UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Data Lake : MinIO

Sommaire

1. Introduction au Data Lake
2. Architecture du stockage objet
3. Introduction à **MinIO**
4. Gestion de la sécurité et IAM
5. Intégration avec les outils data
6. Conception d’un Data Lake avec MinIO
7. Supervision et résilience

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction au Data Lake

Introduction au Data Lake

Concepts clés

1.1. Définition et principes

Un **Data Lake** est un **réservoir centralisé** qui permet de **stocker de grandes quantités de données hétérogènes** (structurées, semi-structurées et non structurées), à l'état **brut** ou **transformé**.

> **Analogie** : le Data Lake est un lac dans lequel on verse de l'eau (les données) provenant de différentes rivières (les sources).
> Les données restent dans leur **forme native**, prêtes à être explorées, transformées ou exploitées.

Caractéristiques clés :

- Stockage massif, scalable et peu coûteux (souvent objet : S3, MinIO, ADLS).
- Données conservées dans leur format original.
- Séparation des couches logiques pour le traitement.
- Compatible avec les outils Big Data : Spark, Hive, Presto, Airflow, etc.

Introduction au Data Lake

Concepts clés

1.2. Les 3 couches logiques : Bronze / Silver / Gold

Cette approche “multi-zones” permet de **structurer le Data Lake** selon le degré de fiabilité et de transformation des données.

🥉 1. Zone Bronze — Données brutes

- Contient les **données sources** telles qu’elles arrivent : fichiers CSV, JSON, logs, images, IoT, etc.
- Aucun nettoyage ni transformation.
- Objectif : **conserver une copie intégrale** et immuable pour audit ou replay.

“ Exemple : `/bronze/transactions/raw_2025-11-07.csv` ”

Introduction au Data Lake

Concepts clés

1.2. Les 3 couches logiques : Bronze / Silver / Gold

Cette approche “multi-zones” permet de **structurer le Data Lake** selon le degré de fiabilité et de transformation des données.

🥈 2. Zone Silver — Données nettoyées

- Données **filtrées, normalisées, typées**.
- Suppression des doublons, correction des formats, harmonisation des schémas.
- Objectif : **préparer** les données pour les usages analytiques.

> Exemple : `/silver/transactions/normalized_2025-11-07.parquet`

Introduction au Data Lake

Concepts clés

1.2. Les 3 couches logiques : Bronze / Silver / Gold

Cette approche “multi-zones” permet de **structurer le Data Lake** selon le degré de fiabilité et de transformation des données.

🥇 3. Zone Gold — Données enrichies et agrégées

- Données prêtes à être consommées par les métiers ou outils BI.
- Agrégations, jointures, KPI, regroupements par client ou produit.
- Objectif : fournir une **source unique de vérité (single source of truth)**.

“ 📂 Exemple : `/gold/transactions/summary_by_country_2025-11-07.parquet` ”

Introduction au Data Lake

1.3. Schéma d’architecture simplifié

Introduction au Data Lake

1.4. Formats de stockage utilisés

Type de données	Format typique	Description
Structurées	CSV, Parquet	Tables, bases de données
Semi-structurées	JSON, Avro	Logs, API, capteurs
Non structurées	Images, PDF, vidéos	Archives, documents

1.5. Rôles et bonnes pratiques

- Séparer clairement les zones par **bucket** ou **préfixe** dans MinIO.
- Activer **versioning et lifecycle policies** pour maîtriser les coûts.
- Documenter les transformations (métadonnées, data catalog).
- Mettre en place un **schéma de nommage uniforme** :

```
s3://datalake/{zone}/{source}/{année}/{mois}/{jour}/fichier.parquet
```

Introduction au Data Lake

Concepts clés

Exemple concret avec MinIO

Couche	Bucket / Dossier	Exemple de contenu
Bronze	`s3://lake-bronze/transactions/`	`raw_2025-11-07.json`
Silver	`s3://lake-silver/transactions/`	`cleaned_2025-11-07.parquet`
Gold	`s3://lake-gold/aggregates/`	`monthly_report_2025-11.parquet`

