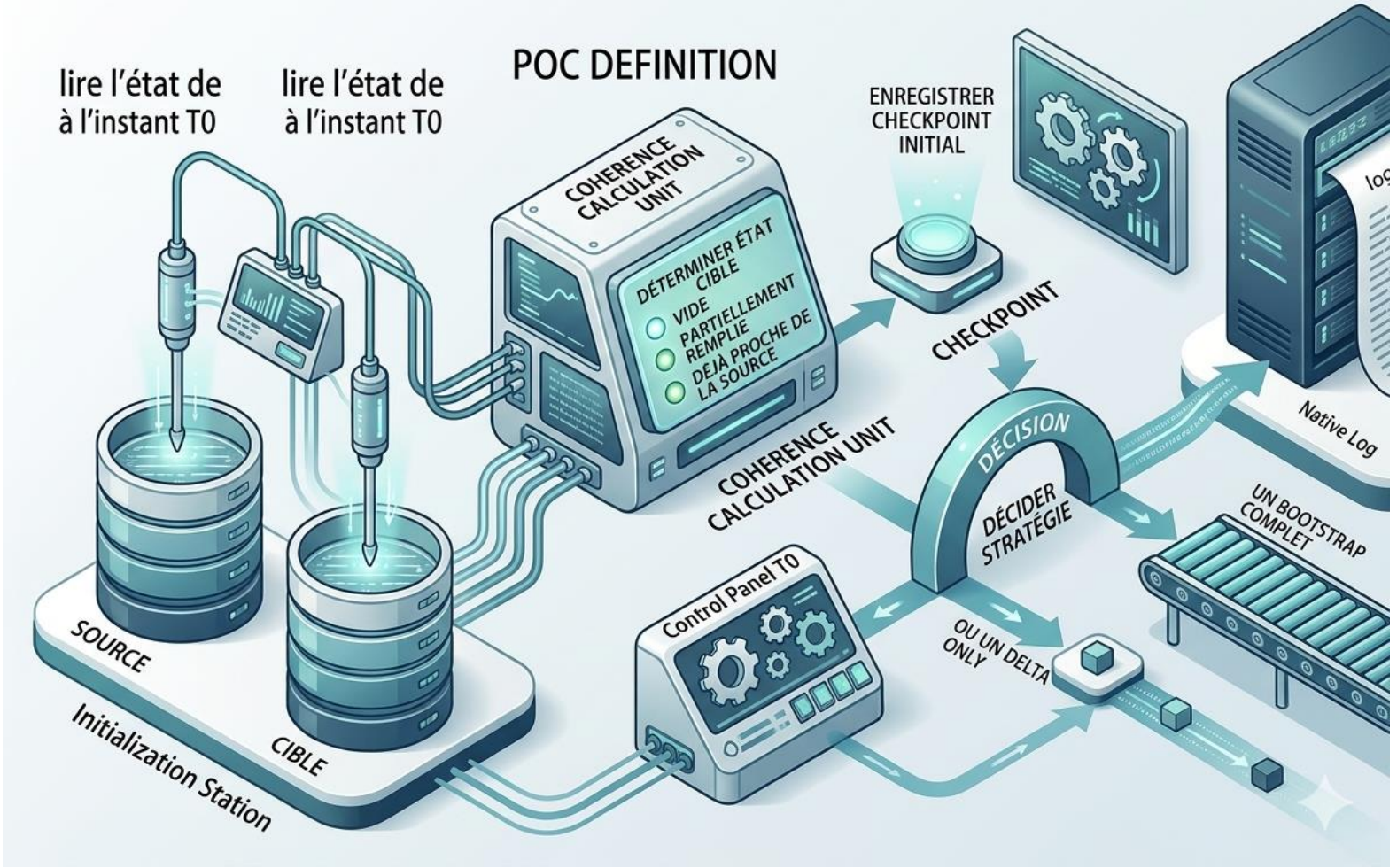
Concrètement :

- 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

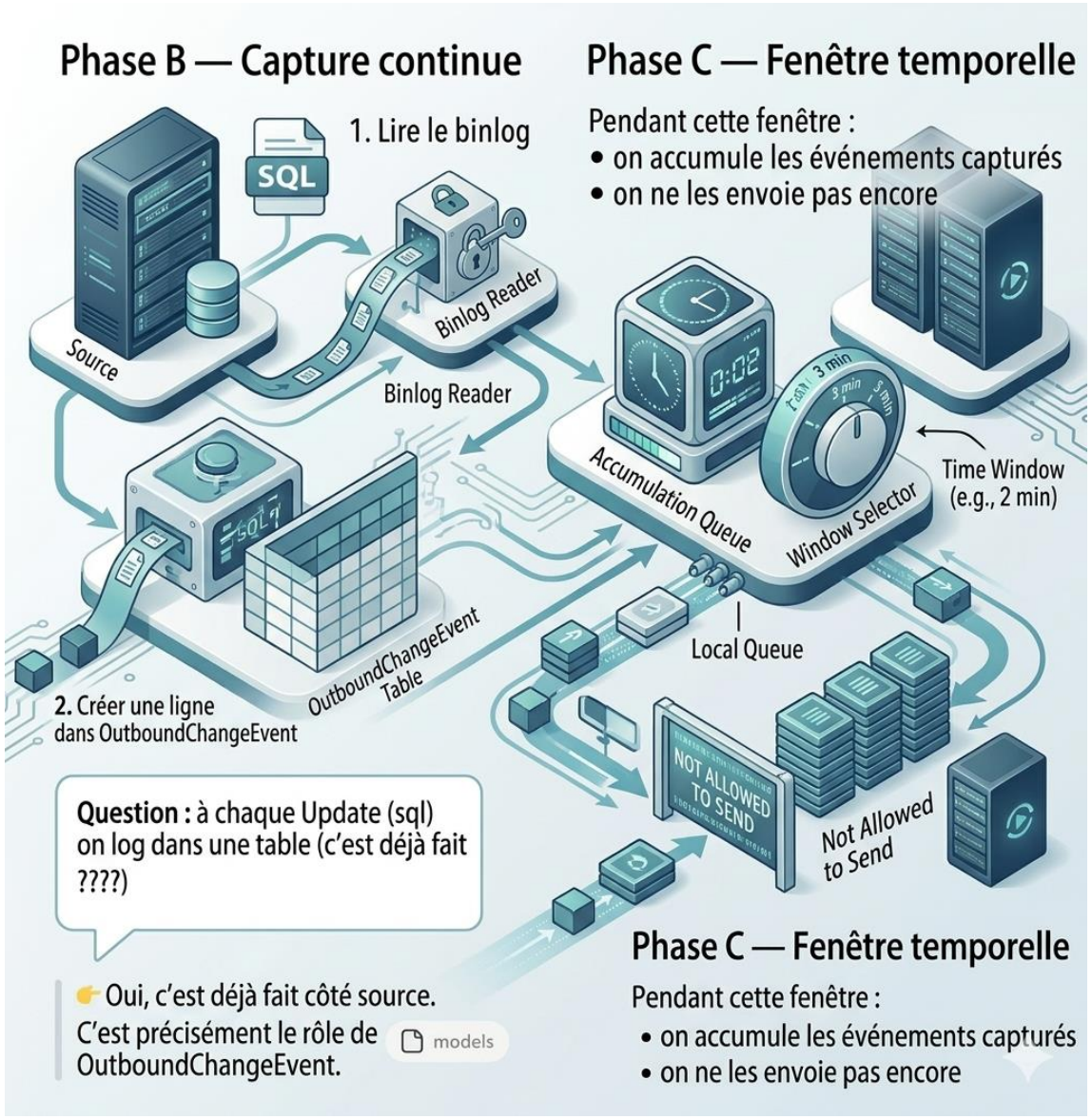
On ne doit pas envoyer chaque ligne immédiatement.

On définit une fenêtre, par exemple :

- 2 min
- ou 3 min
- ou 5 min

Pendant cette fenêtre :

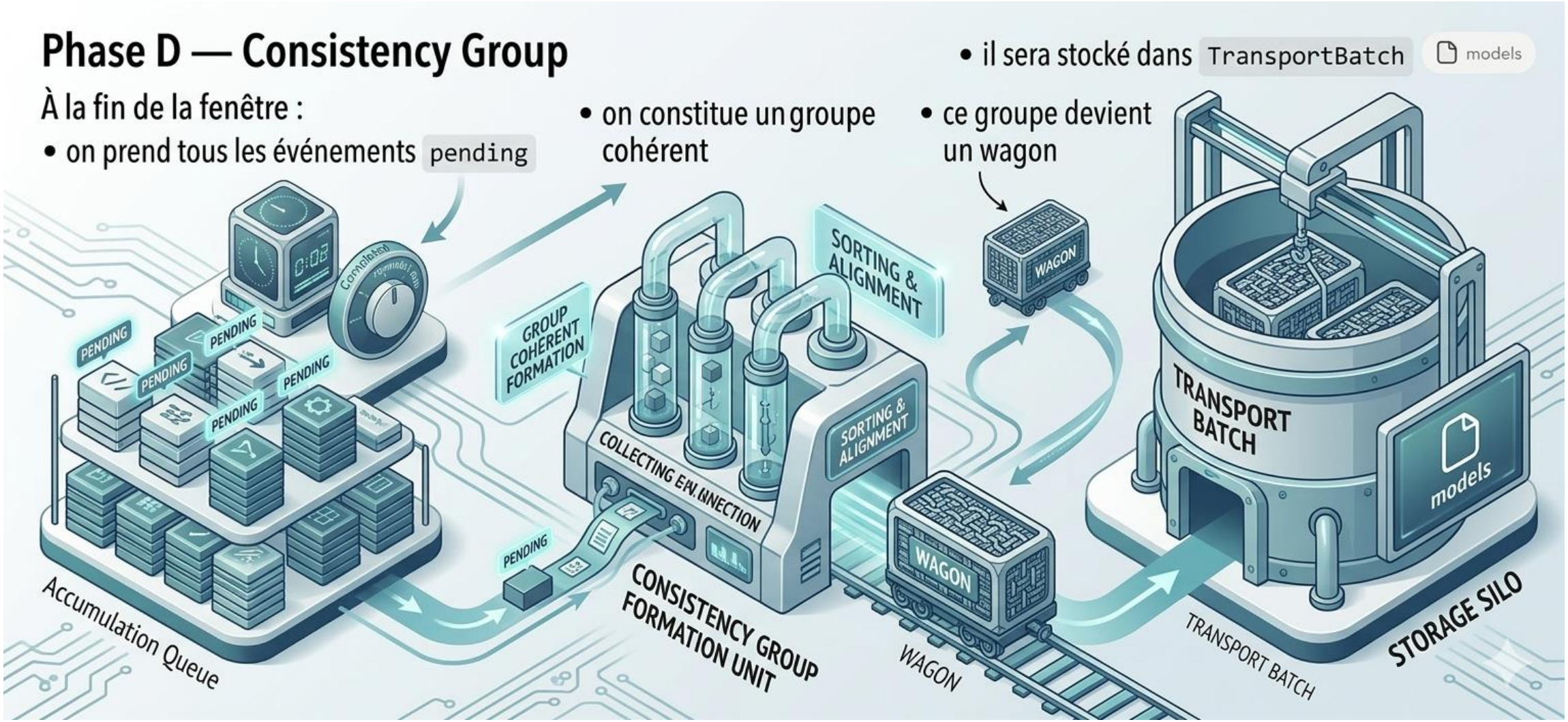
- on accumule les événements capturés
- on ne les envoie pas encore


Phase B — Capture continue
1. Lire le binlog
SQL
Source
Binlog Reader
Binlog Reader
OutboundChangeEvent Table
2. Créer une ligne dans OutboundChangeEvent
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
PENDING
PENDING
PENDING
PENDING
PENDING
GROUP COHERENT FORMATION
SORTING & ALIGNMENT
SORTING & ALIGNMENT
WAGON
WAGON
TRANSPORT BATCH
models
Accumulation Queue
CONSISTENCY GROUP FORMATION UNIT
WAGON
TRANSPORT BATCH
STORAGE SILO

