



RAPPORT DU PROJET

Plateforme d'observabilité APM

Supervision applicative et performance des données

Projet: APM-Observability
Matiere: Administration de données
Sujet: TimescaleDB
Auteur: Nawfal RAZOUK
Mathilde SOULET
Avel LE MEE MORET
Encadrant: Mouad RIYAD
Formation: 3A Genie Informatique - Option : IData
Institution: ENSMR

Rabat
29 décembre 2025

Résumé

Ce rapport s'inscrit dans le cadre du module Administration de données et décrit la conception et la mise en œuvre d'APM-Observability, une plateforme d'observabilité applicative. Après une synthèse sur TimescaleDB et le modèle des séries temporelles, nous présentons l'architecture globale : pipeline d'ingestion via API Django, stockage en hypertables, vues continues pour les KPI et exposition des métriques vers Prometheus. La partie pratique détaille la configuration en mode cluster, la génération de données, les sauvegardes et restaurations pgBackRest vers MinIO, ainsi que la validation par tests Postman/Newman. Enfin, la supervision est illustrée par des tableaux de bord Grafana et des indicateurs de performance (latence, taux d'erreur, volumétrie). L'ensemble fournit une base opérationnelle pour le suivi applicatif et la prise de décision.

Abstract

This report, developed for the Data Administration course, presents the design and implementation of APM-Observability, an application performance monitoring platform. We describe the data pipeline from ingestion to storage in TimescaleDB hypertables, the use of continuous aggregates for KPI computation, and the export of metrics to Prometheus. Operational aspects include cluster deployment, data seeding, pgBackRest backups to MinIO, and automated validation with Postman/Newman. The monitoring layer is showcased through Grafana dashboards covering availability, latency, error rate, and system health. The project delivers a practical, reproducible foundation for observability in data-intensive applications.

Table des matières

I Introduction	4
II Présentation de la base	5
II.1 Use Case de la base	5
II.2 Architecture interne de la base	6
II.3 Particularités de la base de données	6
II.4 Comparaison avec Oracle Database	6
II.5 Comparaison On-Premise et Cloud	6
III Mise en œuvre et expérimentation	8
III.1 Architecture globale	8
III.2 Flux de données	9
III.3 Démarrage et connexion	10
III.4 Prise en main	10
III.5 Cas d'usage et modèle de données	14
III.6 Déploiement en cluster	16
IV Application pratique	17
IV.1 Sauvegardes et restauration	17
IV.2 Données de test	19
IV.3 Tolérance aux pannes	19
IV.4 Optimisation	19
IV.5 Tests et validation	20
V Supervision avec Grafana	22
V.1 Présentation de Grafana	22
V.2 Intégration avec TimescaleDB	22
V.3 Configuration de la datasource TimescaleDB	22
V.4 Exemples de requêtes SQL (panels)	23
V.5 Dashboard de monitoring	24
V.6 Infrastructure et targets (Prometheus)	25
VI Conclusion et perspectives	27
A Commandes et sorties	28
B Structure du projet (extrait)	28
C Exemples de requêtes SQL	29
D Extraits de configuration	29
E Extraits de code et résultats	29
F Autres recommandations	32

Table des figures

1	Page d'accueil du site de TimescaleDB	5
2	Architecture globale (stack principale)	8
3	Flux d'ingestion et d'analyse des données	9
4	Séquence d'ingestion en mode bulk	12
5	Django Admin - liste des ApiRequest	13
6	Cas d'usage principaux de la plateforme	14
7	Diagramme de classes (modèle de données)	15
8	Topologie du cluster	16
9	Flux de sauvegarde pgBackRest vers MinIO	17
10	Bucket hot pgbackrest	18
11	Bucket cold pgbackrest-cold	18
12	Rapport Newman - Step 1 (CRUD + filtres)	20
13	Rapport Newman - Step 5 (KPIs / analytics)	21
14	Datasources Grafana (TimescaleDB et Prometheus)	23
15	Structure logique du dashboard Grafana	24
16	Dashboard Grafana (capture d'écran)	25
17	Dashboard APM Infra Overview	25
18	Dashboard APM Prometheus Targets	26
19	Prometheus /targets (état des cibles)	26

Chapitre I Introduction

L'observabilité applicative (APM) vise à comprendre le comportement d'un système à partir de ses signaux (métriques, traces et logs). L'objectif de ce rapport est de présenter une plateforme d'observabilité basée sur TimescaleDB pour le stockage des données temporelles et Grafana pour la visualisation.

Nous décrivons d'abord les concepts et particularités de la base TimescaleDB, puis la mise en œuvre technique du projet. Nous présentons ensuite les usages pratiques (sauvegarde, restauration, seeding et optimisation) avant d'aborder la supervision avec Grafana. Enfin, une conclusion synthétise les choix et propose des perspectives.

Le code source du projet est disponible sur GitHub :

<https://github.com/NawfalRAZOUK7/apm-observability>

Chapitre II Présentation de la base

TimescaleDB est une base de données open-source spécialisée dans la gestion des séries temporelles. Elle se présente sous la forme d'une extension de PostgreSQL, ce qui lui permet de bénéficier de la fiabilité et de l'écosystème PostgreSQL, tout en ajoutant des optimisations dédiées aux données horodatées (*time-series*). [1]

Elle est conçue pour gérer efficacement de très grands volumes de données insérées en continu, comme les métriques de supervision, les données IoT (Internet of Things), les logs applicatifs ou les données financières.

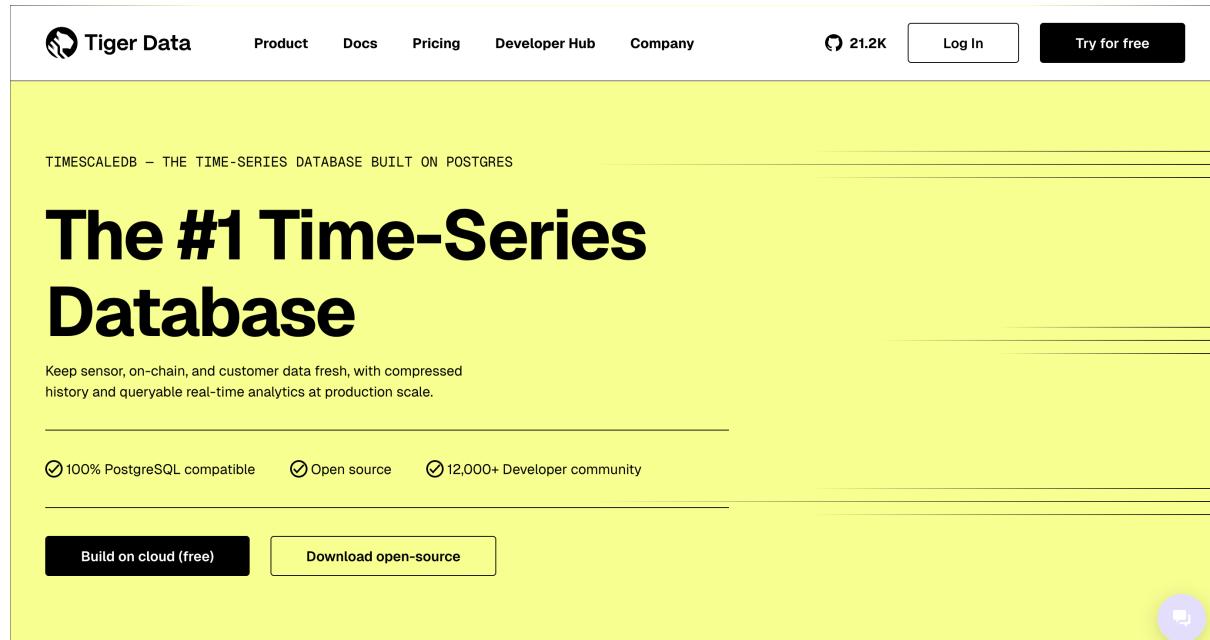


FIGURE 1 – Page d'accueil du site de TimescaleDB

II.1 Use Case de la base

TimescaleDB est utilisée par de nombreuses entreprises à l'échelle mondiale, notamment dans les domaines du monitoring, de l'IoT et de la finance. [2]

Exemples d'entreprises utilisatrices

- Comcast : supervision de réseaux et métriques de performance
- Cisco : monitoring d'équipements réseau
- IBM : gestion de données IoT
- Schneider Electric : capteurs industriels et énergie
- T-Mobile : analyse des performances réseau

Cas d'usage typiques

- Monitoring d'infrastructure et observabilité
- Internet des objets (IoT)
- Analyse financière et trading
- DevOps (Development and Operations) et métriques applicatives

II.2 Architecture interne de la base

TimescaleDB repose sur l'architecture interne de PostgreSQL et ajoute une couche spécifique pour les séries temporelles.

Fichiers et stockage

Les données sont stockées dans des *hypertables*, qui sont automatiquement découpées en *chunks* (partitions temporelles). Chaque chunk est physiquement une table PostgreSQL classique stockée sur le système de fichiers.

