Générateur de DDL pour la couche de restitution

Outil qui dérive l'intégralité du DDL de publication d'un objet vers la couche de restitution depuis l'introspection de la base. Trois interfaces sur un noyau unique, mode simulation fidèle, 203 tests sans base de données.

Résultat mesuré : passage de dix à quinze minutes à moins d'une minute par objet, avec des conventions garanties mécaniquement plutôt que respectées de mémoire.

Fiche de méthode associée : `../../docs/04-generateur-ddl-industrialisation.md`

Le problème résolu

Sur une architecture d'entrepôt en couches avec fédération inter-bases, publier un objet transformé vers la couche de restitution demande de créer deux objets : une table distante fédérée et une vue passthrough. Avec, à chaque fois :

- des conventions de nommage à respecter de mémoire ;
- des colonnes et types à recopier depuis les métadonnées ;
- des droits à redéfinir, différents de ceux de la source ;
- un commentaire de traçabilité à poser.

Toutes ces informations étaient déjà interrogeables dans la base. C'est ce constat, chiffré avant d'écrire une ligne, qui a justifié l'outil.

Architecture

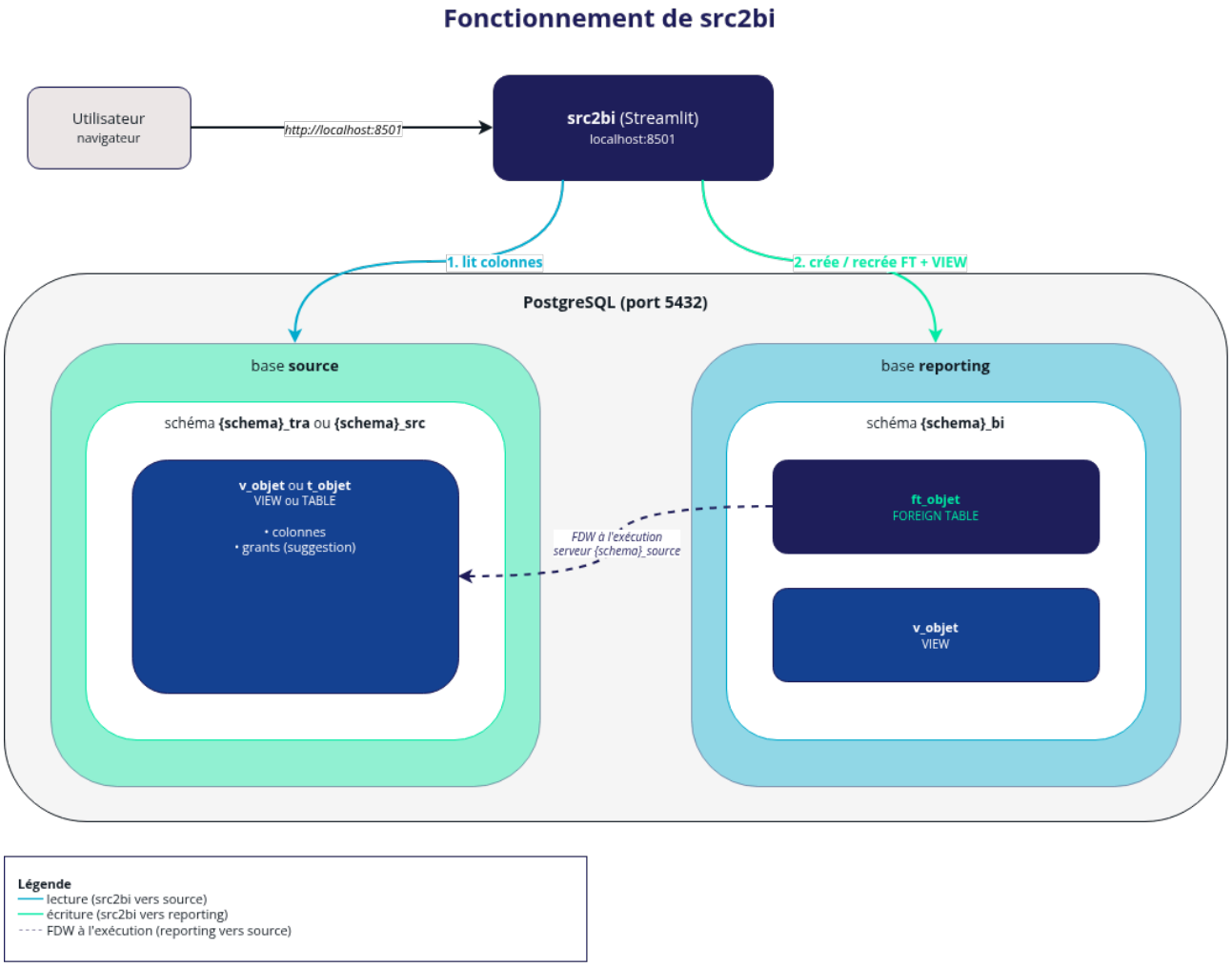


Figure 1: fonctionnement

```
product/
├── app.py                    interface graphique
├── generate_bi_objects.py    interface ligne de commande
├── mcp_server.py             serveur d'outils pour assistant
│
└── src/                      LE NOYAU : aucune interface ne le contourne
    ├── ddl.py                génération de SQL, fonctions pures
    ├── db.py                 accès aux métadonnées, paramètres liés uniquement
    └── helpers.py            logique pure sans entrée/sortie
```

Règle absolue : aucune convention de nommage, aucune construction de SQL, aucun garde-fou ne vit en dehors de `src/`. Les interfaces collectent des paramètres, appellent le noyau, présentent le résultat.