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`

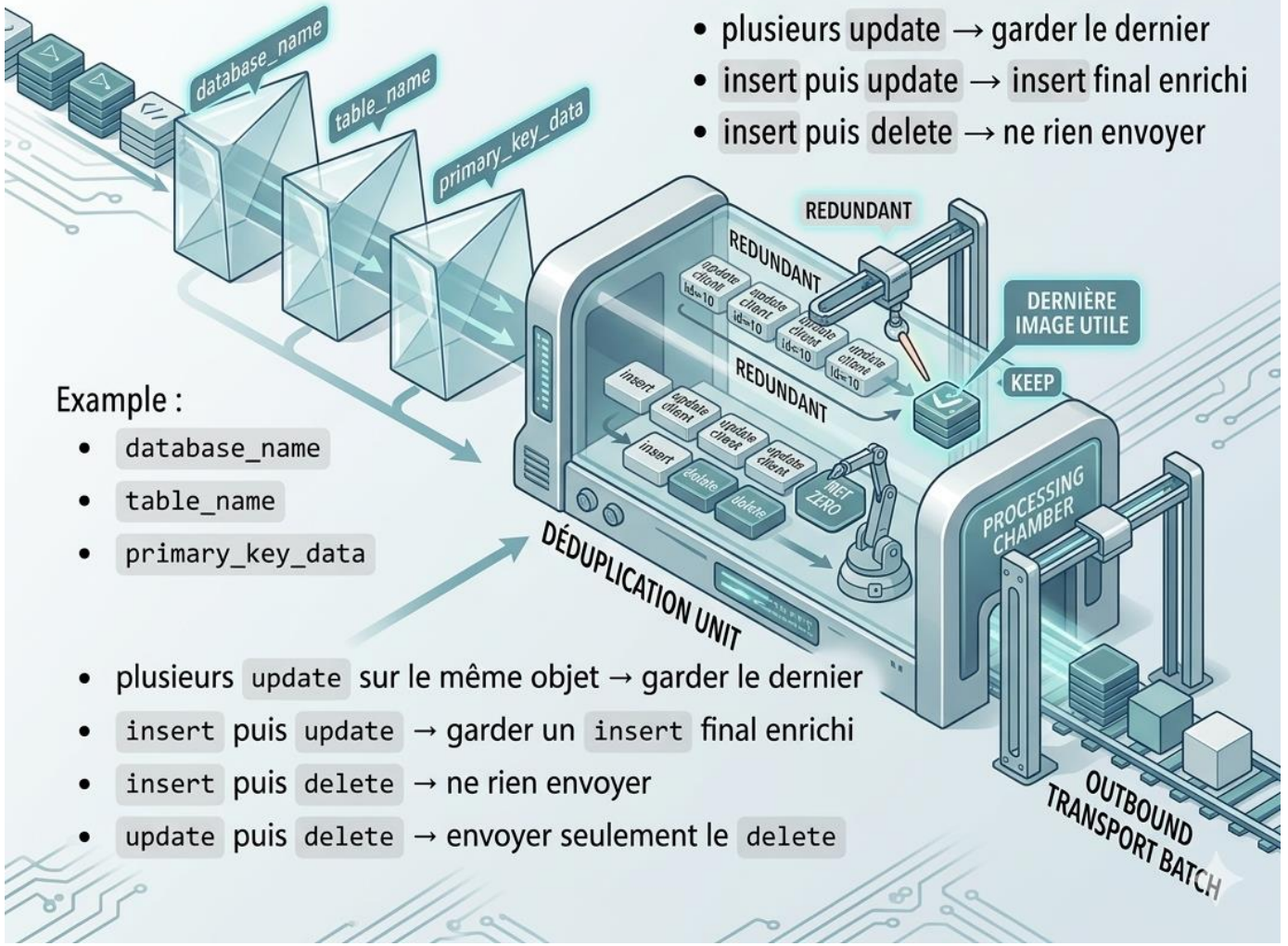
Et la règle sera :

- 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.

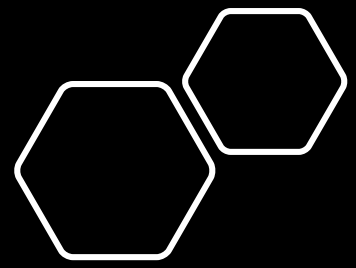
Elle doit d'abord recevoir dans une queue cible :

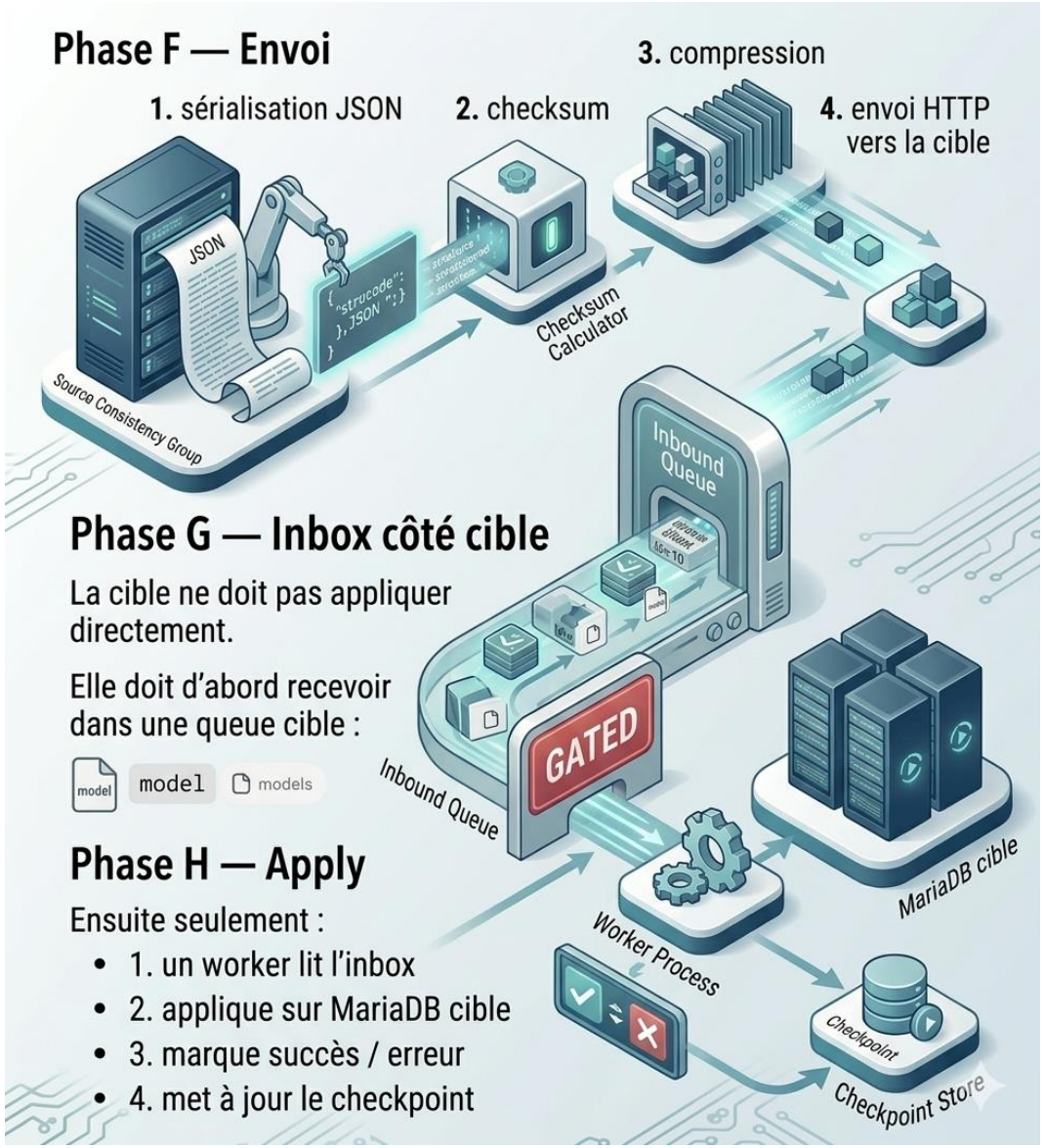
- `InboundChangeEvent` models

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
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
MariaDB cible
Checkpoint
Checkpoint Store

Phase I — Contrôle comparatif

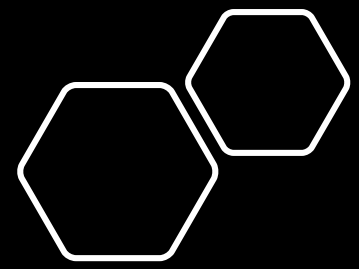
Et tu as raison : sans comparatif source/cible, on pilote à l'aveugle.

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



SRDF DATA MODELS

SRDF DATA MODELS

- A. BootstrapPlan
- B. BootstrapPlanTableSelection
- C. BootstrapPlanExecution
- D. BootstrapPlanExecutionItem
- E. bootstrap_plan_runner.py
- F. table_bootstrap.py

A. BootstrapPlan

C'est l'objet de pilotage principal.

Il définit :

- quel service de réplication utiliser
- quelle base source viser
- quelles tables inclure ou exclure
- la méthode (`sql_copy` ou `mysqldump`)
- les options comme `recreate_table` , `truncate_target` , `validate_snapshot`
- le parallélisme
- les derniers résultats globaux (`last_run_at` , `last_total_tables` , etc.)

admin_srdf

B. BootstrapPlanTableSelection

B. `BootstrapPlanTableSelection`

C'est le niveau table par table, quand on veut un plan fin.

Chaque entrée représente une table explicitement sélectionnée, avec son propre statut, sa dernière erreur et son dernier résultat.

Donc : **le détail opérationnel par table.**

C. BootstrapPlanExecution

C'est le journal d'une exécution complète du plan.

À chaque lancement, on crée une exécution globale avec :

- statut
- durée
- nombre de bases
- nombre de tables
- succès / échecs
- payload global
- éventuellement lien vers une exécution parente en cas de replay

Donc : **une exécution = un run historisé.**

D. BootstrapPlanExecutionItem

C'est le détail d'exécution par table.

À chaque table traitée, on crée une ligne avec :

- base
- table
- méthode
- nombre de lignes copiées
- checksums
- snapshot OK ou non
- erreur éventuelle
- durée

Donc : **la vraie trace fine d'exécution.**

E. bootstrap_plan_runner.py

C'est le moteur.

C'est lui qui :

- résout le plan
- construit les work items
- crée `BootstrapPlanExecution`
- traite les tables
- met à jour `BootstrapPlanTableSelection`
- remplit `BootstrapPlanExecutionItem`
- met à jour le statut final du plan et de l'exécution admin_srdf

Donc : **le runner exécute réellement.**

F. table_bootstrap.py

C'est la brique technique de copie table par table.

Elle sait :

- ouvrir les connexions source/cible
- créer la base cible si besoin
- recréer ou tronquer la table
- copier les données
- compter les lignes
- calculer les checksums
- retourner un résultat propre

Donc : **le runner orchestre,** `table_bootstrap.py` **copie.**

PLAN SRDF V1.1

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

PLANS DE TESTES ET DE VALIDATION

- Phase 0 — Préparation
- Phase 1 — Générer un vrai changement en base
- Phase 2 — Lancer la capture binlog
- Phase 3 — Lancer le build batch
- Phase 4 — Lancer l’envoi batch