Processus internes

TimescaleDB utilise des processus d'arrière-plan (*background workers*) pour :

- la compression automatique des données anciennes
- la suppression des données selon des politiques de rétention
- la mise à jour des agrégations continues

L'ensemble reste conforme aux propriétés ACID (Atomicité, Cohérence, Isolation, Durabilité) grâce au moteur PostgreSQL, basé notamment sur le WAL (Write-Ahead Logging) et le MVCC (Multi-Version Concurrency Control).

II.3 Particularités de la base de données

La principale particularité de TimescaleDB est le partitionnement automatique des données temporelles via les hypertables, totalement transparent pour l'utilisateur.

Elle propose également :

- une compression columnaire très efficace (jusqu'à 90–95 %) [3]
- des fonctions temporelles avancées (ex. *time_bucket*)
- des agrégations continues pour accélérer les requêtes analytiques

II.4 Comparaison avec Oracle Database

Critère	TimescaleDB	Oracle Database
Licence	Open-source	Propriétaire
Spécialisation	Séries temporelles	Généraliste
Partitionnement temporel	Automatique	Manuel (option)
Compression	Native	Option payante
Coût	Faible	Très élevé

Oracle est une base de données généraliste adaptée aux charges OLTP (Online Transaction Processing) et OLAP (Online Analytical Processing), tandis que TimescaleDB est optimisée pour les séries temporelles avec un coût et une complexité bien moindres.

II.5 Comparaison On-Premise et Cloud

Coûts

Une solution on-premise implique des coûts initiaux élevés (matériel, administration, maintenance). Les solutions cloud réduisent ces coûts grâce à un modèle à l'usage.

Performances

- On-premise : performances maximales si l'infrastructure est bien dimensionnée
- Cloud : légère latence réseau, mais excellente scalabilité
- Amazon Aurora (service cloud PostgreSQL managé) : bonnes performances mais coût plus élevé à grande échelle

Chapitre III Mise en œuvre et expérimentation

III.1 Architecture globale

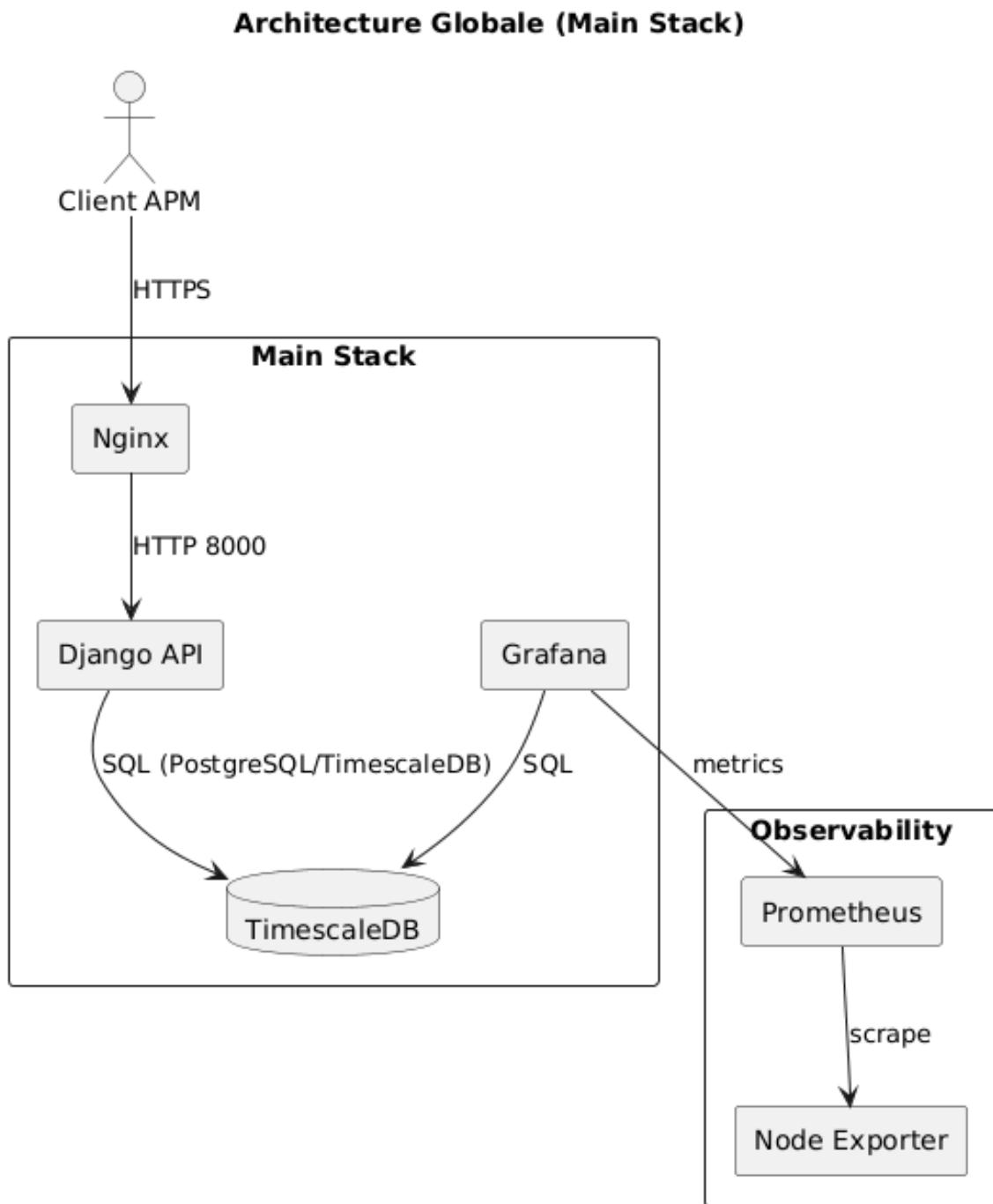


FIGURE 2 – Architecture globale (stack principale)

III.2 Flux de données

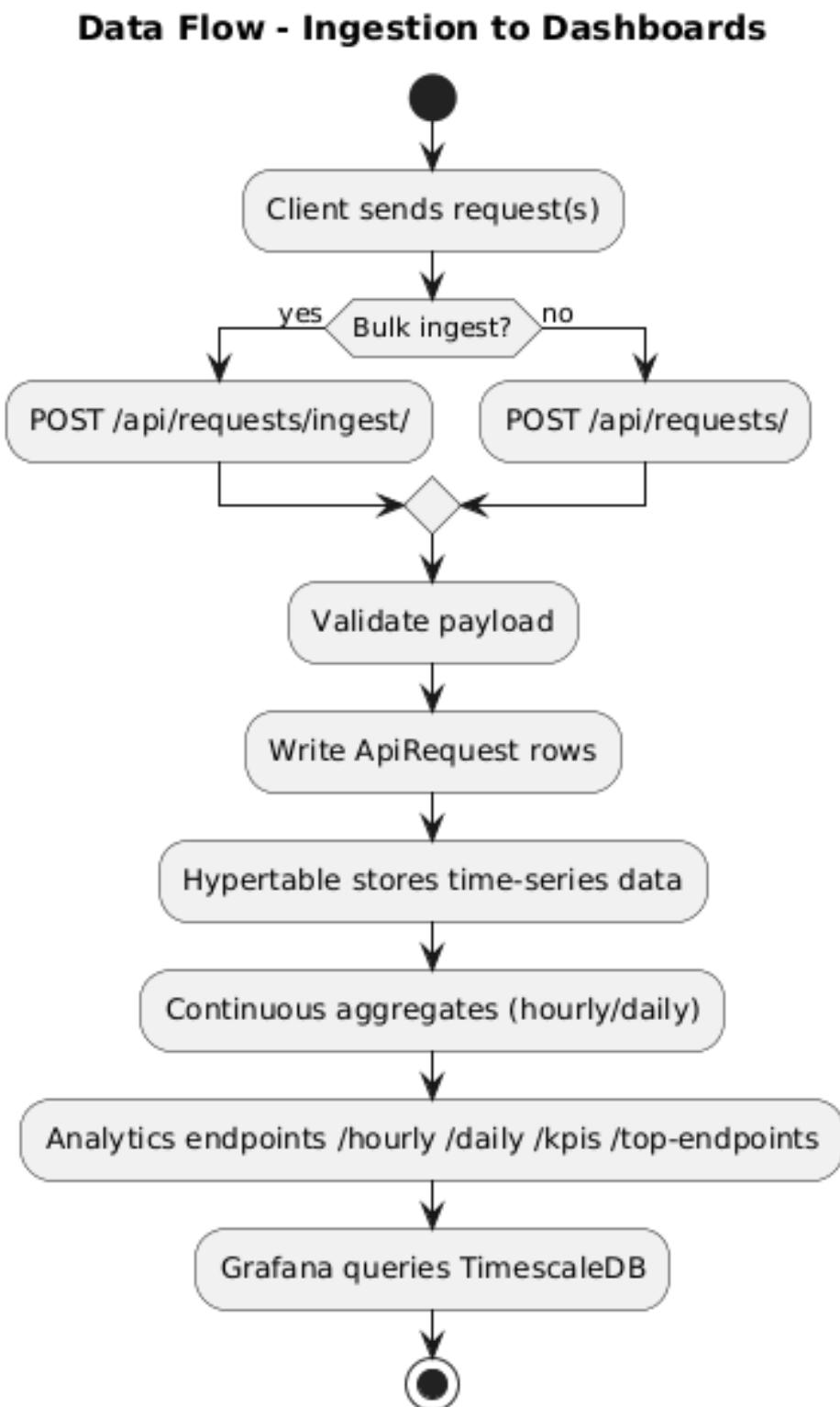


FIGURE 3 – Flux d’ingestion et d’analyse des données

III.3 Démarrage et connexion

Lancement d'une instance TimescaleDB

Le projet fournit un environnement Docker prêt à l'emploi pour lancer une instance TimescaleDB. Il suffit de copier le fichier d'exemple `.env` et de démarrer les services via Docker Compose :

```
cp .env.example .env
docker compose -f docker/docker-compose.yml up --build
```

Cette commande initialise à la fois le conteneur TimescaleDB et le backend Django.