Introduction au Data Lake

Schéma typique

Schéma typique d'un Data Lake : ingestion → stockage → traitement → exposition

Introduction au Data Lake

Étape	Description	Outils typiques
1 **Ingestion**	Les données sont collectées depuis différentes sources (bases, APIs, fichiers, flux temps réel).	Kafka, Apache NiFi, Airbyte, Logstash, Flume
2 **Stockage**	Les données brutes sont déposées dans le **Data Lake** (zone Bronze) via un stockage **objet** compatible S3 (ex : MinIO).	MinIO, AWS S3, Azure Data Lake Storage, HDFS
3 **Traitement**	Les données sont nettoyées, enrichies, transformées et agrégées pour devenir exploitables.	Apache Spark, PySpark, Airflow, dbt
4 **Exposition / Consommation**	Les données finales sont accessibles pour l'analyse, la BI, la data science ou des APIs.	Metabase, Power BI, Superset, Grafana, Streamlit

Introduction au Data Lake

Schéma typique

1. Ingestion

But :

Faire **entrer les données dans le Data Lake** à partir de multiples sources hétérogènes.

Types de sources :

- Bases relationnelles (MySQL, PostgreSQL, Oracle)
- APIs REST / SOAP
- Fichiers CSV / JSON déposés dans un FTP ou un bucket
- Flux temps réel (Kafka topics, MQTT, IoT)

Méthodes :

- **Batch ingestion** : exécution planifiée (Airflow, cron, ETL)
- **Streaming ingestion** : flux continu (Kafka, Spark Streaming)
- **Hybrid ingestion** : mélange des deux pour répondre aux besoins temps réel et historiques

Introduction au Data Lake

Schéma typique

1. Ingestion

“ Exemple :

- Kafka reçoit les transactions bancaires en continu.
- NiFi transforme le flux JSON en format Avro.
- Le fichier Avro est stocké dans MinIO (zone Bronze). ”

Introduction au Data Lake

Schéma typique

2. Stockage

But :
Centraliser les données **dans un environnement scalable, durable et économique**.

Principes clés :

- Les données sont stockées **à l'état brut** dans la zone *Bronze*.
- Organisation en **buckets** et **préfixes** par type de données.
- Le **stockage objet (MinIO, S3)** est privilégié :
  - Extensible à l'infini
  - Moins coûteux qu'un stockage bloc
  - Accès via API REST (S3 compatible)

“ Exemple :

- `s3://bronze/transactions/raw_2025-11-07.json` ”

Introduction au Data Lake

Schéma typique

3. Traitement

But :

Transformer les données pour les rendre exploitables.

Opérations typiques :

- Nettoyage : suppression des doublons, correction de schéma
- Normalisation : typage, mapping, validation
- Enrichissement : ajout de données complémentaires
- Agrégation : regroupement par période, client, produit, etc.

Outils typiques :

- **Apache Spark / PySpark**
- **dbt (data build tool)** pour le SQL transformationnel
- **Airflow** pour l'orchestration

Introduction au Data Lake

Schéma typique

4. Exposition

But :

Rendre les données **accessibles aux utilisateurs métier, analystes et data scientists**.

Méthodes :

- **Visualisation BI** (Power BI, Metabase, Superset)
- **API REST / GraphQL** pour exposer des jeux de données
- **Connexion SQL / JDBC** vers les tables Silver ou Gold
- **Exploration data science** (Python, Pandas, Jupyter)

> Exemple :
> Les tableaux de bord Metabase se connectent à la zone Gold (MinIO + Presto/Trino) pour afficher les indicateurs financiers.

Introduction au Data Lake

Écosystèmes

Le Data Lake moderne repose sur un **écosystème de stockage et de traitement distribué**.