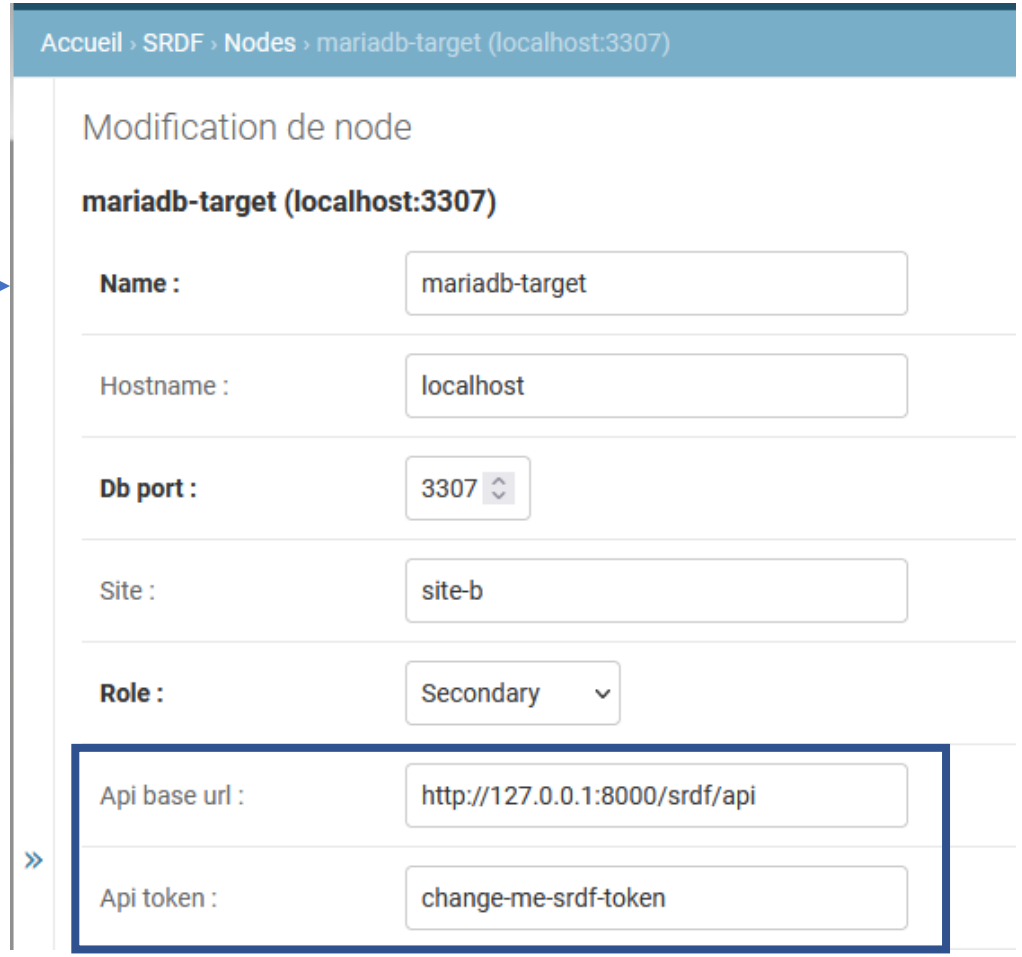
SRDF SETTINGS

PARAMÈTRES

CONFIGURATION SRDF

BASE SRDF SETTINGS

- Django settings :
  - SRDF_API_TOKEN
- Node Data Model :



GESTION
DES PARAMETRES

Global default

- `key = capture.max_runtime_seconds`
- `scope_type = global`
- `value_text = 20`

Par engine

- `key = capture.max_runtime_seconds`
- `scope_type = engine`
- `engine = mariadb`
- `value_text = 30`

Par client/company

- `key = capture.bulk_insert_batch_size`
- `scope_type = company`
- `company_name = IDEO-LAB`
- `value_text = 500`

Par service

- `key = capture.scan_window_max_bytes`
- `scope_type = service`
- `replication_service = Mariadb-replica`
- `value_text = 67108864`

Oui, il faut maintenant introduire une vraie couche :

- **paramètres SRDF globaux**
- **paramètres SRDF par engine**
- **paramètres SRDF par client/company**
- **paramètres SRDF par service**

C'est la bonne base pour industrialiser SRDF.

DATA MODELS

1. Une table de définition de paramètres

Pour décrire ce qu'est un paramètre :

- clé technique
- catégorie
- type
- valeur par défaut plateforme
- description
- phase SRDF concernée
- moteur concerné éventuel

2. Une table de valeurs effectives

Pour stocker les overrides :

- paramètre
- scope_type : global / engine / company / client / service
- scope_ref_id ou nom
- engine optionnel
- valeur
- priorité
- enabled

EXEMPLES DE PARAMTRES A SAUVEGARDER

```
capture.limit
capture.max_scanned_events
capture.max_runtime_seconds
capture.scan_window_max_files
capture.scan_window_max_bytes
capture.bulk_insert_batch_size
capture.wagon_max_events
capture.progress_enabled
capture.progress_every_events
capture.progress_every_seconds
transport.sender.batch_size
transport.sender.retry_limit
apply.limit
bootstrap.parallel_workers
bootstrap.database_parallel_workers
orchestrator.sleep_seconds
```

PIPELINE COMPLET (ordre exact)

- 1. CAPTURE (SOURCE)
- 2. BUILD BATCH
- 3. SEND BATCH
- 4. RECEIVE (TARGET)
- 5. APPLY (TON TEST ACTUEL)

Data Pipeline : Source to Target Apply

CAPTURE & BUILD

1 CAPTURE (SOURCE)

```bash
python manage.py srdf_capture_mariadb_binlog --service-id=1 --limit=50
```

👉 doit créer :

```
OutboundChangeEvent > 0
```

---

2 BUILD BATCH

```bash
python manage.py srdf_transport --action=build_batch --service-id=1
```

👉 doit créer :

```
TransportBatch.status = ready
```

SEND BATCH & RECEIVE (TARGET)

3 SEND BATCH

```bash
python manage.py srdf_transport --action=send_batch --batch-id=X
```

👉 doit :

```
POST → target node API
```

---

4 RECEIVE (TARGET)

👉 doit créer :

```
InboundChangeEvent > 0
```

---

5 APPLY (TON TEST ACTUEL)

```bash
python manage.py srdf_apply_inbound --service-id=1
```

Phase 0 — Préparation

Action à faire

Choisis **une table de test** sur MariaDB source, avec **une ligne connue.**

Exemple :

- table : `customer_test`
- ligne : `id = 101`

Contrôle à faire avant test

Dans Django admin :

- `OutboundChangeEvent` : note le nombre actuel
- `TransportBatch` : note le nombre actuel
- `AuditEvent` : note les derniers événements
- éventuellement `ReplicationCheckpoint` : note `log_file` / `log_pos` actuels

Phase 1 — Générer un vrai changement en base

Action à faire

Exécute **un seul UPDATE réel** sur la base source.

Exemple :

SQL

```sql
UPDATE customer_test
SET last_name = 'SRDF_TEST_001'
WHERE id = 101;
```

Ce que tu contrôles

À ce stade, **rien ne doit encore apparaître tout seul** dans Django admin tant que le cron/service de capture n'a pas tourné.

Phase 2 — Lancer la capture binlog

Résultat attendu en stdout

Tu dois voir un résultat de type :

- capture exécutée
- `captured_count >= 1`
- `log_file`
- `log_pos`

Contrôle Django admin à faire

Dans `OutboundChangeEvent` :

- une **nouvelle ligne** doit apparaître
- `status = pending`
- `operation = update`
- `database_name` correct
- `table_name` correct
- `primary_key_data` contient la PK de la ligne
- `after_data` contient la nouvelle valeur
- `log_pos > 0`

Phase 2 : Validation finale

Dans `AuditEvent` :

- un événement `binlog_capture_run` doit apparaître
  ou `binlog_capture_error` si échec

Dans `ReplicationCheckpoint` :

- le checkpoint `direction = capture` doit avancer (`log_file` / `log_pos`)

Validation de phase

Phase OK si :

- ton UPDATE SQL a produit **au moins 1** `OutboundChangeEvent` **pending**

Phase 3 — Lancer le build batch

Action à faire

Lance **le cron/service/management command actuel** qui appelle :

Python

Exécuter

```
transport.build_batch
```

avec :

- `replication_service_id`
- éventuellement `limit=100`

Résultat attendu en stdout

Tu dois voir quelque chose comme :

- service trouvé
- batch créé
- `event_count >= 1`
- `batch_uid`
- `checksum`

Phase 3 – Controle Admin tool

Contrôle Django admin à faire

Dans `TransportBatch` :

- une **nouvelle ligne** doit apparaître
- `status = ready`
- `event_count >= 1`
- `first_event_id` rempli
- `last_event_id` rempli
- `checksum` rempli
- `compression = none`
- `payload` non vide

Dans `OutboundChangeEvent` :

- l'event précédemment en `pending` doit passer en :
  - `status = batched`
  - `batched_at` rempli

Dans `AuditEvent` :

- un événement `transport_batch_built` doit apparaître

Phase 4 — Lancer l’envoi batch

Action à faire

Lance **le cron/service/management command actuel** qui appelle :

Python

```
transport.send_batch
```

avec :

- `batch_id = <id du batch ready>` action_runner

Cas réel à accepter aujourd’hui

Comme le receiver distant n’est pas encore clairement en place dans le code actuel, il y a 2 **résultats acceptables** :

Cas A — envoi échoue
C’est acceptable aujourd’hui.

Contrôle Django admin

Dans `TransportBatch` :

- `status = failed`
- `retry_count` augmente
- `last_error` rempli models

Phase 4 - Controle Final

Dans `AuditEvent` :

- événement `transport_batch_failed` models

Cas B — envoi réussit
Si le node distant répond.

Contrôle Django admin

Dans `TransportBatch` :

- `status = sent`
- `sent_at` rempli models

Dans `AuditEvent` :

- événement `transport_batch_sent`

Validation de phase

Phase OK si :

- le batch quitte l'état `ready`
- il passe soit en `sent`, soit en `failed` avec erreur tracée

Résumé final sur le Plan de test

Résumé ultra-simple du test

Test minimal complet

Toi, tu fais :

1. un `UPDATE` SQL réel sur la base source
2. tu lances le cron/service de **capture**
3. tu lances le cron/service de **build batch**
4. tu lances le cron/service de **send batch**

Et tu contrôles dans l'admin :

1. `OutboundChangeEvent`
   - un event `pending` apparaît après capture
2. `TransportBatch`
   - un batch `ready` apparaît après build
3. `OutboundChangeEvent`
   - l'event devient `batched`
4. `TransportBatch`
   - le batch devient `sent` ou `failed`
5. `AuditEvent`
   - les traces existent à chaque étape

Ce qu’il faut contrôler exactement, table par table

- OutboundChangeEvent
- TransportBatch
- AuditEvent
- ReplicationCheckpoint

OutboundChangeEvent

`OutboundChangeEvent`

Après capture

- nouvelle ligne
- `status = pending`

Après build batch

- même ligne
- `status = batched`

Après send

- pas forcément de changement immédiat supplémentaire aujourd'hui

TransportBatch

Après build

- nouvelle ligne
- `status = ready`

Après send

- `status = sent` ou `failed`

AuditEvent

`AuditEvent`

Tu dois voir apparaître successivement :

- `binlog_capture_run`
- `transport_batch_built`
- `transport_batch_sent` **ou** `transport_batch_failed`

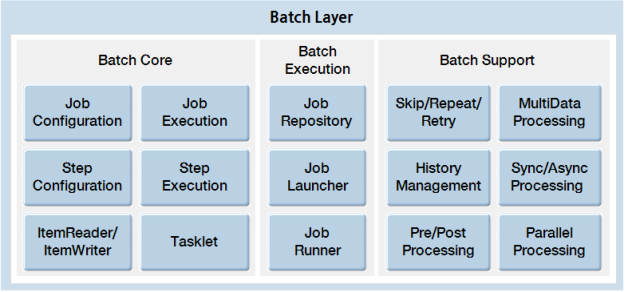
Après capture

- le checkpoint capture doit avancer

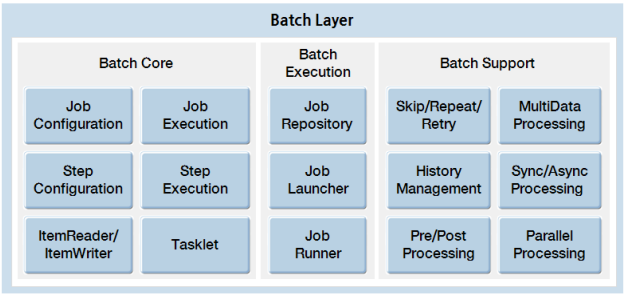
ReplicationCheckpoint


CRON DJANGO
12
SRDF
02:00
SCRIPTS RUNNING...
TASKS
BATCH PROCESSING
0 2 * * * script.py

CRON REFERENCES



SRDF CRON REFERENCES



- **1. srdf_capture_mariadb_binlog** : Capture Updates on DataBas
- **2. srdf_transport** : Read Replication/Count Pending/Reseau
  - **build_batch**
  - **send_batch**
- **3. srdf_apply_inbound** : appliquer sur la base cible MariaDB
- **4. srdf_bootstrap_service** : Bootstrap complet d'une DB, Tables
- **5. srdf_capture_agent** : un vrai premier daemon/service-ready.
- **6. srdf_pipeline_agent** : SRDF PIPELINE AGENT
- **7. srdf_monitor_status** : Le target appelle le source pour confirmer le batch.
- **8. srdf_run_bootstrap_plan** : Chargement initial de certaines tables
- **9. srdf_refresh_source_catalog** : PRE Chargement de la liste DB et Tables
- **10. srdf_daemon** : AUTOMATED DEAMON SERVICES

srdf_refresh_source_catalog

rafraîchir le catalogue SRDF (DB et Tables)

Commande :

```bash
python manage.py srdf_refresh_source_catalog --service-id=1
```

Ça remplit automatiquement :

- `SourceDatabaseCatalog`
- `SourceTableCatalog`

```
  warnings.warn(eol_message.format("3.8"), FutureWarning)
Catalog refreshed. service_id=1 databases=16 tables=2013
```

srdf_capture_mariadb_binlog

srdf_transport

CRON : srdf_transport

Ce qu'elle fait

- lit le `ReplicationService`
- compte les `pending` avant
- appelle `build_transport_batch_once(...)`
- relit :
  - `pending_after`
  - `batched_after`
  - état du batch créé
- affiche un résumé clair en stdout

Ce que tu contrôles ensuite dans l'admin

- `TransportBatch` : nouvelle ligne `ready`
- `OutboundChangeEvent` : les `pending` pris deviennent `batched`

4) OPTION `--strict-send`

Commande :

```bash
python manage.py srdf_transport --action cycle --service-id 1 --strict-send
```

Comportement

- sans `--strict-send`, `cycle` accepte :
  - `sent`
  - ou `failed`
- avec `--strict-send`, `cycle` exige :
  - `sent`

C'est utile quand le remote sera plus stable.