Connexion à la base de données

Une fois le conteneur en fonctionnement, la connexion peut se faire directement depuis Docker :

```
docker compose -f docker/docker-compose.yml exec db psql -U apm -d apm
```

Il est également possible de se connecter depuis l'hôte si le port est exposé :

```
psql -h localhost -p 5432 -U apm -d apm
```

III.4 Prise en main

Création des tables

Les modèles Django sont déjà définis. La commande de migration crée automatiquement les tables nécessaires, incluant les hypertables de TimescaleDB :

```
python manage.py migrate
```

En environnement Docker, l'équivalent peut être lancé avec :

```
make docker-migrate
```

Insertion et manipulation des données

Deux modes d'ingestion sont disponibles : un CRUD unitaire et un mode bulk.

Mode unitaire (CRUD) :

```
curl -X POST http://127.0.0.1:8000/api/requests/ \
-H "Content-Type: application/json" \
-d '{
"time": "2025-12-14T12:00:00Z",
"service": "billing",
"endpoint": "/health",
"method": "GET",
"status_code": 200,
"latency_ms": 42
}'
```

Mode bulk (ingest) :

```
curl -X POST "http://127.0.0.1:8000/api/requests/ingest/?strict=false" \
-H "Content-Type: application/json" \
-d '[
  {
    "time": "2025-12-14T12:00:00Z",
    "service": "billing",
    "endpoint": "/health",
    "method": "GET",
    "status_code": 200,
    "latency_ms": 42
  },
  {
    "time": "2025-12-14T12:00:05Z",
    "service": "billing",
    "endpoint": "/pay",
    "method": "POST",
    "status_code": 500,
    "latency_ms": 180
  }
]'
```

Le paramètre `strict=true` permet d'échouer la requête si un item est invalide.

Diagramme de séquence d'ingestion

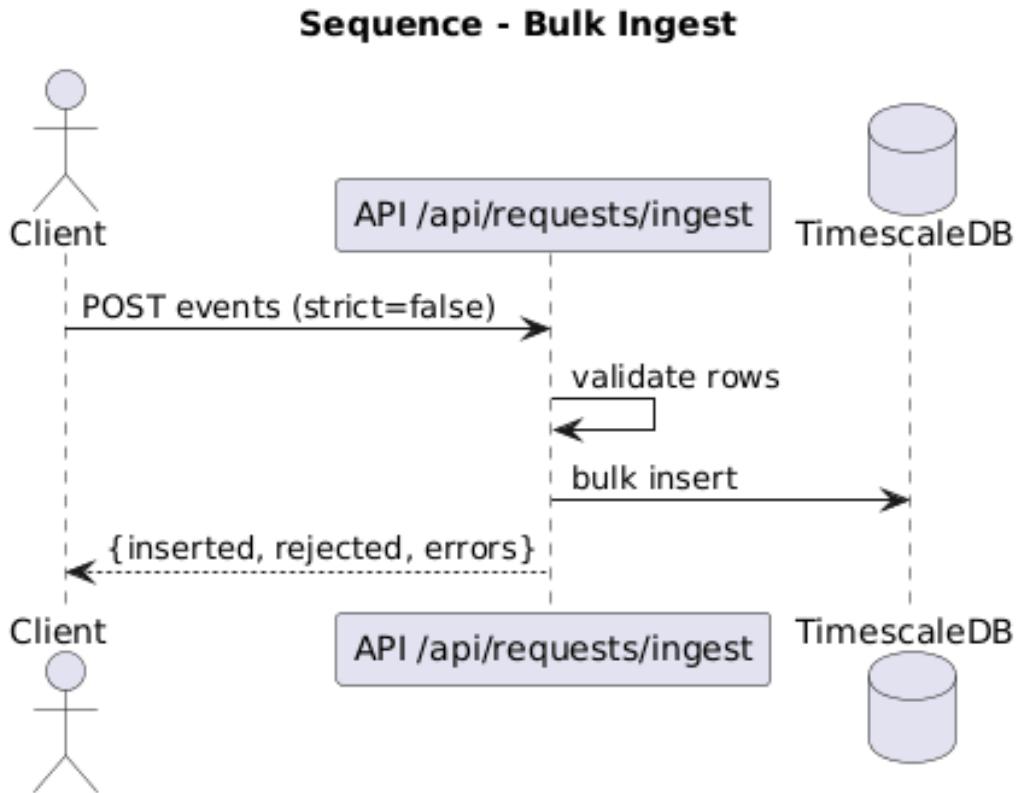


FIGURE 4 – Séquence d'ingestion en mode bulk

Les requêtes peuvent ensuite être consultées avec :

```
curl http://127.0.0.1:8000/api/requests/
```

Des endpoints d'Analytics sont aussi disponibles :

- /api/requests/hourly/
- /api/requests/daily/
- /api/requests/kpis/
- /api/requests/top-endpoints/

L'interface d'administration Django permet de consulter les requêtes ingérées (Figure 5).

	Time	Service	Method	Endpoint	Status code	Latency ms	Trace id	User ref
<input type="checkbox"/>	Dec. 28, 2025, 10:37 a.m.	api	GET	/health	204	92	c67e1ad0-5c1e-4390-afed-ae2ea4af0f0a	matthewbarnes
<input type="checkbox"/>	Dec. 28, 2025, 10:37 a.m.	auth	POST	/login	202	275	ca51eb43-1b45-4a62-a116-de518e37220d	amandalahodges
<input type="checkbox"/>	Dec. 28, 2025, 10:36 a.m.	auth	POST	/login	502	323	87ed89ae-80ee-4336-bd81-5ed42037f6b7	jonespeter
<input type="checkbox"/>	Dec. 28, 2025, 10:35 a.m.	web	GET	/home	200	237	60b2d07a-2b7d-4547-8d3c-724a0bb5c74	gphilips
<input type="checkbox"/>	Dec. 28, 2025, 10:31 a.m.	auth	POST	/login	200	398	e73ad284-cfd4-401b-9c1e-cc142e9f9161	ryan32
<input type="checkbox"/>	Dec. 28, 2025, 10:30 a.m.	api	GET	/orders	202	428	df3f44c8-8d60-4f71-b171-62b5a0757ed	-
<input type="checkbox"/>	Dec. 28, 2025, 10:30 a.m.	api	GET	/search	204	284	9b222195-a8ab-4bda-8162-b91b07af6acc0	-
<input type="checkbox"/>	Dec. 28, 2025, 10:28 a.m.	web	GET	/search	204	99	48e6fa04-9169-4513-8ac4-ec7dc0d0306	paulsmith
<input type="checkbox"/>	Dec. 28, 2025, 10:27 a.m.	web	GET	/home	200	193	98aae3937-226f-485e-95c2-d4bde4f407ab4	yjenkins
<input type="checkbox"/>	Dec. 28, 2025, 10:26 a.m.	web	GET	/search	201	379	a12a446b-5357-4e8d-9352-8f02bd9780f	alec87
<input type="checkbox"/>	Dec. 28, 2025, 10:25 a.m.	auth	POST	/login	202	162	28a276c3-f2a7-4891-bcfb-325e99c8874c	-
<input type="checkbox"/>	Dec. 28, 2025, 10:24 a.m.	api	GET	/orders	202	214	b145e4eb-2e2b-4e9f-b458-d8b3136cc1f	patrickhensley
<input type="checkbox"/>	Dec. 28, 2025, 10:23 a.m.	api	GET	/search	201	136	3c049e62-4047-4caf-b410-8d2fa4544abe7	mwillis
<input type="checkbox"/>	Dec. 28, 2025, 10:21 a.m.	web	GET	/home	200	27	7b880e53-74fa-4c5f-90b9-7f634de7019b	-
<input type="checkbox"/>	Dec. 28, 2025, 10:21 a.m.	web	GET	/home	202	276	3d2dc4fe-cc5f-4e6a-bcce-fea0126af97	khardy
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	api	POST	/orders	201	165	9e75fc1a-1e9b-4b23-9d61-81c275c488fa	tcook
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	web	GET	/home	200	354	804a7290-h252-44ac-9525-90c7d140a29	yulouglas
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	api	GET	/health	200	164	e1873e54-98c7-462c-9ac0-24aacb1124cc	xgoodwin
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	api	GET	/orders	204	318	f550384f-6b15-49fc-b51b-d7c00499c2eb	ericawilliams
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	auth	POST	/login	201	310	7cb452c-cafe-45b6-b9fa-46d9a066d647	hschneider
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	api	GET	/health	204	74	8a4860a3-54c5-4d04-8501-c3781e86d4c7	mcclaincarolyn
<input type="checkbox"/>	Dec. 28, 2025, 10:20 a.m.	auth	POST	/login	204	84	8ad65ba1-98da-47d1-b489-f41b35f3511c	-

FIGURE 5 – Django Admin - liste des ApiRequest

Fonctionnalités spécifiques de TimescaleDB

Le projet utilise les particularités de TimescaleDB pour :

- Créer des hypertables pour le partitionnement automatique.
- Mettre en place des agrégats continus (hourly/daily) pour accélérer les requêtes analytiques.
- Assurer un fallback sur PostgreSQL standard si TimescaleDB n'est pas disponible.