Ce qui arrive quand on ne tient pas cette règle, et qui est arrivé : le contrôle existe dans l'interface graphique, la ligne de commande tombe sur une erreur brute de la base, et l'assistant crée des objets malformés.

Ce qui vaut d'être regardé

1. Le mode simulation fidèle (`generate_bi_objects.py`, `mcp_server.py`) Le piège le plus grave du projet, et sa correction.

La version fautive construisait le DDL, l'affichait si le mode simulation était actif, puis retournait immédiatement : les contrôles d'existence, la garde de recréation et la détection de dépendances écrits plus bas ne s'exécutaient jamais. La simulation affichait un DDL de création propre, même quand l'exécution réelle aurait déclenché des suppressions ou aurait été bloquée.

La version correcte inverse l'ordre : tous les contrôles d'abord, puis la construction du DDL, puis l'affichage qui inclut les suppressions réellement prévues.

Contrat tenu : *une simulation verte garantit qu'une exécution réelle passe.* Sinon c'est du théâtre de sécurité, plus dangereux qu'aucune simulation.

2. La règle du `raise` en transaction (`mcp_server.py`)

Le bug le plus destructeur, et il était silencieux.

Un `return` à l'intérieur d'un bloc transactionnel sort sans exception, donc le gestionnaire de contexte valide la transaction, y compris les suppressions déjà exécutées. Conséquence réelle : objets détruits en production, non recréés, aucun message d'erreur.

Le test qui verrouille ce comportement (`tests/test_mcp_server.py`) ne vérifie pas le message d'erreur : il vérifie que l'exception atteint bien le gestionnaire de contexte, en inspectant l'argument reçu par la sortie du bloc.

3. La sécurité SQL à trois niveaux (`src/ddl.py`)

Niveau	Mécanisme
Validation d'entrée	liste blanche sur les identifiants, y compris ceux venant de la base
Échappement	fonction dédiée aux identifiants, fonction distincte pour les littéraux
Valeurs	paramètres liés exclusivement, jamais de concaténation

La distinction identifiant contre littéral est celle qu'on manque : les options d'un serveur de fédération sont des chaînes, pas des identifiants. Confondre les deux ouvre une injection. Un test vérifie qu'aucune apostrophe n'est laissée non doublée dans le DDL généré.

4. Les garde-fous d'écriture

Garde-fou	Ce qu'il empêche
Détection de dépendances anormales avant suppression	une cascade silencieuse qui casse des objets tiers
Droits en lecture seule à la recréation	qu'une opération de routine altère des droits déjà validés
Garde de recréation obligatoire	qu'une modification de structure semble s'appliquer sans le faire
Confirmation explicite avant suppression	avec le nombre réel d'objets impactés dans le message

Le troisième mérite une note : la table distante est créée avec une clause conditionnelle, donc idempotente mais inopérante sur un objet existant, alors que la vue est remplacée, donc opérante. Appliquer le DDL à la main sur un objet existant met les deux objets en désaccord. L'outil refuse de tourner sans le drapeau de recréation, simulation comprise.