CRON : srdf_transport sample use

5) EXEMPLES DE TEST CONCRETS

Cas 1 — build seul

```bash
python manage.py srdf_transport --action build_batch --service-id 1
```

Cas 2 — send seul

```bash
python manage.py srdf_transport --action send_batch --batch-id 25
```

Cas 3 — cycle complet

```bash
python manage.py srdf_transport --action cycle --service-id 1
```

Cas 4 — cycle complet avec fenêtre plus large

```bash
python manage.py srdf_transport --action cycle --service-id 1 --limit 500
```

CRON : srdf_transport Hard Reset

```bash
python manage.py srdf_transport --action=hard_reset --service-id=1
```

Effet

Nettoie intelligemment **pour ce service uniquement** :

- `InboundChangeEvent`
- `TransportBatch`
- `OutboundChangeEvent`
- `ReplicationCheckpoint`

Sans toucher à :

- `Node`
- `ReplicationService`
- `AuditEvent`
- `ActionExecution`

CRON : srdf_transport BinLog Reset

`hard_reset` **+ reset binlog source**

Commande :

```bash
python manage.py srdf_transport --action=hard_reset --service-id=1 --reset-so
```

Effet supplémentaire

Exécute :

```sql
RESET MASTER
```

sur la DB source.

CRON : srdf_transport stdOut

Build

Exemple :

```
[BUILD] Starting build_batch
[BUILD] service=MariaDB Source -> DR id=1
[BUILD] pending_before=3
[BUILD] batch_count_before=12
[BUILD] pending_after=0
[BUILD] batched_after=3
[BUILD] batch_count_after=13
[BUILD][OK] batch_id=13 batch_uid=... status=ready event_count=3 check
```

Send

Exemple :

```
[SEND] Starting send_batch
[SEND] batch_id=13
[SEND] batch_uid=...
[SEND] current_status=ready
[SEND] target_node=node-b
[SEND] status_after=failed
[SEND] retry_count=1
[SEND] last_error=Action 'transport.receive_batch' failed ...
[SEND][FAILED-ACCEPTED] batch_id=13 error=...
```

RESUME : ORDRE DE TEST RECOMMANDÉ

Reprise d'un batch failed existant

Bash

```
python manage.py srdf_transport --action=resume_batch --batch-id=3
python manage.py srdf_transport --action=build_batch --service-id=1
python manage.py srdf_transport --action=send_batch --batch-id=<NOUVEAU_BATCH_
```

Remise à zéro propre du pipeline SRDF

Remise à zéro propre du pipeline SRDF

```bash
python manage.py srdf_transport --action=hard_reset --service-id=1
```

Puis :

- refaire un update source
- capture
- build
- send
- receive
- apply

CRON : srdf_transport (Cycle)

Cycle

Exemple :

```
[CYCLE] Starting cycle
[CYCLE] service=MariaDB Source -> DR id=1
[CYCLE] pending_before=2
[CYCLE][BUILD] batch_id=14 status=ready event_count=2
[CYCLE][SEND] batch_id=14 status_after_send=failed retry_count=1 last_
[CYCLE] pending_after=0
[CYCLE] batched_after=5
[CYCLE] sent_after=0
[CYCLE] failed_after=2
[CYCLE][OK] cycle completed
```

CRON : srdf_transport (Test)

```
============================================================
SRDF TRANSPORT COMMAND - START
============================================================
action=build_batch
service_id=1
batch_id=None
limit=100
strict_send=False

[BUILD] Starting build_batch
[BUILD] service=Mariadb-replica id=1
[BUILD] pending_before=101
[BUILD] batch_count_before=0
[BUILD] pending_after=1
[BUILD] batched_after=100
[BUILD] batch_count_after=1
[BUILD][OK] batch_id=1 batch_uid=37d91a43-68a8-48f5-ba24-f9bccef4c644 status=rea
dy event_count=100 checksum=3a3520d1278fad1d58980ec7affd25b399adc36d29f285ac185f
8a6b1017defe

============================================================
SRDF TRANSPORT COMMAND - FINAL SUMMARY
============================================================
ok=True
message=TransportBatch built for service 'Mariadb-replica'
duration_seconds=0.103

JSON_RESULT_BEGIN
{
  "ok": true,
  "message": "TransportBatch built for service 'Mariadb-replica'",
  "action": "build_batch",
  "service_id": 1,
  "service_name": "Mariadb-replica",
  "pending_before": 101,
  "pending_after": 1,
  "batched_after": 100,
  "batch_count_before": 0,
  "batch_count_after": 1,
  "build_result": {
    "ok": true,
    "message": "TransportBatch built for service 'Mariadb-replica'",
    "replication_service_id": 1,
    "transport_batch_id": 1,
    "batch_uid": "37d91a43-68a8-48f5-ba24-f9bccef4c644",
    "batch_created": true,
    "event_count": 100,
    "first_event_id": 1,
    "last_event_id": 100,
    "checksum": "3a3520d1278fad1d58980ec7affd25b399adc36d29f285ac185f8a6b1017def
e",
    "status": "ready"
  },
  "transport_batch": {
    "id": 1,
    "batch_uid": "37d91a43-68a8-48f5-ba24-f9bccef4c644",
    "status": "ready",
    "event_count": 100,
    "first_event_id": 1,
    "last_event_id": 100,
    "checksum": "3a3520d1278fad1d58980ec7affd25b399adc36d29f285ac185f8a6b1017def
e"
  }
}
JSON_RESULT_END

============================================================
SRDF TRANSPORT COMMAND - END
============================================================
```

CRON : srdf_transport (cycle complet)

```
============================================================
SRDF TRANSPORT COMMAND - START
============================================================
action=cycle
service_id=1
batch_id=None
limit=100
strict_send=False

[CYCLE] Starting cycle
[CYCLE] service=Mariadb-replica id=1
[CYCLE] pending_before=1
[CYCLE][BUILD] batch_id=2 status=ready event_count=1
[CYCLE][SEND] batch_id=2 status_after_send=failed retry_count=1 last_error=Node
'mariadb-target' has no api_base_url configured
[CYCLE] pending_after=0
[CYCLE] batched_after=101
[CYCLE] sent_after=0
[CYCLE] failed_after=1
[CYCLE][OK] cycle completed

============================================================
SRDF TRANSPORT COMMAND - FINAL SUMMARY
============================================================
ok=True
message=cycle completed
duration_seconds=0.11

JSON_RESULT_BEGIN
{
  "ok": true,
  "message": "cycle completed",
  "action": "cycle",
  "service_id": 1,
  "service_name": "Mariadb-replica",
  "pending_before": 1,
  "pending_after": 0,
  "batched_after": 101,
  "sent_after": 0,
  "failed_after": 1,
  "build_result": {
    "ok": true,
    "message": "TransportBatch built for service 'Mariadb-replica'",
    "replication_service_id": 1,
    "transport_batch_id": 2,
    "batch_uid": "7babbcbe-63ee-4f44-8e6d-6ae41d92c85e",
    "batch_created": true,
    "event_count": 1,
    "first_event_id": 101,
    "last_event_id": 101,
    "checksum": "59618d3e303ccfb1ceebe6b767e4b56a44e247a3d4482385e67ae6a518e0b6
c",
    "status": "ready"
  },
  "send_result": {},
  "send_exception": "Node 'mariadb-target' has no api_base_url configured",
  "transport_batch": {
    "id": 2,
    "batch_uid": "7babbcbe-63ee-4f44-8e6d-6ae41d92c85e",
    "status": "failed",
    "retry_count": 1,
    "last_error": "Node 'mariadb-target' has no api_base_url configured",
    "sent_at": null
  }
}
JSON_RESULT_END

============================================================
SRDF TRANSPORT COMMAND - END
============================================================


guillaume@guillaume-PC MINGW32 /d/wamp5/idea_lab (MASTER_MAIN_2026)
```

srdf_transport (build batch)

```
============================================================
SRDF TRANSPORT COMMAND - START
============================================================
action=build_batch
service_id=1
batch_id=None
limit=100
strict_send=False

[BUILD] Starting build_batch
[BUILD] service=Mariadb-replica id=1
[BUILD] pending_before=0
[BUILD] batch_count_before=2
[BUILD] pending_after=0
[BUILD] batched_after=101
[BUILD] batch_count_after=2
[BUILD] no batch created

============================================================
SRDF TRANSPORT COMMAND - FINAL SUMMARY
============================================================
ok=True
message=No pending outbound events for service 'Mariadb-replica'
duration_seconds=0.034

JSON_RESULT_BEGIN
{
  "ok": true,
  "message": "No pending outbound events for service 'Mariadb-replica'",
  "action": "build_batch",
  "service_id": 1,
  "service_name": "Mariadb-replica",
  "pending_before": 0,
  "pending_after": 0,
  "batched_after": 101,
  "batch_count_before": 2,
  "batch_count_after": 2,
  "build_result": {
    "ok": true,
    "message": "No pending outbound events for service 'Mariadb-replica'",
    "replication_service_id": 1,
    "batch_created": false,
    "event_count": 0
  },
  "transport_batch": {
    "id": null,
    "batch_uid": "",
    "status": null,
    "event_count": 0,
    "first_event_id": null,
    "last_event_id": null,
    "checksum": ""
  }
}
JSON_RESULT_END

============================================================
SRDF TRANSPORT COMMAND - END
============================================================
```

SRDF TRANSPORT

Bash

```
python manage.py srdf_transport --action build_batch --service-id 1
python manage.py srdf_transport --action send_batch --batch-id 12
python manage.py srdf_transport --action cycle --service-id 1
```

Bash

```
python manage.py srdf_transport --action build_batch --service-id 1
python manage.py srdf_transport --action send_batch --batch-id <ID_BATCH>
```

SRDF TRANSPORT BUILD BATCH

```
============================================================
SRDF TRANSPORT COMMAND - START
============================================================
action=build_batch
service_id=1
batch_id=None
limit=100
strict_send=False

[BUILD] Starting build_batch
[BUILD] service=Mariadb-replica id=1
[BUILD] pending_before=0
[BUILD] batch_count_before=2
[BUILD] pending_after=0
[BUILD] batched_after=101
[BUILD] batch_count_after=2
[BUILD] no batch created

============================================================
SRDF TRANSPORT COMMAND - FINAL SUMMARY
============================================================
ok=True
message=No pending outbound events for service 'Mariadb-replica'
duration_seconds=0.034

JSON_RESULT_BEGIN
{
  "ok": true,
  "message": "No pending outbound events for service 'Mariadb-replica'",
  "action": "build_batch",
  "service_id": 1,
  "service_name": "Mariadb-replica",
  "pending_before": 0,
  "pending_after": 0,
  "batched_after": 101,
  "batch_count_before": 2,
  "batch_count_after": 2,
  "build_result": {
    "ok": true,
    "message": "No pending outbound events for service 'Mariadb-replica'",
    "replication_service_id": 1,
    "batch_created": false,
    "event_count": 0
  },
  "transport_batch": {
    "id": null,
    "batch_uid": "",
    "status": null,
    "event_count": 0,
    "first_event_id": null,
    "last_event_id": null,
    "checksum": ""
  }
}
JSON_RESULT_END

============================================================
SRDF TRANSPORT COMMAND - END
============================================================
```

srdf_apply_inbound

appliquer sur la base cible MariaDB

```bash
python manage.py srdf_apply_inbound --service-id=1
```

La commande existe déjà et appelle `apply_inbound_events_once(...)`.