III.5 Cas d'usage et modèle de données

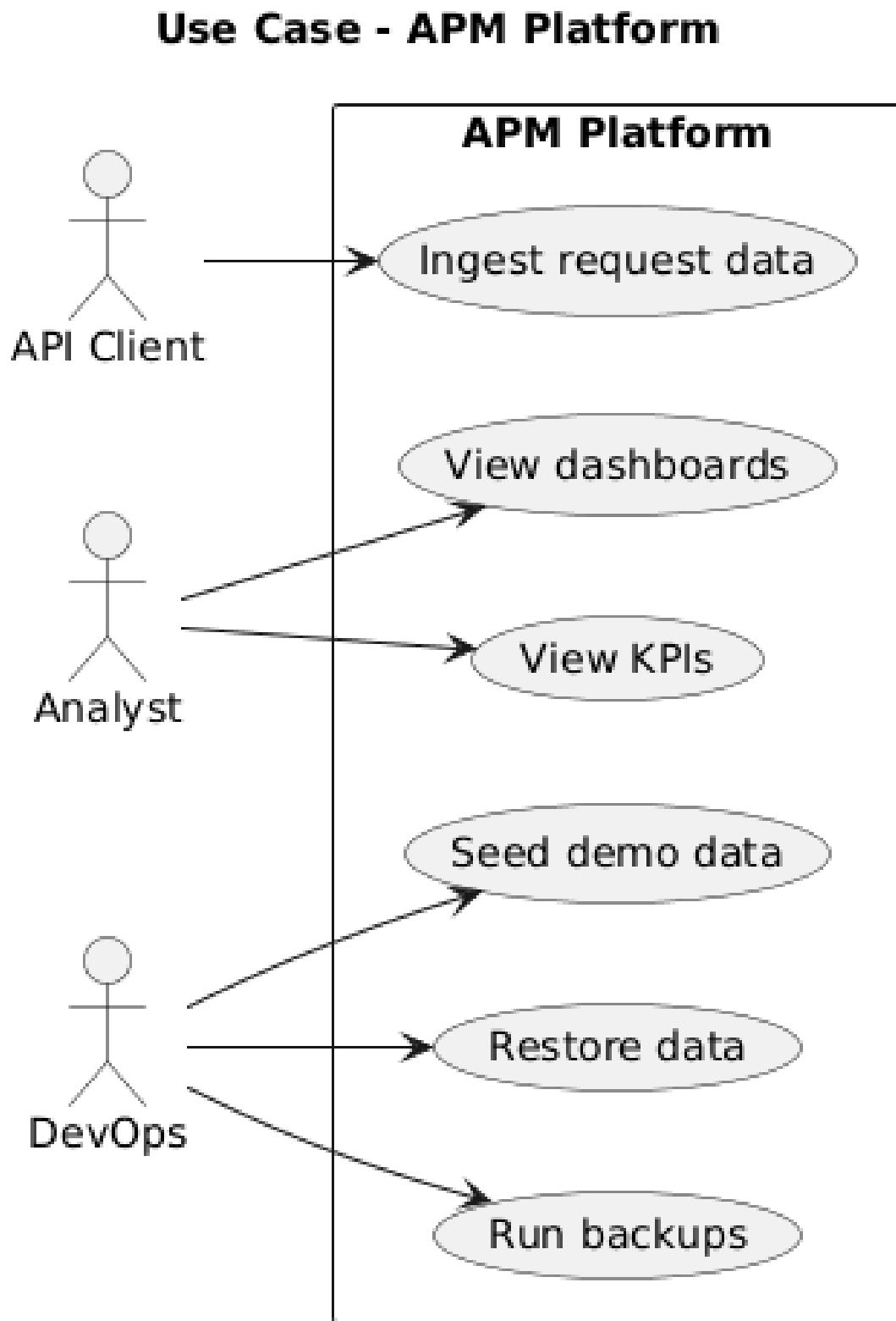


FIGURE 6 – Cas d'usage principaux de la plateforme

Class Diagram - Core Models

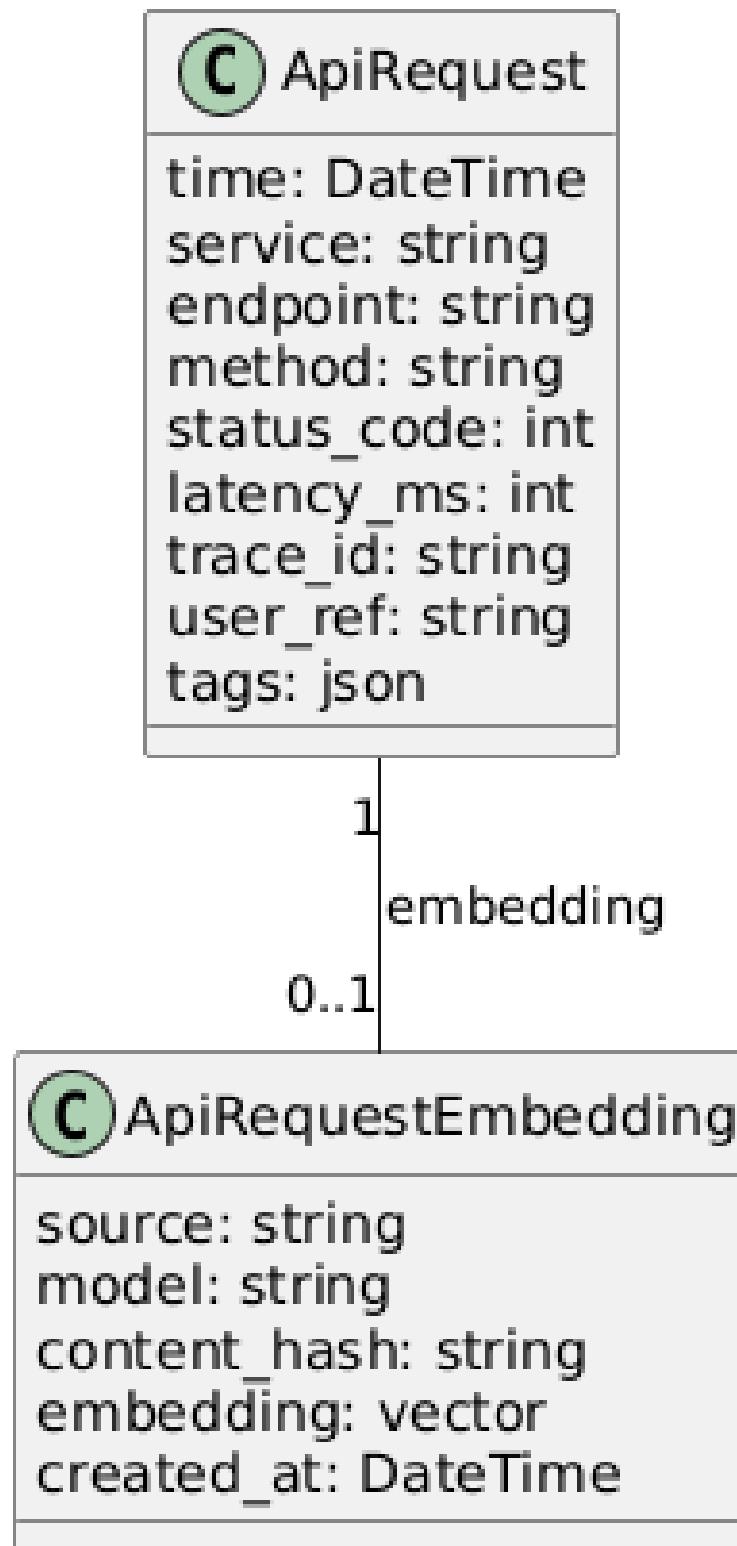


FIGURE 7 – Diagramme de classes (modèle de données)

III.6 Déploiement en cluster

Architecture générale

Le cluster repose sur plusieurs noeuds TimescaleDB orchestrés via Docker Compose, avec un backend Django commun pour l'API et l'ingestion des données.