Catégorie	Solution	Fournisseur	Type	Description
Open Source	**Hadoop HDFS**	Apache	Stockage distribué en bloc	Base historique des Data Lakes (Hadoop 1.x / 2.x)
Cloud Public	**Amazon S3**	AWS	Stockage objet	Référence du marché, standard de facto (API S3)
Cloud Public	**Azure Data Lake Storage (ADLS)**	Microsoft	Stockage hiérarchique (objets + répertoires)	Intégré à Azure Synapse et Databricks

Introduction au Data Lake

Écosystèmes

:

Catégorie	Solution	Fournisseur	Type	Description
Cloud Public	**Google Cloud Storage (GCS)**	Google	Stockage objet	Compatible S3 via API REST et SDK
Hybride / On-prem	**MinIO**	Open Source	Stockage objet	Compatible S3, multi-cloud et auto-hébergé

Introduction au Data Lake

Écosystèmes

1 Hadoop (HDFS)

Description :

- Premier système de stockage distribué à grande échelle.
- Fichiers découpés en blocs (par défaut 128 Mo) et répliqués sur plusieurs nœuds.
- Utilisé dans les premiers Data Lakes avec Hive, Pig, MapReduce.

Avantages :

- Haute tolérance aux pannes.
- Intégration native avec l'écosystème Hadoop (YARN, Spark).

Limites :

- Complexité de gestion.
- Coûts élevés de maintenance.
- Moins adapté aux environnements cloud modernes.

Introduction au Data Lake

Écosystèmes

2 Amazon S3 (Simple Storage Service)

Description :

- Service managé de stockage objet sur AWS.
- Interface REST + SDKs dans la plupart des langages.
- Standard de facto du marché : **tous les outils Big Data** supportent le protocole S3.

Avantages :

- Haute durabilité (11 9s).
- Sécurité et versioning intégrés.
- Énorme écosystème compatible.

Limites :

- Lié à AWS (propriétaire).
- Coûts à surveiller (stockage + transfert + requêtes).

Introduction au Data Lake

Écosystèmes

3 Azure Data Lake Storage (ADLS Gen2)

Description :

- Évolution du service Azure Blob Storage.
- Combine stockage objet et structure hiérarchique.
- Support natif du protocole HDFS.

Avantages :

- Sécurité fine (ACL POSIX).
- Intégré à Azure Synapse, Databricks, Power BI.
- Haute performance en lecture/écriture séquentielle.

Limites :

- Dépendance à Azure.
- API S3 non native (interopérabilité plus limitée).

Introduction au Data Lake

Écosystèmes

4 Google Cloud Storage (GCS)

Description :

- Stockage objet distribué sur Google Cloud.
- API REST + compatibilité S3 optionnelle.
- Sécurité basée sur IAM Google.

Avantages :

- Performance élevée (latence faible).
- Intégration native avec BigQuery et Dataflow.
- Gestion simplifiée via console ou CLI (`gsutil`).

Limites :

- Lié à GCP.
- Coûts réseau interrégionaux.

Introduction au Data Lake

5️⃣ MinIO

Description :

- Solution **open-source**, **self-hosted** et **S3-compatible**.
- Fonctionne sur Docker, Kubernetes ou bare-metal.
- Fournit les mêmes API que S3 (compatibilité complète).

Avantages :

- 🚀 Ultra performant (Go natif, très rapide en I/O).
- 🔒 Sécurisé (chiffrement, IAM, versioning).
- ⚙️ Déploiement flexible (standalone, cluster, multi-tenant).
- 🧩 Compatible cloud : AWS, Azure, GCP, ou on-prem.
- 💰 Économique : aucune dépendance à un fournisseur.

Limites :

- Pas de service managé (nécessite supervision).
- Moins d'outils intégrés que les hyperscalers (mais interopérable avec eux).