Ce que tu dois vérifier

Dans la stdout :

- `processed > 0`
- `applied > 0`

Puis en SQL :

```sql
SELECT id, status, applied_at, last_error
FROM srdf_inboundchangeevent
ORDER BY id DESC
LIMIT 20;
```

Attendu :

- `status = applied`

Et surtout, dans la **base cible MariaDB**, la ligne doit avoir réellement changé.

Cas “Failed Object” and “Full initial Resynchro”

```bash
python manage.py srdf_apply_inbound --service-id=1 --retry-failed
```

Et SRDF retravaillera aussi les `failed`.

```bash
python manage.py srdf_bootstrap_table --service-id=1 --database-name=oxygen_dl
```

srdf_bootstrap_service

EXEMPLES D'UTILISATION

1 Bootstrap complet d'une DB

Bash

```
python manage.py srdf_bootstrap_service \
    --service-id=1 \
    --database-name=oxygen_db6
```

2 Seulement certaines tables

Bash

```
--tables=users,orders,products
```

3 Pattern SQL

Bash

```
--like=accounts_%
```

EXEMPLES D'UTILISATION

4 Exclusion

Bash

```
--exclude=django_migrations,srdf_*
```

5 Mode safe prod

Bash

```
--no-truncate-target
```

6 Mode rebuild complet

Bash

```
--recreate-table
```

AMÉLIORATIONS FUTURES

- `--parallel=N` (multi-thread bootstrap)
- `--dry-run`
- `--progress %`
- stockage historique bootstrap
- exclusion automatique tables SRDF
- snapshot + checksum validation

bootstrap complet, exclusions auto SRDF/Django

bootstrap complet, exclusions auto SRDF/Django

Bash

```
python manage.py srdf_bootstrap_service --service-id=1 --database-name=oxygen
```

dry-run

Bash

```
python manage.py srdf_bootstrap_service --service-id=1 --database-name=oxygen
```

validation snapshot/checksum

Bash

```
python manage.py srdf_bootstrap_service --service-id=1 --database-name=oxygen
```

multi-thread

Bash

```
python manage.py srdf_bootstrap_service --service-id=1 --database-name=oxygen
```

srdf_capture_agent

Capture Agent : Fonctions de base

Exécution continue

```bash
python manage.py srdf_capture_agent --service-id=1 --interval-seconds=5 --lim
```

Avec initialisation au démarrage

```bash
python manage.py srdf_capture_agent --service-id=1 --initialize-on-start --fo
```

Run contrôlé pour test

```bash
python manage.py srdf_capture_agent --service-id=1 --max-iterations=10 --inte
```

srdf_pipeline_agent

FONCTIONS DE BASE

✅ 1. Orchestration complète

- capture → build → send
- exactement ton pipeline SRDF

✅ 2. Lock distribué

Grâce à :

Python

Exécuter

```python
acquire_lock(lock_name)
```

👉 basé sur ton modèle `ExecutionLock` models

👉 évite double agent / cron overlap

✅ 3. Résilience production

- try/except global
- audit DB (`AuditEvent`)
- erreurs non bloquantes sur send

COMMANDES DU CRON

Run simple test

```bash
python manage.py srdf_pipeline_agent --service-id=1 --once
```

Mode daemon

```bash
python manage.py srdf_pipeline_agent --service-id=1 --loop --interval=2
```

COMPTE RENDU D'EXECUTION

```
================================================
SRDF PIPELINE AGENT START
================================================
service=Mariadb-replica id=1
interval=2s limit=100

[PIPELINE] cycle start

                    Before using MARIADB 10.5.0 and MYSQL 8.0.14 versions,
                    use python-mysql-replication version Before 1.0 version
[CAPTURE] captured=100 skipped=0 ignored_older=13151
[BUILD] batch_id=7 batch_uid=9bc24759-99f5-4d41-9666-c3f61747f849
[SEND] batch_id=7 sent status=sent
[APPLY] processed=100 applied=1 failed=99
[PIPELINE][OK] cycle_duration=6.92s
```

srdf_monitor_status

ACK - CHECKPOINT ET MONITORING

1. ACK réel

Le target appelle maintenant le source pour confirmer le batch.
Le source passe le `TransportBatch` en `acked` et marque aussi les outbound events en `acked`.

2. Checkpoints étendus

Tu as maintenant :

- `capture`
- `shipper`
- `applier`

en utilisant ton modèle existant `ReplicationCheckpoint`.

3. Monitoring opérable

Avec :

```bash
python manage.py srdf_monitor_status --service-id=1
```

Tu peux voir :

- backlog outbound
- backlog inbound
- batches par statut
- checkpoints courants

COMPTE RENDU EXECUTION

```
============================================================
SRDF MONITOR STATUS
============================================================
service        : Mariadb-replica (id=1)
source_node    : mysql-source
target_node    : mariadb-target

---- OUTBOUND ----
pending : 8
batched : 0
sent    : 0
acked   : 0
failed  : 0
dead    : 0

---- INBOUND ----
received : 0
applied  : 0
failed   : 0
dead     : 0

---- BATCHES ----
ready  : 0
sent   : 0
acked  : 0
failed : 0

---- CHECKPOINTS ----
capture  log_file=mysql-bin.000002 log_pos=90512625 updated_at=2026-04-06 09:18
02.647000+00:00

============================================================
```

ACK : Tests à faire

Test 1 — envoyer un batch

```bash
python manage.py srdf_pipeline_agent --service-id=1 --once
```

Test 2 — vérifier ACK

Dans admin / SQL :

- `TransportBatch.status` doit pouvoir passer à `acked`
- `OutboundChangeEvent.status` doit pouvoir passer à `acked`

Test 3 — monitoring

```bash
python manage.py srdf_monitor_status --service-id=1
```

srdf_run_bootstrap_plan

CREATION DES OBJETS DANS ADMIN

Cas 1 — database complète

- `scope_mode = database`
- `source_database_name = oxygen_db6`
- `include_tables_csv = ""`
- `exclude_tables_csv = ...`
- `execution_order = 10`

Cas 2 — tables ciblées

- `scope_mode = tables`
- `source_database_name = oxygen_db6`
- `include_tables_csv = users,orders,invoices`
- `execution_order = 20`

EXECUTION DU CRON

Un plan

```bash
python manage.py srdf_run_bootstrap_plan --plan-id=1
```

Tous les plans actifs

```bash
python manage.py srdf_run_bootstrap_plan --all-enabled
```

Dry run

```bash
python manage.py srdf_run_bootstrap_plan --all-enabled --dry-run
```

CE QUE ÇA T'APPORTE

8) CE QUE ÇA T'APPORTE

- une **vraie phase de synchronisation initiale pilotée par l'admin**
- plusieurs plans ordonnés
- database complète ou listes de tables
- exclusions
- réexécution simple
- historique du dernier run
- base propre pour la phase "préliminaire" SRDF

srdf_daemon

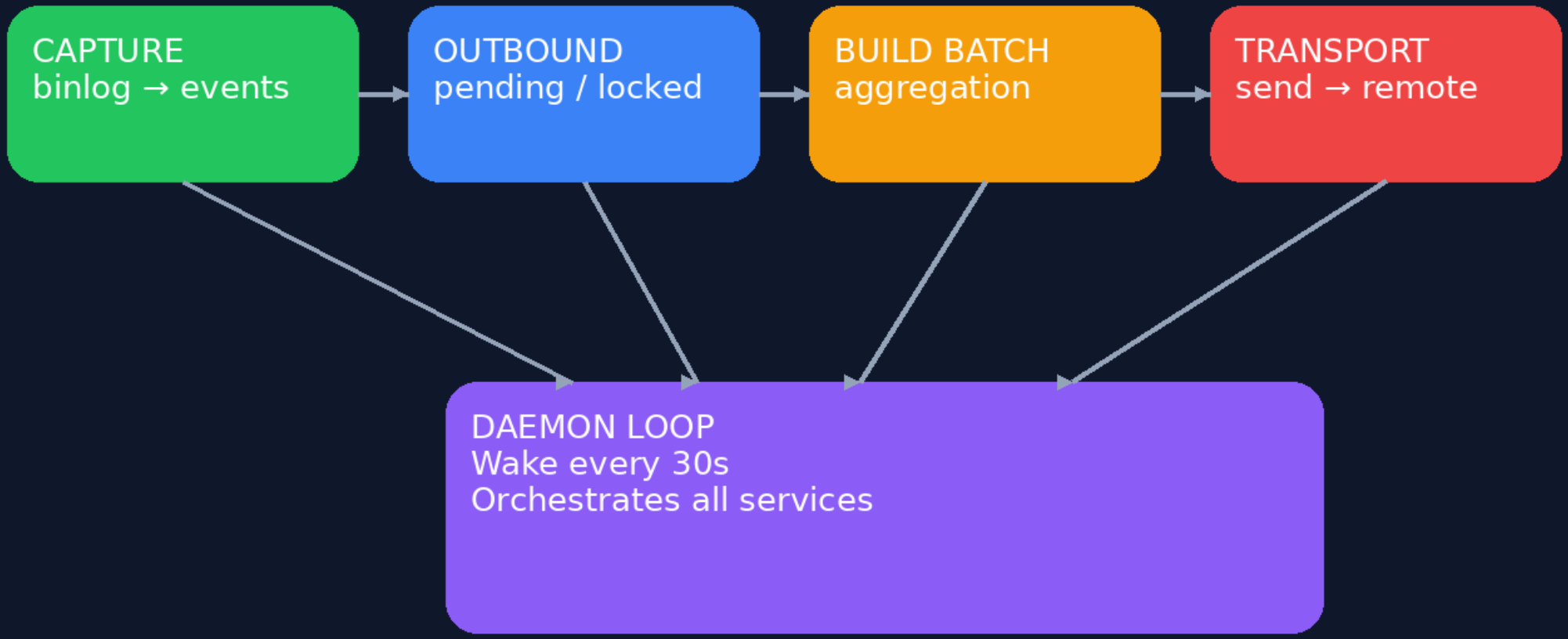
SYSTEMD / SERVICES FOR SRDF

Le but :

- **un seul daemon central**
- boucle permanente
- pilotage de **tous** les services SRDF
- pause / resume / stop
- mode status
- un seul point d'entrée Django
- capture déjà branchée via `capture_engine.py` capture_engine
- orchestrator déjà présent comme base solide srdf_orchestrator

AGENT / DAEMON (systemd)

SRDF DAEMON ORCHESTRATOR



Comment le processus SRDF doit fonctionner

Processus cible unique

1. `systemd` lance un seul service

Par exemple :

- `srdf_orchestrator_daemon.service`

2. Ce service lance une seule commande Django

Par exemple :

- `python manage.py srdf_orchestrator_daemon`

3. Cette commande instancie `SRDFOrchestrator`

Puis appelle :

- `run_forever()`

À chaque cycle

L'orchestrator :

- charge les services actifs
- ignore les services pausés
- ignore les services stoppés
- prend le lock service
- exécute les phases
- met à jour runtime + phase execution + cron execution
- relâche le lock

Règles de base

5. Chaque phase peut elle-même créer son “run cron”

Exemple capture :

- `capture_engine.capture_changes()`
- crée `SRDFCronExecution`
- locke la capture
- exécute le moteur binlog
- retourne `return_code`

6. Le daemon gère les états opérateur

Via `SRDFServiceRuntimeState` :

- `is_paused=True` → on saute le service
- `stop_requested=True` → on termine proprement puis on met `idle/stopped`
- `next_phase` / `current_phase` → pour resume ciblé
- `last_return_code` → lecture claire du dernier état.