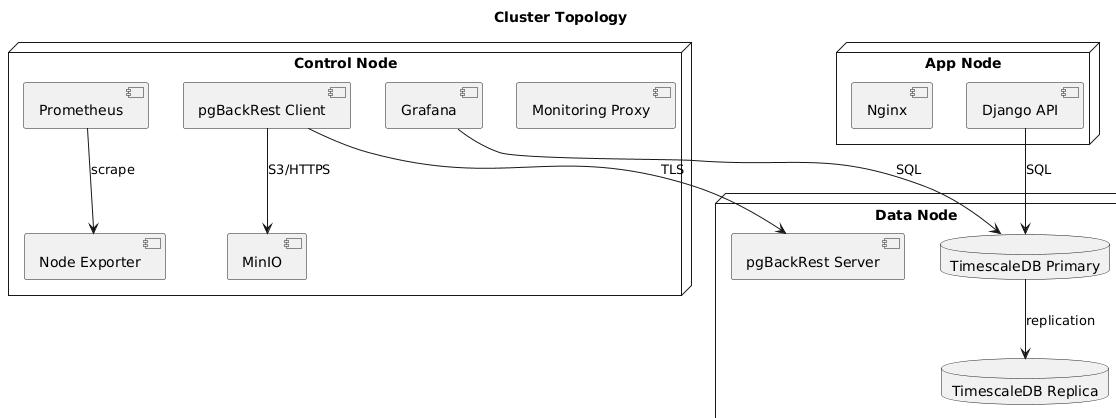


FIGURE 8 – Topologie du cluster

Cluster TimescaleDB avec Docker Compose

Chaque noeud est défini dans les fichiers suivants :

- docker/cluster/docker-compose.control.yml
- docker/cluster/docker-compose.data.yml
- docker/cluster/docker-compose.app.yml

Les conteneurs communiquent entre eux et partagent les configurations de réPLICATION.

```
make up-data
make up-control
make up-app
```

Ou en une seule commande :

```
make up-all
```

Communication entre les noeuds

Les noeuds utilisent les ports internes Docker pour échanger les données et maintenir la cohérence des hypertables. Les services Django et TimescaleDB se connectent via des noms de service Docker.

Sécurisation des accès

L'accès aux conteneurs se fait uniquement via les variables d'environnement définies dans .env. Les scripts de backup/restauration gèrent automatiquement les clés SSH pour sécuriser les opérations sans intervention manuelle.

Chapitre IV Application pratique

IV.1 Sauvegardes et restauration

Backups hot et cold

Les sauvegardes sont assurées via pgBackRest avec deux dépôts (hot/cold) sur un stockage compatible S3 (MinIO). Les opérations sont automatisées et idempotentes.

Stack backup dédiée :

```
docker compose -f docker/docker-compose.backup.yml up -d --build
make setup-backup-ssh
docker compose -f docker/docker-compose.backup.yml exec pgbackrest \
    pgbackrest --stanza=apm info
docker compose -f docker/docker-compose.backup.yml exec pgbackrest \
    pgbackrest --stanza=apm --type=full backup
```

Mode cluster :

```
make pgbackrest-full
make pgbackrest-full-repo2
```

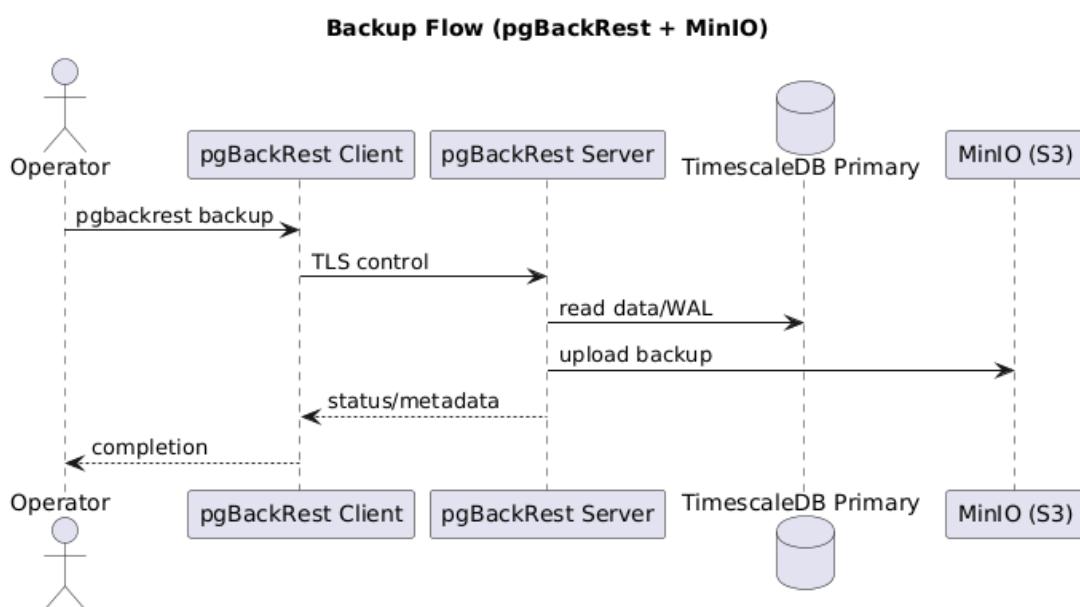


FIGURE 9 – Flux de sauvegarde pgBackRest vers MinIO

Stockage compatible S3

Les backups peuvent être exportés vers un stockage compatible S3 (MinIO). La configuration S3 se trouve dans :

- docker/backup/pgbackrest-client.conf
- docker/backup/pgbackrest-server.conf

Les buckets hot/cold sont séparés pour isoler les stratégies de rétention.

Les Figures 10 et 11 illustrent la séparation hot/cold côté MinIO.

The screenshot shows the MinIO Object Store interface. On the left, the sidebar has 'Create Bucket', 'Filter Buckets', 'Buckets' (with 'pgbackrest' and 'pgbackrest-cold' listed), 'Documentation', 'License', and 'Sign Out'. The main area is titled 'Object Browser' for the 'pgbackrest' bucket. It shows the bucket was created on Thu, Dec 25 2025 14:34:31 (GMT+1) with PRIVATE access, containing 68.7 MiB - 5000 Objects. The objects listed are 'pgbackrest / pgbackrest' (Last Modified: 2025-12-25T14:34:31Z, Size: -), 'archive' (Last Modified: 2025-12-25T14:34:31Z, Size: -), and 'backup' (Last Modified: 2025-12-25T14:34:31Z, Size: -). There are buttons for 'Rewind', 'Refresh', 'Upload', and 'Create new path'.

FIGURE 10 – Bucket hot pgbackrest

The screenshot shows the MinIO Object Store interface. On the left, the sidebar has 'Create Bucket', 'Filter Buckets', 'Buckets' (with 'pgbackrest' and 'pgbackrest-cold' listed), 'Documentation', 'License', and 'Sign Out'. The main area is titled 'Object Browser' for the 'pgbackrest-cold' bucket. It shows the bucket was created on Thu, Dec 25 2025 14:34:32 (GMT+1) with PRIVATE access, containing 5.2 MiB - 1528 Objects. The object listed is 'pgbackrest' (Last Modified: 2025-12-25T14:34:32Z, Size: -). There are buttons for 'Rewind', 'Refresh', 'Upload', and 'Create new path'.

FIGURE 11 – Bucket cold pgbackrest-cold

IV.2 Données de test

Seeding de la base avec Faker

Pour les tests et démonstrations, la base peut être peuplée avec des données factices générées via la librairie Faker.

Mode cluster :

```
make seed
```

Stack principale (local) :

```
docker compose -f docker/docker-compose.yml exec web \
    python manage.py seed_apirequests --count 1000 --days 1
```

Script utilitaire :

```
scripts/seed_faker.sh --count 1000 --days 1
```

IV.3 Tolérance aux pannes

Simulation de pannes

Des tests de tolérance sont possibles en simulant l'arrêt de nœuds Docker pour vérifier la résilience du cluster. Des scripts de drill sont fournis dans `scripts/drills` :

```
CONFIRM=YES scripts/drills/02_failover_replica.sh
CONFIRM=YES scripts/drills/03_minio_outage.sh
```

Procédures de recovery

La restauration automatique depuis les backups permet de remettre rapidement le système en état de fonctionnement.

```
docker compose -f docker/docker-compose.backup.yml exec pgbackrest \
    pgbackrest --stanza=apm restore
CONFIRM=YES scripts/drills/01_primary_restore.sh
```

IV.4 Optimisation

Index et performances

L'utilisation des hypertables, des agrégats continus et d'index spécifiques permet d'optimiser les performances des requêtes analytiques, notamment pour les KPI et les tableaux de bord. Après un gros seeding, un rafraîchissement des CAGG garantit des résultats cohérents :

```
python manage.py refresh_apirequest_hourly
python manage.py refresh_apirequest_daily
```

IV.5 Tests et validation

Les scénarios API sont automatisés via Postman et exécutés en CLI avec Newman. Les collections et environnements sont disponibles dans `postman/` :

- `APM_Observability_Step1.postman_collection.json`
- `APM_Observability_Step5.postman_collection.json`
- `APM_Observability.cluster.postman_environment.json`
- `APM_Observability.main.postman_environment.json`

Les rapports HTML/JUnit de l'exécution sont disponibles dans `reports/all_tests_20251225_134101/`. Le tableau ci-dessous synthétise les résultats.