5. La traçabilité cumulative (`src/ddl.py`)

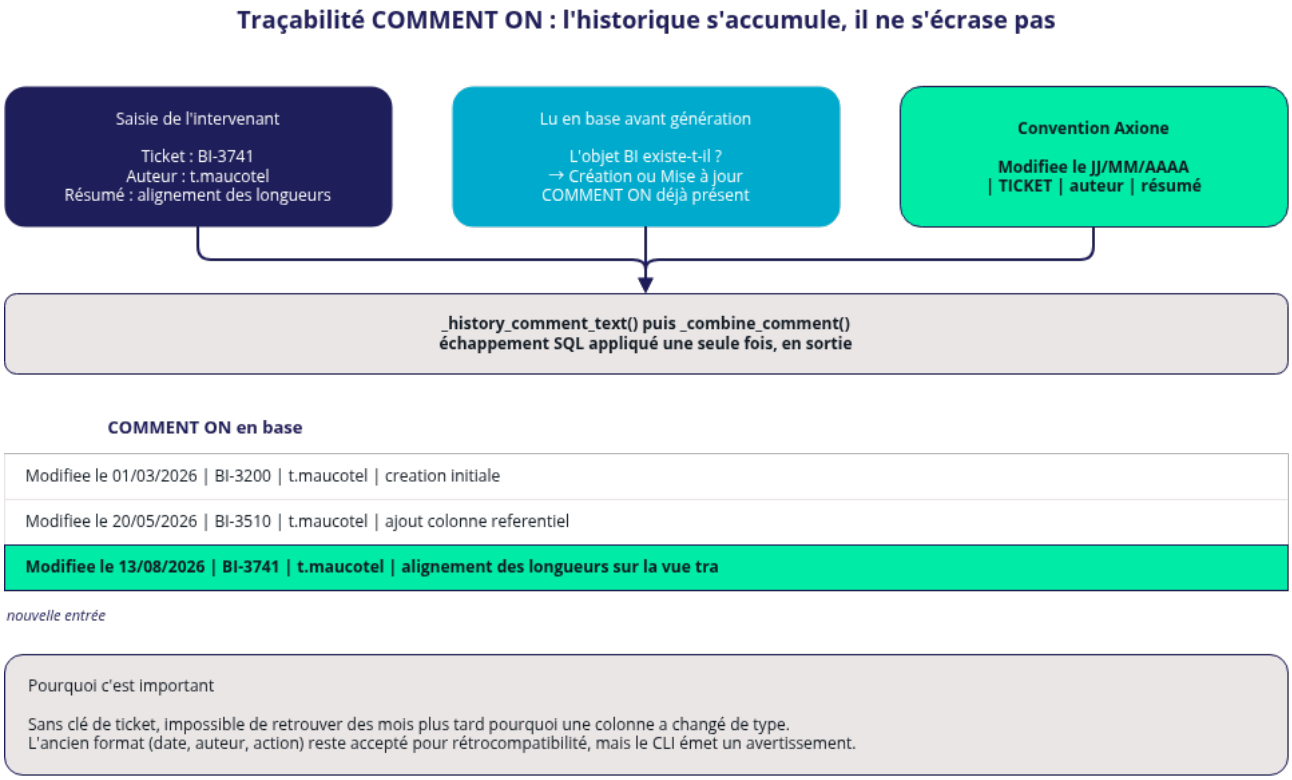


Figure 2: traçabilité

Chaque objet créé ou modifié porte son historique dans un commentaire de base, cumulatif et dédupliqué, interrogeable en SQL. L'objet est auto-documenté sans dépendre d'un wiki externe.

Détail d'implémentation qui compte : l'échappement des apostrophes se fait une seule fois, en dernier. Le faire plus tôt double les apostrophes à chaque passage.

6. Les tests sans base de données (`product/tests/`)

203 tests, l'essentiel sur des fonctions pures ou des curseurs simulés.

Couvert en priorité : chaque convention de nommage avec ses cas limites, chaque type du mapping plus un type inconnu qui doit tomber sur le repli, l'injection SQL sur les littéraux, l'annulation de transaction, et le déterminisme (deux générations produisent un texte identique, donc diffable).

7. Le serveur d'outils pour assistant (`mcp_server.py`)

Trois règles pour un outil qui écrit en production, exposé à un agent :

- mode simulation actif par défaut : l'agent doit demander explicitement l'écriture ;
- revalidation des valeurs autorisées côté serveur : le schéma déclaratif ne contraint rien ;
- réponse structurée décrivant l'action, les suppressions prévues et les contrôles passés, pour que l'agent puisse annoncer précisément ce qui va s'exécuter avant de demander un feu vert.

Le protocole d'écriture en production est écrit dans `CLAUDE.md` : valider en test, relire en production en simulation, n'exécuter qu'après feu vert humain explicite.

8. Le déploiement (`deployment/`, `distribution/`, `.gitlab-ci.yml`)

Conteneurisation multi-étapes durcie (image épinglée par empreinte, utilisateur non privilégié, contrôle de santé applicatif, télémétrie désactivée), gabarits de configuration et de découverte de services, automatisation avec attente active du contrôle de santé puis test de fumée.

Pipeline branché sur les gabarits partagés de l'organisation, avec vérification de types stricte sur le noyau et normale sur les points d'entrée, et déploiement en validation manuelle.

Fiche associée : `../../docs/13-docker-deploiement.md`

Vérification après anonymisation

Contrôle	Résultat
Tests	203 passent
Analyse de code	aucune remarque
Schémas régénérés depuis les sources anonymisées	3 images
Fichier d'exemple de configuration	noms de variables uniquement
Gabarit de déploiement	expressions de templating uniquement, aucun secret
Motifs sensibles résiduels	aucun

Ce qui a été retiré

Retiré	Raison
`.env`	secrets réels
`.mcp.json`	contenait un jeton d'accès en clair
`context.xml`	export du dépôt pour assistant, régénérable
`graphify-out/`	graphe de connaissance généré
`.claude/`	permissions locales personnelles
historique de contrôle de version	contient ce qui a été supprimé

Anonymisation des identifiants métier : les noms de rôles, schémas et bases internes ont été remplacés par des équivalents génériques et parlants, de façon cohérente dans tout le projet (un même identifiant réel donne toujours le même remplacement). Le code reste donc lisible et les tests continuent de passer.

Type	Exemple de remplacement
Rôles applicatifs	`appuser_ops`, `appuser_reporting`, `appuser_sales`
Domaines métier	`ops`, `sales`
Bases	`warehouse_prod`, `warehouse_selfservice`, `warehouse_test`