Plan de développement du Daemon

On décide que :

- `SRDFOrchestrator` = le cerveau
- `capture_engine.py` = l'adaptateur de phase capture
- `srdf_capture_mariadb_binlog.py` = outil manuel / diagnostic / run forcé
- `srdf_daemon.py` = à remplacer par une enveloppe minimale qui appelle `SRDFOrchestrator.run_forever()`, ou à supprimer

Et on ajoute enfin :

- `pause`
- `resume`
- `stop_requested`
- un vrai command/daemon unique
- compatibilité `systemd`
- status clair dans l'admin

SRDF DAEMON : Les commandes

Démarrer le daemon

```bash
python manage.py srdf_daemon
```

Un seul cycle

```bash
python manage.py srdf_daemon --once
```

Un seul service

```bash
python manage.py srdf_daemon --service SRDF-MARIADB-PRIMARY
```

Pause d'un service

```bash
python manage.py srdf_daemon --pause-service SRDF-MARIADB-PRIMARY
```

SRDF DAEMON : LES COMMANDES

Resume d'un service

```bash
python manage.py srdf_daemon --resume-service SRDF-MARIADB-PRIMARY
```

Resume à partir d'une phase

```bash
python manage.py srdf_daemon --resume-service SRDF-MARIADB-PRIMARY --resume-from-phase captur
```

Demande d'arrêt propre d'un service

```bash
python manage.py srdf_daemon --stop-service SRDF-MARIADB-PRIMARY
```

Afficher le status runtime

```bash
python manage.py srdf_daemon --status
```

DIAGRAMME DU SERVICE DAEMON

- SERVICE / DAEMON:
  - **BOUCLE EN PERMANENCE**
    - Pilotage par les arguments classique des System/Services :
      - --start , --stop , --restart, --check , --status et --resume
  - **REVEIL TOUTES LES 30 SECONDES**
    - 1. **CHECK** SI DES CAPTURES D'UPDATES PEUVENT ËTRE LANCEES :
      - **srdf_capture_maridb_binlog** (*Running, arret, erreurs, Pending, Queued, ...)*
      - Renvoi de code retour/Signal et Update des Tables SQL de controle d'execution des Cron
    - 2. **CHECK** SI DES UPDATES ONT ETE DETECTES ET STOCKES DANS « **OutboundChangeEvent** »
      - EXECUTION : **srdf_transport** : Action = « **build_batch** » ➔ **Batch-ID=Nnnnn**
    - 3. **CHECK** SI DES « Batch Transport » ont été générés et prêts pour le transport ➔ Serveur Remote
      - EXECUTION : **srdf_transport** : Action = « **send_batch** » **Batch Id = Nnnnn (*Balayage des Batch*)**
    - **4. CHECK** des retours d'execution des services d'apply sur le serveur Remote
      - Controle de la File d'attente
      - Marquer les Updates comme « Appliqués »

POINTS CRITIQUES du daemon srdf

- ✓ paramètres en entrée → config service + runtime overrides
- ✓ codes retour → struct result (ok / retry / fatal)
- ✓ enchaînement → next_phase
- ✓ dedup → phase dédiée
- ✓ compression → phase dédiée
- ✓ encryption → phase dédiée
- ✓ stop / resume → state persisted
- ✓ reprise → state machine
- ✓ crash safe → checkpoint DB
- ✓ retry → scheduler interne
- ✓ logs → par phase
- ✓ multi-service → boucle

ORCHESTRATEUR GENERAL SRDF

👉 C'est un **ENGINE / SCHEDULER / ORCHESTRATOR** avec :

Pipeline réel

1. CAPTURE
2. DEDUP
3. COMPRESS
4. ENCRYPT
5. QUEUE
6. TRANSPORT
7. ACK
8. APPLY
9. CHECKPOINT

DAEMON CONTRAINTES REELLES

+ contraintes

- chainage strict des phases
- codes retour
- gestion erreurs fine
- retry / backoff
- stop / resume
- reprise sur crash
- idempotence
- verrouillage
- journalisation
- multi-service

LA BONNE ARCHITECTURE

1. Un STATE MACHINE ENGINE

Chaque service SRDF est dans un état :

```
IDLE
CAPTURE
DEDUP
COMPRESS
ENCRYPT
QUEUE
TRANSPORT
WAIT_ACK
APPLY
CHECKPOINT
ERROR
PAUSED
```

LES REGLES DE BASE DE L'ORCHESTRATEUR

- 1. Un STATE MACHINE ENGINE
- 2. Chaque phase retourne un RESULT STRUCTURÉ
- 3. Le daemon devient un ORCHESTRATEUR D'ÉTATS

- SERVICES PREVUS DE L'ORCHESTRATEUR SRDF :
  - srdf engine
  - capture Handler
  - Dedup Handler
  - Compress Handler
  - Encrypt Handler
- SRDF Daemon

CE QUI MANQUE ET RESTE A FAIRE

1. Le modèle `SRDFServiceState`

- phase
- last_success_phase
- retry_count
- last_error
- checkpoint_id

2. Le modèle `SRDFExecutionLog`

- phase
- durée
- volume
- erreurs

3. Le système de retry intelligent

- exponentiel
- max attempts
- dead-letter

4. Le STOP / RESUME propre

- flag pause
- resume depuis phase exacte

5. Le scheduler adaptatif

- sleep dynamique
- priorité services

BOOSTRAP

SRDF - BOOTSRAP – INITIALISATION

- Étape 1 — choix du plan
- Étape 2 — création d’une exécution
- Étape 3 — construction des work items
- Étape 4 — exécution table par table
- Étape 5 — consolidation

ENCHAINEMENT DES RUN 1-4

Étape 1 — choix du plan

On choisit un `BootstrapPlan` :

- soit par nom
- soit par ID
- soit via admin
- soit via UI/dashboard
- soit plus tard via daemon

Étape 2 — création d'une exécution

Le runner crée une ligne `BootstrapPlanExecution` en statut `running`.

Étape 3 — construction des work items

Le moteur décide quelles tables traiter :

- soit depuis `BootstrapPlanTableSelection`
- soit en découvrant les tables automatiquement dans la base source
- soit, en replay, depuis une exécution précédente

Étape 4 — exécution table par table

Pour chaque table :

- création d'un `BootstrapPlanExecutionItem`
- mise à jour du `BootstrapPlanTableSelection` si applicable
- appel à `bootstrap_table_once()`
- stockage du résultat

BOOTSTRAP - CONSOLIDATION

Étape 5 — consolidation

Quand tout est terminé :

- mise à jour du plan
- mise à jour de l'exécution globale
- calcul des compteurs
- journalisation globale

BOUCLE DE DETECTION DE RUN

boucle

- chercher les plans activés
- vérifier s'ils doivent être lancés
- poser un lock
- appeler `run_bootstrap_plan_once(...)`
- capter les erreurs
- attendre X secondes
- recommencer

Pourquoi c'est bien

Parce que :

- simple à déployer
- facile à lancer en service Linux
- cohérent avec Django
- pas besoin de rajouter Celery tout de suite

guide de fonctionnement

Niveau 1 — Configuration

- créer `Node`
- créer `ReplicationService`
- configurer la connexion source/cible
- cataloguer databases/tables
- créer `BootstrapPlan`

Niveau 2 — Exécution manuelle

- run normal
- dry-run
- suivi dans admin
- lecture des erreurs

Niveau 3 — Historique

- lire `BootstrapPlanExecution`
- lire `BootstrapPlanExecutionItem`
- analyser les lignes copiées
- identifier les tables en échec

Niveau 4 — Replay

- replay last
- replay failed
- replay failed only

Niveau 5 — Automatisation

- daemon
- boucle de scrutation
- lock
- logs
- restart systemd

Niveau 6 — Exploitation prod

- supervision
- alertes
- accusé réception source
- retry automatique
- politique de reprise

LOGIQUE

La suite logique

La suite logique n'est pas de continuer à empiler des petits crons.

La suite logique, maintenant, c'est de développer :

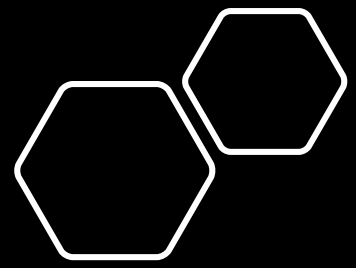
le vrai daemon SRDF source

avec :

- boucle infinie
- interval configurable
- capture
- build
- send
- logs
- lock
- systemd-ready

Et ensuite :

le daemon applier côté cible

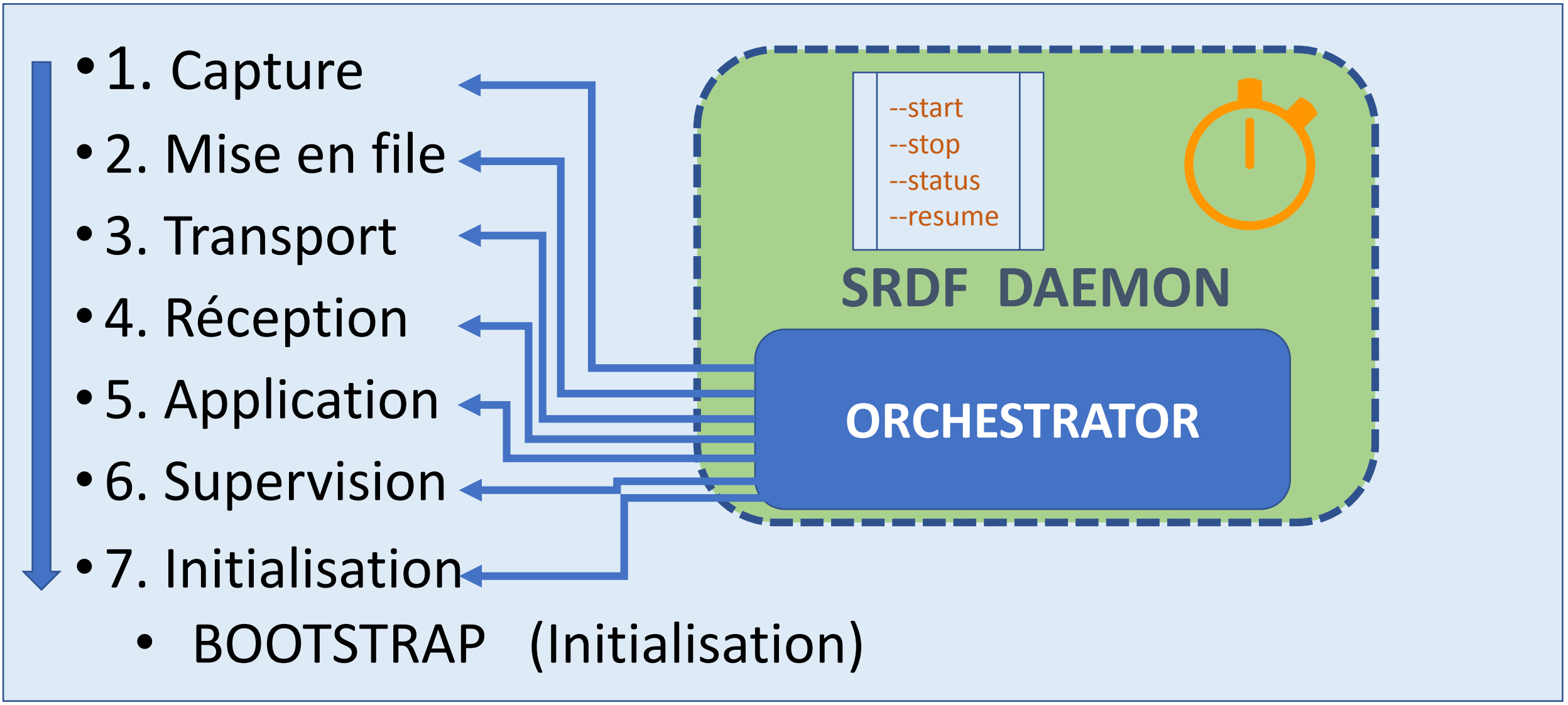


Les daemon et agents (cible/Target)

- daemon source
- daemon cible
- systemd
- boucle continue
- supervision

VISION GLOBALE SRDF

Enchaînement global SRDF



- BOOTSTRAP (Initialisation)

FONCTIONNALITES PREVUES EN VERSION 2.0

SRDF EN MODE PARALLELE

- ON DOIT ENVISAGER DE POUVOIR EXECUTER PLUSIEURS SESSIONS DE REPLICATIONS DE DATA EN PRALLELE. CELA EN FONCTION DES CAPACITES DES SERVEURS; ECU / VCU / NTWORK BANDWITH

INITIALISER A MINIMA LE SERVER REMOTE

- Imaginer un code Python standalone, sans SQL sans Django pouvant :
  - Installer le SQL Engine (Mysql, MariaDB, PostgreSQL)
  - Installer les services et serveurs système Linux minimum requis
  - Installer l'environnement SRDF :
    - Code SRDF (git, repo, ...)
    - Application Django
    - Addon requis pour SRDF
    - Installer Supervisor
    - Installer les services/Daemon pour accueillir les demandes d'Updates et accusé réception
- Un package/Script minimum pour rendre le serveur distant opérationnel :
  - Nginx
  - DNS
  - GIT

A FAIRE EN V2.0

- pas de transaction multi-events globale
- pas encore de `status="applying"` intermédiaire
- pas de rollback cross-event
- pas encore de mapping avancé source/cible
- pas encore de quoting spécial SQL complexe
- pas encore de gestion DDL

initial sync complète

La vraie **initial sync complète** :

- introspection schéma source
- création tables/index/contraintes côté cible
- export/import initial des données
- checkpoint de départ cohérent
- puis passage en mode delta

Ça, c'est une **phase à part entière**.