Step	Tests	Failures	Errors	Time (s)
Step 1	14	0	0	3.394
Step 2	4	0	0	1.232
Step 3	3	0	0	0.285
Step 4	2	0	0	0.410
Step 5	5	0	0	0.609
Total	28	0	0	5.930

TABLE 1 – Synthèse des tests Newman (Postman)

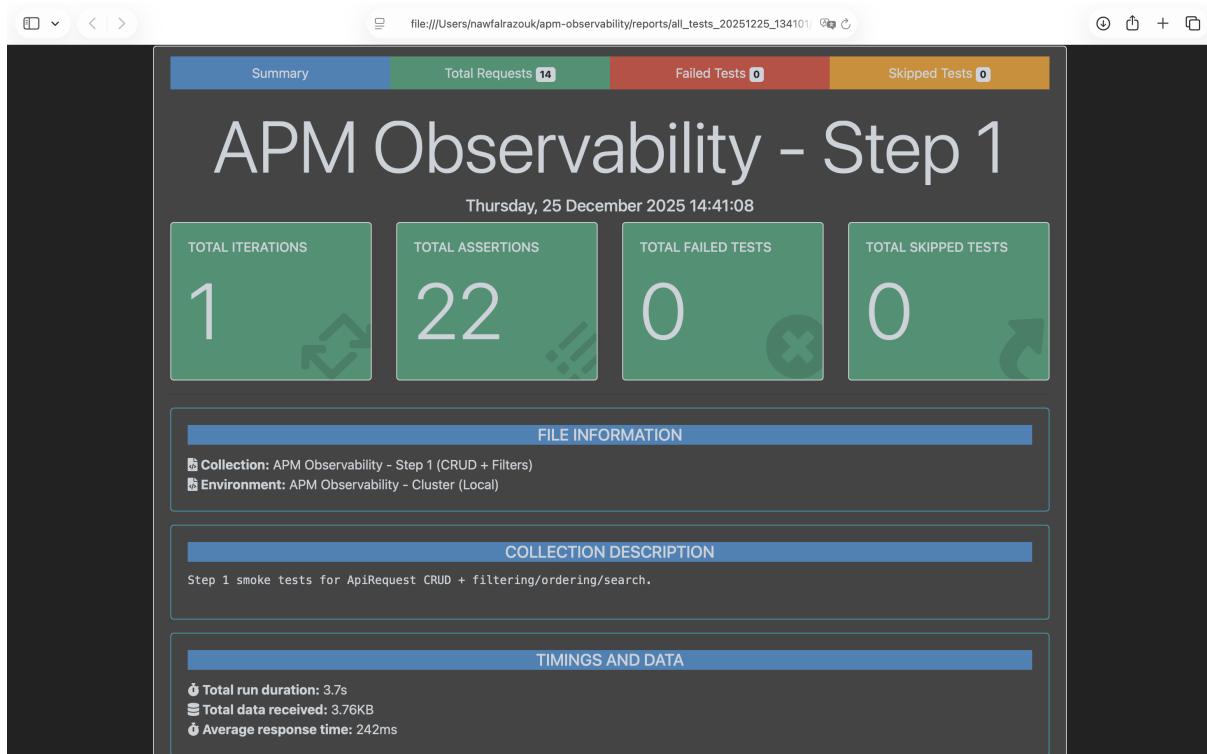


FIGURE 12 – Rapport Newman - Step 1 (CRUD + filtres)

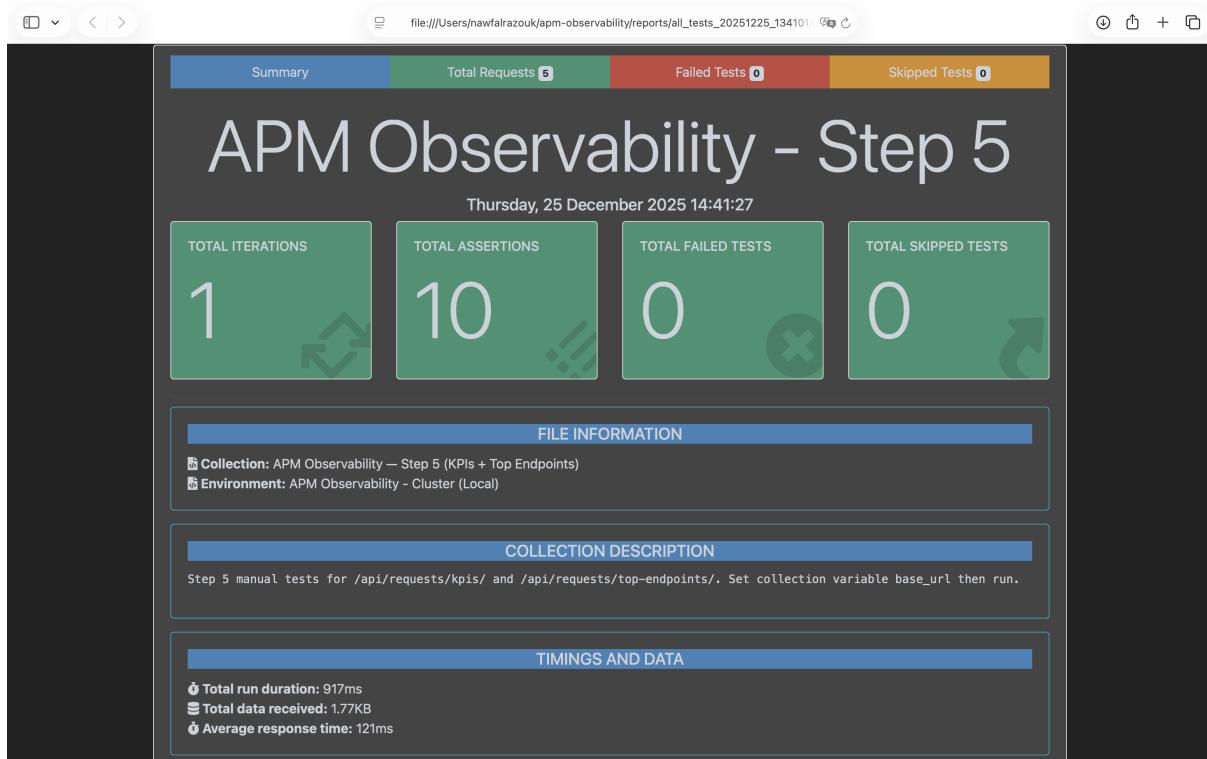


FIGURE 13 – Rapport Newman - Step 5 (KPIs / analytics)

Chapitre V Supervision avec Grafana

V.1 Présentation de Grafana

Grafana est une plateforme open-source de visualisation et de surveillance de données. Elle permet de créer des tableaux de bord interactifs et personnalisables, facilitant le suivi en temps réel des métriques provenant de diverses sources. Son interface intuitive rend l'analyse des données accessible, même pour des utilisateurs non techniques. [4]

V.2 Intégration avec TimescaleDB

Grafana s'intègre facilement avec TimescaleDB, une base de données optimisée pour les séries temporelles. Cette intégration permet de récupérer et d'afficher efficacement des données historiques et en temps réel, offrant ainsi une visibilité complète sur les tendances et les anomalies.

V.3 Configuration de la datasource TimescaleDB

Grafana est exposé en local sur `http://localhost:33000` (stack principale) ou via `make grafana` en mode cluster. Deux modes sont possibles :

Stack principale (local) :

- URL : `db:5432`
- Database : `apm`
- User : `apm` (ou un utilisateur read-only)
- SSL mode : `disable` en local

Mode cluster : la datasource est provisionnée automatiquement via le fichier `docker/monitoring/grafana/provisioning/datasources/datasources.yml`. [5]

La Figure 14 illustre les datasources configurées (TimescaleDB + Prometheus).