PREPARE NEW DATABASE

Brique unitaire : prepare target database

avec une action du genre :

```
transport.prepare_target_db
```

ou

```
bootstrap.prepare_target_db
```

qui fera simplement :

- connexion SQL au serveur distant
- création de la DB si absente
- AuditEvent
- stdout propre

Ça reste simple, propre, et ça colle à ton besoin.

INITIAL DB SYNC

Initial sync

avec potentiellement :

- export schéma source
- création schéma cible
- dump initial des données
- import cible
- marquage checkpoint initial
- démarrage delta

Là, on commencera à se rapprocher sérieusement du comportement SRDF/EMC-like que tu évoques.

SRDF doit évoluer vers 3 axes

Axe 1 — Multi-database capture

- capter la vraie DB source
- pouvoir écouter une DB ou toutes les DB
- pouvoir exclure les schémas techniques

Axe 2 — Auto-provision côté cible

- create database if missing
- plus tard create table if missing
- plus tard ddl sync optionnel

Axe 3 — Error management professionnel

- si auto-create off → erreur normale
- erreur loggée proprement
- table dédiée d'erreurs
- audit + visibilité admin

Ce que doit être la V1 minimale qui fonctionne

Mode 1 — Full resync initiale

Quand la cible est vide ou divergente :

- lire le schéma de la table source
- créer la table cible **à l'identique utile** :
  - colonnes
  - types
  - PK
  - indexes simples
- vider/recréer la table cible selon option
- copier **toute la table** source vers target
- marquer la table comme "baseline done"

Mode 2 — Delta replication

Après la baseline :

- capturer les changements source
- batcher
- envoyer
- appliquer les deltas sur la cible

Architecture V1 corrigée

Action 1

`transport.bootstrap_table`

Entrées :

- `replication_service_id`
- `database_name`
- `table_name`
- options :
  - `drop_if_exists`
  - `truncate_if_exists`
  - `copy_data`
  - `create_indexes`
  - `baseline_only`

Action 2

`transport.full_resync_table`

Fait :

- create table if missing
- optional rebuild
- full copy source → target

Action 3

`transport.apply_inbound`

Ne traite un delta que si :

- target DB existe
- target table existe
- table baseline is ready

Sinon :

- soit auto-bootstrap si option activée
- soit erreur propre

creation des daemon/systemd

- Il Faut à minimam créer 2 services (agent/Daemon) au minimum :
- Capable de monitorer, Lancer, arréter divers services et Cron
  - Serveur Origine
    - Capture
    - Batch process
    - transport
  - Server Cible :
    - Ecoute
    - Reception des Updates
    - Execution des Updates
    - renvoi accusé réception

REMOTE ACCESS ON SERVER

- Il faut developper une **API standalone** sur le serveur Remote :
  - Capable de communiquer avec le serveur origine
  - Capable d'installer à distance n'importe quel services, soft, Databases
    - Database (Mysql, MariaDB, postgresql, ..)
    - Services tiers
    - Framework
    - Auto Install de Djang + Addon
  - Capable de s'auto-diagnostiquer et renvoyer une alarme le cas échéant
  - Dialoguer avec AWS (GCP et Azure plustard)
  - Cet API de communication « universelle » devra pouvoir s'auto-installer :
    - Avec un Script
    - Prévoir des scripts Bash ou perl
  - **Liaison encryptée**

SEQUENCES DE TESTS ET DE VALIDATION

OPERATIONS DANS L'ORDRE

- 1. Renseigner le node mariadb-target
- 2. Renseigner **Api token**
- **3.** Ensuite, ordre de test
  - Corriger mariadb-target
  - Vérifier que l'API répond, par exemple avec : node.ping
- 4. Créer un nouveau Batch :
  - python manage.py srdf_transport --action=build_batch --service-id=1
- 5. Puis envoyer ce nouveau batch :
  - python manage.py srdf_transport --action=send_batch --batch-id=<NOUVEL_ID>

PROBLEME AVEC LES BATCH « NON READY »

1. Créer un nouveau batch

Bash

```
python manage.py srdf_transport --action=build_batch --service-id=1
```

2. Identifier le nouvel `id` de batch

Dans l'admin ou en SQL :

SQL

```
SELECT id, status, batch_uid, created_at
FROM srdf_transportbatch
ORDER BY id DESC;
```

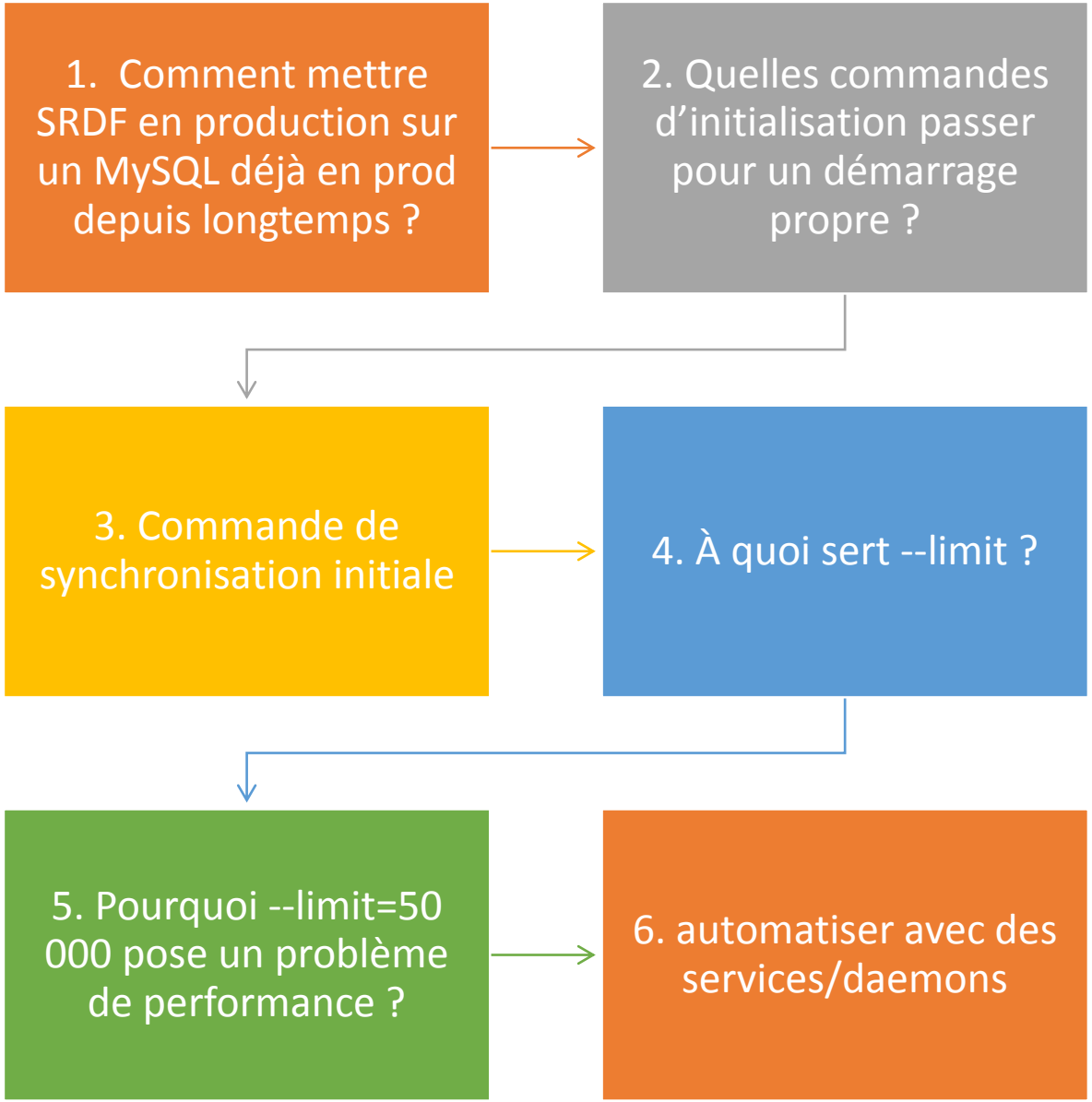
3. Envoyer ce nouveau batch

Bash

```
python manage.py srdf_transport --action=send_batch --batch-id=<NOUVEL_ID>
```

MISE EN PRODUCTION DE SRDF

AGENDA DE MISE EN PRODUCTION



ETAPE 2 : Quelles commandes d'initialisation passer pour un démarrage propre ?

Étape 1 — hard reset SRDF si environnement de test

Étape 2 — bootstrap initial des tables critiques

Étape 3 — armer le checkpoint capture au point courant

Étape 4 — vérifier

Étape 5 — démarrer la boucle delta

Étape 1 — hard reset SRDF si environnement de test

```bash
python manage.py srdf_transport --action=hard_reset --service-id=1
```

ETAPE 2 - PRECHARGEMENT DU CATALOGUE SRDF

Étape A — rafraîchir le catalogue

Commande :

```bash
python manage.py srdf_refresh_source_catalog --service-id=1
```

Ça remplit automatiquement :

- `SourceDatabaseCatalog`
- `SourceTableCatalog`

Étape B — ouvrir le plan bootstrap

Dans `Bootstrap plans`

Étape C — sélectionner les tables du catalogue

Dans l'inline `BootstrapPlanTableSelection`

Donc :

- le catalogue = technique / auto-rempli
- le plan = métier / sélection utilisateur

Étape 2.2 — bootstrap initial des tables critiques

La commande `srdf_bootstrap_table` sert justement à créer la table cible à partir du vrai `SHOW CREATE TABLE` source puis à recopier toutes les lignes. C'est le bon outil pour l'initialisation table par table. Le service de bootstrap fait :

- lecture config source/target
- `CREATE DATABASE IF NOT EXISTS`
- `SHOW CREATE TABLE`
- création cible
- copie complète des lignes source → target.

Exemple :

```bash
python manage.py srdf_bootstrap_table --service-id=1 --database-name=oxygen_db6 --table-name=
```

Etape 2.3 : Lancer le Plan de Migration

4. Lancer le plan

Toujours :

```bash
python manage.py srdf_run_bootstrap_plan --plan-id=1
```

Le runner lira maintenant les sélections relationnelles.

Étape 3 — armer le checkpoint capture au point courant

La commande de capture supporte maintenant :

- `--initialize-only`
- `--force-current-pos`

Donc :

```bash
python manage.py srdf_capture_mariadb_binlog --service-id=1 --initialize-only --force-current
```

Étape 4 — vérifier

- `SHOW MASTER STATUS`
- table `ReplicationCheckpoint`
- les deux doivent être alignés

Étape 5 — démarrer la boucle delta

Ensuite seulement :

Bash

```
python manage.py srdf_capture_mariadb_binlog --service-id=1 --limit=...
python manage.py srdf_transport --action=build_batch --service-id=1
python manage.py srdf_transport --action=send_batch --batch-id=...
python manage.py srdf_apply_inbound --service-id=1 --retry-failed
```

REGLAGES INITIAUX

- Loader les Tables qui ne doivent pas être sous la supervision de SRDF
- Django settings ou SRDF settings plus globalement genre « srdf.ini »
- SRDF Dynamic Settings (SQL Table) :

1. Baseline globale

Bash

```
python manage.py srdf_seed_setting_values --scope=global --preset=baseline
```

2. Override moteur MariaDB

Bash

```
python manage.py srdf_seed_setting_values --scope=engine --engine=mariadb --preset=baseline
```

3. Preset gros binlog pour un service précis

Bash

```
ython manage.py srdf_seed_setting_values --scope=service --service-id=1 --preset=large_binlog
```

SRDF SETTINGS BASED ON PROFILE AND ENGINE (1)

1. Baseline globale

Bash

```
python manage.py srdf_seed_setting_values --scope=global --preset=baseline
```

2. Override moteur MariaDB

Bash

```
python manage.py srdf_seed_setting_values --scope=engine --engine=mariadb --preset=baseline
```

3. Preset gros binlog pour un service précis

Bash

```
ython manage.py srdf_seed_setting_values --scope=service --service-id=1 --preset=large_binlog
```

4. Preset debug sur un service précis

Bash

```
python manage.py srdf_seed_setting_values --scope=service --service-id=1 --preset=debug_captu
```

COMPANY SETTINGS

5. Override company

Bash

```
python manage.py srdf_seed_setting_values --scope=company --company-name=IDEO-LAB --preset=la
```

6. Override client

Bash

```
python manage.py srdf_seed_setting_values --scope=client --client-name=ClientA --preset=debug
```

Bash

```
python manage.py srdf_seed_settings
python manage.py srdf_seed_setting_values --scope=global --preset=baseline
python manage.py srdf_seed_setting_values --scope=engine --engine=mariadb --preset=baseline
python manage.py srdf_seed_setting_values --scope=service --service-id=1 --preset=large_binlo
```

3) Commande de synchronisation initiale

Commande

```bash
python manage.py srdf_bootstrap_table --service-id=1 --database-name=<DB> --table-name=<TABLE
```

Ce qu'elle fait

- lit la structure source via `SHOW CREATE TABLE`
- crée la DB cible si nécessaire
- crée/recrée la table cible
- recopie tout le contenu source → cible

Important

Pour une prod réelle, il faudra probablement enchaîner cette commande sur **une liste de tables**, pas une seule.
Donc à court terme, il manque une vraie commande de niveau supérieur du style :

```bash
python manage.py srdf_bootstrap_service --service-id=1
```

qui :

- lit les tables à répliquer
- les bootstrap une à une
- produit un rapport

4) À quoi sert --limit ?

Dans la capture

Dans `capture_binlog_once(...)`, `limit` est le **nombre maximal d'événements outbound effectivement capturés** dans un run. La boucle s'arrête quand `captured_count >= limit`. binlog_capture

Donc :

- `--limit=100`
  = au plus 100 `OutboundChangeEvent` créés
- ce n'est **pas** une limite en temps
- ce n'est **pas** une limite de lignes SQL métier
- c'est une limite de **row events capturés**

Dans build/apply

Le même principe existe ailleurs :

- build batch : prend jusqu'à `limit` events pending
- apply inbound : traite jusqu'à `limit` inbound events

Pourquoi c'est utile

Parce que ça permet :

- de découper la charge
- d'éviter qu'un seul run prenne trop longtemps
- de piloter le débit

5) Pourquoi --limit=50 000 pose un problème de performance ?

Améliorations à prévoir

Pour améliorer les performances, il faudra :

A. bulk insert des `OutboundChangeEvent`

au lieu de `create()` unitaire

B. réduire le poids du `event_payload`

ou le rendre optionnel

C. timebox du run

par exemple :

- stop après N secondes
- même si `limit` pas atteint

D. daemon avec itérations fréquentes et petites

au lieu de gros crons monolithiques

automatiser avec des services/daemons

Service 1 — `srdf-capture-agent`

Boucle :

- wake up toutes les N secondes
- lance capture
- met à jour checkpoint
- loggue metrics

Service 2 — `srdf-shipper-agent`

Boucle :

- build batch si pending
- send batch
- retry failed batch
- backoff si remote KO

Service 3 — `srdf-apply-agent`

Boucle :

- apply inbound
- retry failed
- metrics / errors

Service 4 — `srdf-supervisor-agent`

Boucle :

- health checks
- queue depth
- lag
- drift detection
- reporting

Ce que doivent gérer ces daemons

Obligatoire

- start / stop / restart
- sleep interval paramétrable
- lock d'exécution
- retry
- backoff
- logs
- queue depth
- failure counters
- resume propre

Très utile

- mode dry-run
- mode bootstrap
- mode resume failed only
- time budget par cycle
- seuils d'alerte

SRDF BATCH OPERATIONS – CROSS REFERENCE

COMMON INITIAL SRDF BATCH OPERATIONS

- **2. GLOBAL RESET OF SRDF PROCESS TABLES**
  - **srdf_transport** --action=hard_reset --service-id=**Nnn**
  - *Toutes les Tables de Capture, Transport, Send, Checkpoint, ... sont RAZ*

- **3. GLOBAL RESET OF BIN LOG POSITION**
  - **srdf_capture_mariadb_binlog** --service-id=**Nnn** --initialize-only –force-current-pos

- **1. EXCLUDE SYSTEMS AND SRDF TABLES + SETTINGS**:
  - **seed_srdf_exclusions**
  - **srdf_seed_settings**

- **4. INITIAL LOADING OF SQL TABLES :**
  - **Load in Admin tool List of DataBases and/or Tables to be initially Loaded**
  - **srdf_run_bootstrap_plan**

CAPTURE SRDF BATCH OPERATIONS

- **GLOBAL CAPTURE OF SQL UPDATES**
  - **srdf_capture_mariadb_binlog** --service-id=**Nnn** **--limit=50**
  - *On analyse toutes les mises jour sur la base de donnée (Bin Log)*

- **CONSTITUTION DU BATCH D'ENVOI DU GOURPEMENT D'UPDATE**
  - **srdf_transport** --action=**build_batch** --service-id=**Nnn**
  - retourne un Nouveau Batch Id : **Bbb**

- **ENVOI DU GROUPE D'UPDATES AU SERVEUR SQL DISTANT**
  - **srdf_transport** **--action=send_batch** **--batch-id=Bbb**

- **APPLIQUER LES UPDATES SQL ENVOYES PAR LE SERVEUR ORIGINE**
  - **srdf_apply_inbound** **--service-id=Nnn**

PROCESS DE BOOTSTRAP INITIAL

-

AGENT ET MONITORING

- xxxxxxxxxxxxxxxxxxxxxxxxxxx
  - srdf_pipeline_agent --service-id=Nnn   --once

- xxxxxxxxxxxxxxxxxxxxxxxxxxx
  - srdf_monitor_status --service-id=Nnn

CURRENT PATCH IN PROGRESS

CURRENT PATCH FOR 5TH APRIL 2026

Patch 1

Auto-create database cible

Quand `srdf_apply_inbound` démarre, si option activée :

- tester existence DB
- créer DB si absente
- puis reconnecter
- puis appliquer

C'est le plus urgent, car c'est exactement ton erreur actuelle.

Patch 2

Config explicite de scope de capture

- une DB
- ou toutes les DB
- avec exclusions

Patch 3

Table dédiée ApplyErrorLog

- pour historiser proprement les erreurs SQL d'apply

Patch 4

Auto-create table if missing

- à partir des colonnes observées
- en version simple au début

PATCH POUR PLAN MIGRATION INITIAL

- 1) Barre de progression + messages par table
- 2) --plan-id numérique ou nom texte
- 3) Table SQL de log d’exécution
- 4) Rejouer la migration
- 5) Accusé de réception envoyé au serveur source
- 6) Option pour respecter le nom de la database d’origine sur la cible

1) Barre de progression + messages par table

1) Barre de progression + messages par table

Oui, indispensable.

Pour le cron/CLI, je ferais :

- affichage début plan
- affichage par database
- affichage par table
- compteur progressif du type

  `[3/18] db=crm table=customers OK rows=152340 duration=2.84s`
- résumé final

Et dans `last_result_payload`, on enrichit aussi les détails table par table.

2) --plan-id numérique **ou** nom texte

2) `--plan-id` numérique ou nom texte

Oui, très utile.

Je te propose même mieux :

- `--plan-id=12`
- `--plan-name="bootstrap crm full"`
- et éventuellement résolution intelligente si l'utilisateur passe un texte à `--plan-id`

Mais, pour rester propre, je préfère un vrai :

- `--plan-id`
- `--plan-name`

C'est plus clair et moins ambigu.

3) Table SQL de log d’exécution

3) Table SQL de log d’exécution

Oui, c’est le vrai saut de maturité.

Je te recommande 2 tables :

- `BootstrapPlanExecution`
  - 1 ligne par exécution de plan
  - date, statut, durée, totaux, options, mode dry-run, etc.
- `BootstrapPlanExecutionItem`
  - 1 ligne par table migrée
  - database, table, nombre de lignes, durée, statut, erreur, mode utilisé, etc.

C’est beaucoup plus propre que de tout laisser seulement dans `last_result_payload`.

5) Accusé de réception envoyé au serveur source

Oui, mais il faut préciser la forme.

Je vois 2 options :

- **AuditEvent côté source** via API SRDF
- ou **callback HTTP** vers `source_node.api_base_url`

Vu ton modèle `Node` contient déjà `api_base_url` et `api_token` models, on a déjà une base propre pour ça.

Je ferais :

- en fin d'exécution : POST de callback
- avec statut, compteurs, durée, détails synthétiques
- et si échec callback, ne pas faire échouer la migration elle-même, seulement logger l'erreur

6) Option pour respecter le nom de la database d’origine sur la cible

Oui, très important.

Aujourd’hui, le code `mysqldump` importe vers `target_database` défini dans la config, donc on n’est pas en “same database name” .

Je te propose une option de plan du genre :

- `preserve_source_database_name = models.BooleanField(default=False)`

Comportement :

- `False` : comportement actuel
- `True` : la base cible prend exactement `database_name` source

Et si la base n’existe pas :

- soit erreur
- soit option complémentaire `auto_create_target_database`

PATCH EN COURS V3.2 – V3.nn

Patch V3.2 — Journal SQL d'exécution

Contenu :

- modèle `BootstrapPlanExecution`
- modèle `BootstrapPlanExecutionItem`
- persistance complète des runs
- admin Django

Patch V3.3 — Replay

Contenu :

- relancer un plan
- rejouer une exécution
- option "failed only"

Patch V3.4 — Callback source + preserve database name

Contenu :

- callback HTTP d'accusé de réception
- option `preserve_source_database_name`
- adaptation `sql_copy` et `mysqldump`

1. Codes retour d’exécution normalisés

Il faut une grille stable, par exemple :

- `CAPTURE_OK`
- `CAPTURE_WARNING`
- `CAPTURE_BLOCKED_ALREADY_RUNNING`
- `CAPTURE_INIT_ONLY`
- `CAPTURE_NO_EVENTS`
- `CAPTURE_TIMEOUT`
- `CAPTURE_ERROR`
- `CAPTURE_CONTROL_LIST_OK`
- `CAPTURE_CONTROL_CHECK_OK`
- `CAPTURE_BAD_OPTIONS`

Le plus important n’est pas le nom exact, c’est que :

- le cron retourne toujours un code fonctionnel stable
- l’orchestrator puisse le lire sans parser du texte libre

2. Contrôle “déjà en cours d’exécution”

Il faut prendre un verrou dédié au cron capture, avant toute exécution métier.

Le plus simple est d’utiliser `SRDFServiceExecutionLock` avec un nom déterministe du type :

- `srdf_capture_cron:service_<id>`

C’est cohérent avec ce que tu fais déjà côté contrôle binlog et côté orchestrator. Le lock service existe déjà en base et en admin.

3. Compte rendu d’exécution dans SRDFCronExecution

Ça, la table est déjà prête. Il “suffit” maintenant de la brancher réellement au cron capture.

Chaque run doit écrire :

- `cron_name = srdf_capture_mariadb_binlog`
- `cron_type = capture`
- `status`
- `return_code`
- `error_code`
- `error_message`
- `input_params`
- `result_payload`
- `progress_payload`
- `started_at`, `finished_at`, `duration_seconds`

Le modèle supporte déjà tout ça. models

PLAN DES PATCHS 11TH APRIL 2026

Patch M

Dans `srdf_capture_mariadb_binlog.py` :

- création d'un `SRDFCronExecution` au démarrage
- acquisition d'un lock `SRDFServiceExecutionLock`
- blocage propre si déjà en cours
- mapping du résultat en `return_code`
- finalisation SQL du run

Patch N

Dans `srdf_orchestrator.py` :

- consommation propre du `return_code`
- propagation dans `SRDFServicePhaseExecution.return_code`
- mise à jour de `SRDFServiceRuntimeState.last_return_code`
- règles simples de décision selon le code retour

Donc oui :

- **1) codes retour** : pas encore bien traités
- **2) anti double-run** : pas encore branché proprement dans le cron capture
- **3) log SQL des crons** : la table existe déjà, mais le cron capture ne l'utilise pas encore réellement