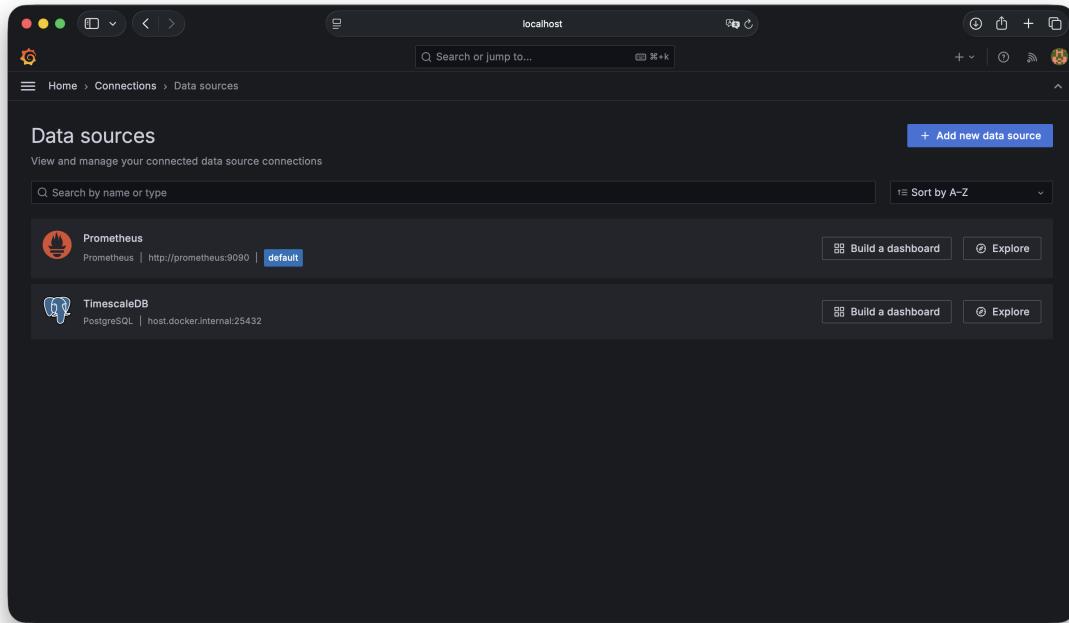


FIGURE 14 – Datasources Grafana (TimescaleDB et Prometheus)

V.4 Exemples de requêtes SQL (panels)

Les vues continues sont disponibles pour accélérer les requêtes. Exemple : `apirequest_hourly`. Exemples :

```
-- Hits par heure
SELECT bucket AS time, hits
FROM apirequest_hourly
WHERE $__timeFilter(bucket)
ORDER BY bucket;

-- Taux d'erreur (%)
SELECT
    bucket AS time,
    100.0 * errors / NULLIF(hits, 0) AS error_rate
FROM apirequest_hourly
WHERE $__timeFilter(bucket)
ORDER BY bucket;

-- Latence moyenne par service (raw table)
SELECT
    time_bucket('1 hour', time) AS time,
    service,
    AVG(latency_ms) AS avg_latency_ms
FROM observability_apirequest
WHERE $__timeFilter(time)
GROUP BY 1, 2
ORDER BY 1;
```

V.5 Dashboard de monitoring

Le tableau de bord ci-dessous montre la supervision d'un système avec Grafana et TimescaleDB.

Métriques affichées :

- Nombre de requêtes par heure (hits/hour)
- Latence moyenne par service
- Taux d'erreur par endpoint

En mode cluster, des dashboards sont déjà provisionnés dans `docker/monitoring/grafana/provisioning/dashboards`. Exemples :

- `apm-timescale.json`
- `apm-infra.json`

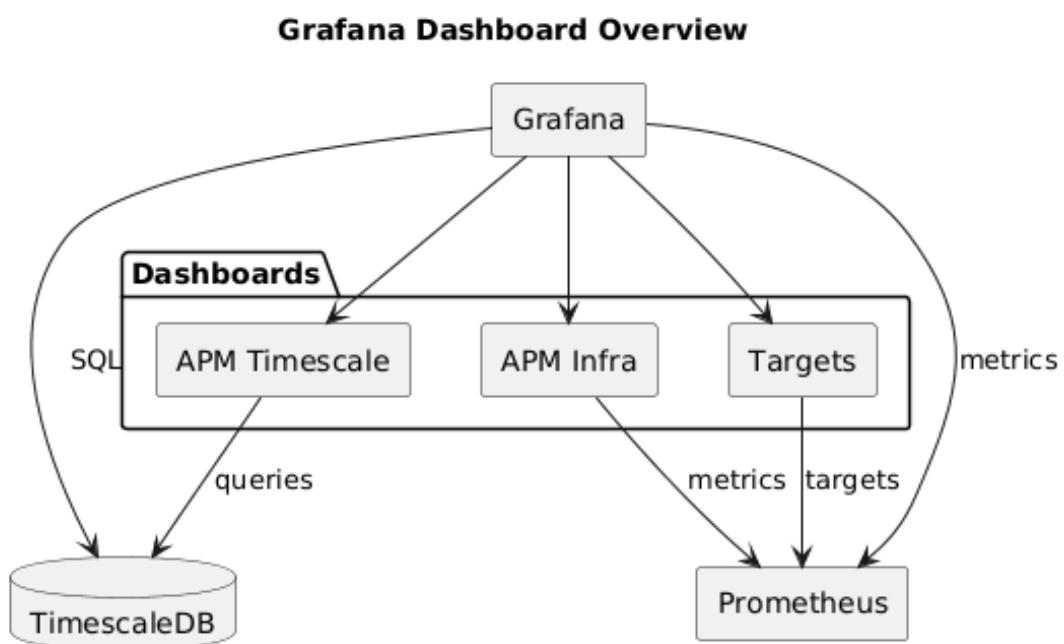


FIGURE 15 – Structure logique du dashboard Grafana



FIGURE 16 – Dashboard Grafana (capture d'écran)

V.6 Infrastructure et targets (Prometheus)

Les tableaux de bord Prometheus mettent en évidence l'état des cibles de scraping et l'infrastructure sous-jacente (CPU, mémoire, disque).



FIGURE 17 – Dashboard APM Infra Overview

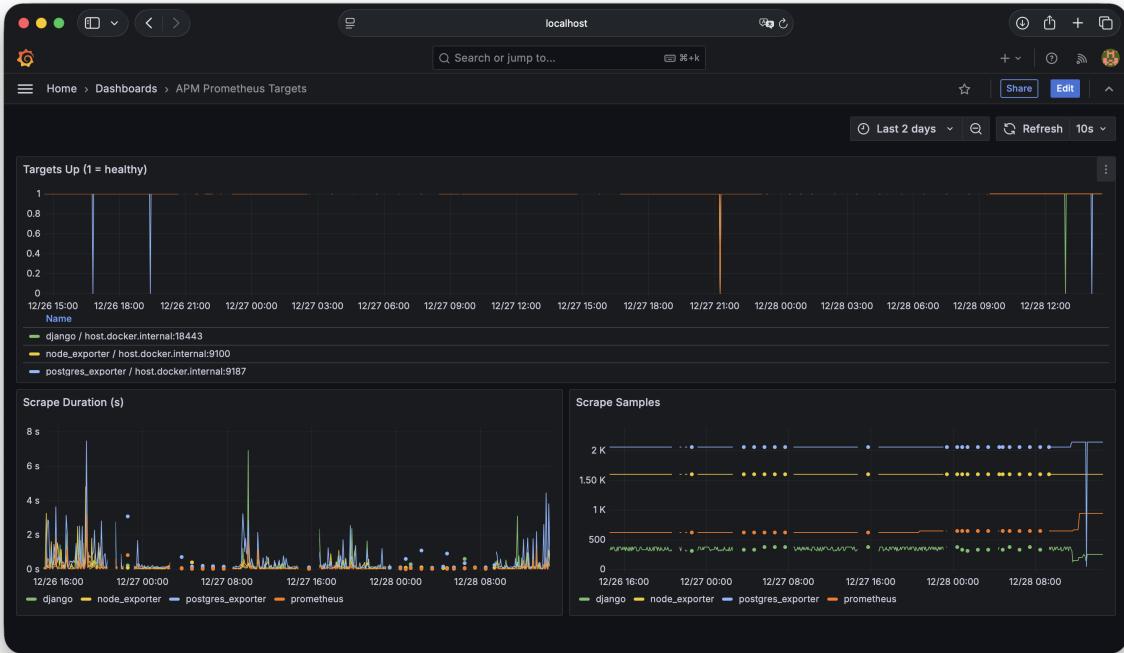


FIGURE 18 – Dashboard APM Prometheus Targets

La Figure 19 montre l'état des targets directement dans l'UI Prometheus.

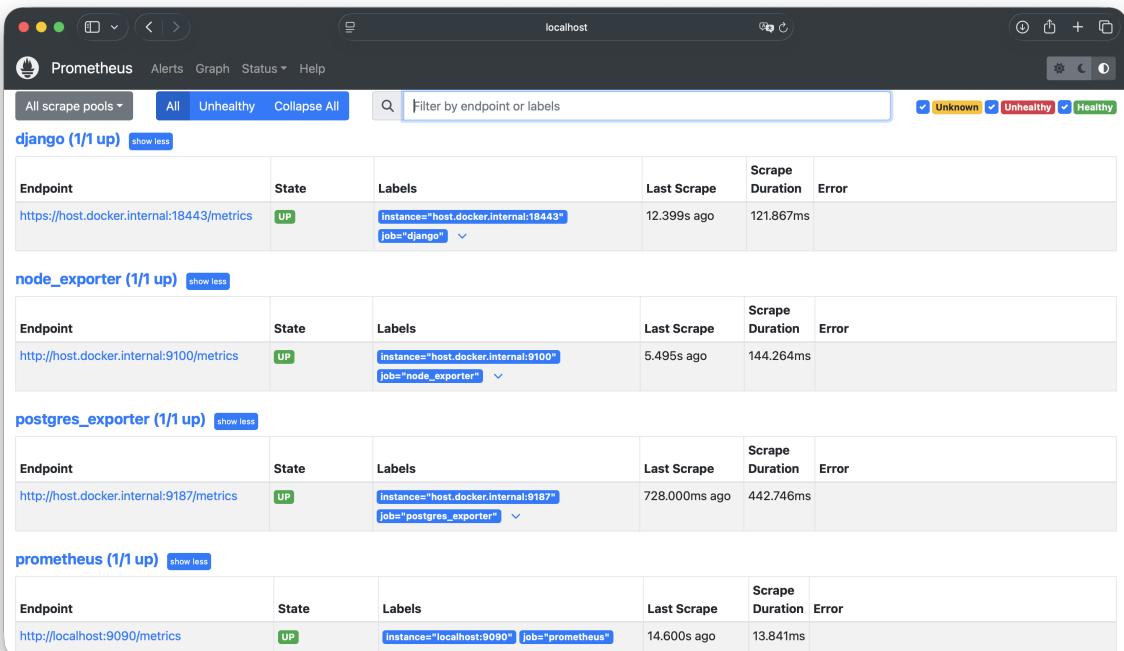


FIGURE 19 – Prometheus /targets (état des cibles)

Chapitre VI Conclusion et perspectives

Ce travail a permis de mettre en place une plateforme d'observabilité APM cohérente, couvrant l'ingestion, le stockage, l'analyse et la visualisation des données. L'usage de TimescaleDB pour les séries temporelles, associé à des vues continues pour les KPI, apporte un socle performant et maintenable. La supervision via Prometheus et Grafana fournit une vision claire de la disponibilité, de la latence et du taux d'erreur, tandis que l'intégration des sauvegardes pgBackRest avec MinIO améliore la résilience globale. Les jeux de tests Postman/Newman et les scripts de déploiement documentés rendent la solution reproductible.

Sur le plan d'architecture, nous avons validé une configuration en mode cluster adaptée aux besoins de supervision et de charge. Ce choix facilite l'évolutivité, la séparation des responsabilités et la montée en puissance future. Dans un contexte de production, le cloud reste souvent le meilleur compromis en termes de coût, d'élasticité et d'exploitation, tandis que l'on-premise répond surtout aux exigences fortes de souveraineté et de conformité.

Perspectives

Plusieurs axes peuvent enrichir la plateforme :

- mise en place d'alertes avancées et d'objectifs SLO/SLA ;
- ajout de traces distribuées (OpenTelemetry) pour compléter métriques et logs ;
- optimisation des coûts via politiques de rétention, compression et archivage ;
- industrialisation CI/CD pour automatiser tests, migrations et déploiements ;
- durcissement sécurité (RBAC, rotation des secrets, auditabilité).

Chapitre A Commandes et sorties

Initialisation (cluster)

```
make up-all
```

Sortie (extrait) :

```
[+] Running 3/3
OK Network apm-data_default    Created
OK Container apm-data-db-1     Started
OK Container apm-app-web-1     Started
...
```

Seed de donnees

```
make seed
```

Sortie (extrait) :

```
Seeded via ORM: inserted=1000
```

Backups pgBackRest

```
make pgbackrest-full
```

Sortie (extrait) :

```
P00    INFO: backup start
P00    INFO: backup stop
P00    INFO: backup complete
```

Healthcheck API

```
curl -kI https://localhost:8443/api/health/ | head -n 5
```

Sortie (extrait) :

```
HTTP/2 200
content-type: application/json
...
```

Chapitre B Structure du projet (extrait)

```
docs/
  report_latex/
    main.tex
    intro.tex
    Chap1_Base.tex
    Chap2_Mise.tex
```

```

Chap3_Pratique.tex
Chap4_Graphana.tex
conclusion.tex
annexes.tex
references.bib
images/
docker/
  cluster/
  monitoring/
observability/

```

Chapitre C Exemples de requetes SQL

```

-- Hits par heure
SELECT bucket AS time, hits
FROM apirequest_hourly
WHERE $_timeFilter(bucket)
ORDER BY bucket;

-- Taux d'erreur (%)
SELECT
  bucket AS time,
  100.0 * errors / NULLIF(hits, 0) AS error_rate
FROM apirequest_hourly
WHERE $_timeFilter(bucket)
ORDER BY bucket;

```

Chapitre D Extraits de configuration

Grafana datasource (cluster)

```

type: postgres
url: ${DATA_NODE_IP}:${CLUSTER_DATA_DB_HOST_PORT}
database: ${POSTGRES_DB}
user: ${POSTGRES_READONLY_USER}
sslmode: ${DB_SSLMODE}

```

pgBackRest (client)

```

repo1-type=s3
repo1-s3-endpoint=https://minio:9000
repo1-s3-bucket=pgbackrest
repo2-s3-bucket=pgbackrest-cold

```

Chapitre E Extraits de code et resultats

Ingestion API (Django)

```

def _parse_ingest_payload(self, data: Any) -> list[Any]:
    if isinstance(data, list):
        return data
    if isinstance(data, dict):
        if "events" not in data:
            raise ValidationError(
                {"detail": "Expected a list payload or an object with an 'events' list."}
            )
        events = data.get("events")
        if not isinstance(events, list):
            raise ValidationError({"events": "Must be a list of event objects."})
        return events
    raise ValidationError({"detail": "Expected JSON list or object payload."})

@action(detail=False, methods=["post"], url_path="ingest")
def ingest(self, request, *args, **kwargs):
    strict = self._get_bool_qp(request, "strict", default=False)
    events = self._parse_ingest_payload(request.data)
    # ...
    if strict and invalid_found:
        return Response(
            {"detail": "Strict mode enabled: payload contains invalid items.",
             "inserted": 0, "rejected": len(events), "errors": errors},
            status=status.HTTP_400_BAD_REQUEST,
        )
    instances = [ApiRequest(**row) for row in validated_rows]
    if instances:
        with transaction.atomic():
            ApiRequest.objects.bulk_create(instances, batch_size=batch_size)

```

Migration TimescaleDB (hypertable)

```

def forwards(apps, schema_editor):
    statements = [
        """
DO $$
BEGIN
    CREATE EXTENSION IF NOT EXISTS timescaledb;
END $$;
""",
        """
DO $$
BEGIN
    PERFORM create_hypertable(
        'observability_apirequest',
        'time',

```

```

        if_not_exists => TRUE,
        migrate_data => TRUE,
        create_default_indexes => FALSE,
        chunk_time_interval => INTERVAL '1 day'
    );
END $$;
""",
]
with schema_editor.connection.cursor() as cursor:
    for sql in statements:
        cursor.execute(sql)

```

Extrait JUnit (Step 1)

```

<?xml version="1.0" encoding="UTF-8"?>
<testsuites name="APM Observability - Step 1 (CRUD + Filters)" tests="14" time=
  "3.394">
  <testsuite name="00 - Create sample A (older time, 500)" tests="1" failures="0" errors="0">
    <testcase name="Status 201" time="0.252"/>
  </testsuite>
  <testsuite name="10 - GET list (default ordering -time)" tests="3" failures="0" errors="0">
    <testcase name="Status 200" time="0.053"/>
    <testcase name="Has array" time="0.053"/>
    <testcase name="Ordered by -time (non-increasing)" time="0.053"/>
  </testsuite>
</testsuites>

```

Resultat KPI (JSON)

```
{
  "hits": 7,
  "errors": 1,
  "error_rate": 0.14285714285714285,
  "avg_latency_ms": 151.14285714285714,
  "p95_latency_ms": 446.99999999999994,
  "max_latency_ms": 540,
  "source": "hourly"
}
```

Top endpoints (JSON)

```
{
  "source": "hourly",
  "results": [

```

```
{"service": "api", "endpoint": "/health", "hits": 2, "errors": 0, "error_rate": 0.0, "avg_latency_ms": 15.0, "p95_latency_ms": null, "max_latency_ms": 15}, {"service": "billing", "endpoint": "/x", "hits": 2, "errors": 1, "error_rate": 0.5, "avg_latency_ms": 330.0, "p95_latency_ms": null, "max_latency_ms": 540}, {"service": "api", "endpoint": "/orders", "hits": 1, "errors": 0, "error_rate": 0.0, "avg_latency_ms": 105.0, "p95_latency_ms": null, "max_latency_ms": 105}, {"service": "auth", "endpoint": "/login", "hits": 1, "errors": 0, "error_rate": 0.0, "avg_latency_ms": 230.0, "p95_latency_ms": null, "max_latency_ms": 230}, {"service": "web", "endpoint": "/home", "hits": 1, "errors": 0, "error_rate": 0.0, "avg_latency_ms": 33.0, "p95_latency_ms": null, "max_latency_ms": 33} ] }
```

Chapitre F Autres recommandations

- Ajouter une liste d'acronymes (APM, CAGG, WAL, MVCC).
- Documenter les prérequis exacts (versions Docker/Python/Node).
- Ajouter des captures avec sources pour les figures.
- Normaliser la terminologie (FR/EN) dans tout le document.
- Ajouter un court paragraphe de limites et risques.

Références

- [1] “Timescaledb documentation,” <https://docs.timescale.com/>.
- [2] “Timescale customers,” <https://www.timescale.com/customers/>.
- [3] “Timescaledb compression,” <https://docs.timescale.com/use-timescale/latest/compression/>.
- [4] “Grafana documentation,” <https://grafana.com/docs/grafana/latest/>.
- [5] “Postgresql data source - grafana,” <https://grafana.com/docs/grafana/latest/datasources/postgres/>.