Critère	Hadoop (HDFS)	AWS S3	Azure ADLS	GCS	MinIO
Type	Bloc distribué	Objet	Objet/hiérarchique	Objet	Objet
API S3 compatible	❌	✅	⚙️ (via connecteurs)	⚙️	✅
Cloud natif	❌	✅	✅	✅	✅/On-prem
Open Source	✅	❌	❌	❌	✅
Auto-hébergement	✅	❌	❌	❌	✅
Intégration Spark	✅	✅	✅	✅	✅
Performance	⚙️	🔥	🔥	🔥	🚀
Coût	Matériel	Pay-per-	Pay-per-use	Pay-per-	Libre / infra

Introduction au Data Lake

Écosystèmes

Schéma de positionnement

Introduction au Data Lake

Écosystèmes

Cas d'usage concret : MinIO dans un Data Lake hybride

Contexte :

Une entreprise veut stocker ses données sensibles **on-premise** tout en exposant certaines zones au cloud.

Solution :

- **MinIO** installé sur 3 nœuds locaux pour les zones Bronze/Silver.
- Synchronisation **avec AWS S3** pour les données agrégées (Gold) destinées à Power BI.
- Lecture directe par **Spark** via `s3a://minio-endpoint:9000/lake-gold`.

> Avantage : maîtrise totale des coûts + compatibilité totale avec les outils S3 existants.

Introduction au Data Lake

Cas d'usage

1 Secteur bancaire – Détection des fraudes et conformité

Objectif

Centraliser l'ensemble des transactions financières pour :

- détecter les opérations suspectes,
- répondre aux exigences réglementaires (AML, RGPD),
- fournir une traçabilité complète.

⚙ Flux typique

```
[Système de transaction bancaire]
          ↓ (Kafka / Debezium)
   [Data Lake MinIO - Bronze]
          ↓ (Spark Batch/Streaming)
   [MinIO - Silver : données normalisées]
          ↓ (ETL Airflow)
   [MinIO - Gold : agrégats, alertes]
          ↓
     [Tableau de bord Power BI / Metabase]
```

Introduction au Data Lake

Cas d’usage

1️⃣ Secteur bancaire – Détection des fraudes et conformité

Exemple concret

- **Bronze** : dépôt brut des flux Kafka (`transactions_raw_*.json`)
- **Silver** : Spark applique des règles KYC (Know Your Customer) et nettoie les données.
- **Gold** : données agrégées par client/pays pour analyse des anomalies.

> ✅ **Bénéfices** : conformité, traçabilité, et détection de fraudes temps réel.

Introduction au Data Lake

Cas d’usage

2️⃣ E-commerce – Personnalisation et analyse comportementale

Objectif

Analyser les parcours clients et comportements d’achat afin de :

- personnaliser les recommandations,
- optimiser les stocks,
- améliorer le taux de conversion.

Flux typique

```
[Logs web / API / CRM / Commandes]
          ↓ (Ingestion NiFi / Airbyte)
   [MinIO - Bronze]
          ↓ (Transformation PySpark)
   [MinIO - Silver : sessions nettoyées]
          ↓ (Agrégations SQL / dbt)
   [MinIO - Gold : KPI et tableaux BI]
          ↓
       [Power BI / Superset]
```

Introduction au Data Lake

Cas d’usage

Exemple concret

- **Bronze** : logs de navigation et achats bruts.
- **Silver** : normalisation des identifiants clients et timestamps.
- **Gold** : analyse des ventes par segment, produits recommandés.

> **Bénéfices** : vision 360° du client, meilleure expérience utilisateur, décisions marketing basées sur la donnée.

Introduction au Data Lake

3 IoT – Supervision d'équipements connectés

Objectif

Collecter en temps réel les mesures provenant de capteurs ou d'objets connectés pour :

- anticiper les pannes,
- détecter les anomalies,
- optimiser la maintenance.

Flux typique

```
[Capteurs IoT → MQTT Broker]
          ↓
     [Kafka Topic]
          ↓
  [MinIO - Bronze : flux JSON]
          ↓
  [Spark Streaming : nettoyage]
          ↓
  [MinIO - Silver : séries temporelles Parquet]
          ↓
  [Prometheus / Grafana / Power BI]
```

Introduction au Data Lake

Cas d’usage

3 IoT – Supervision d’équipements connectés

Exemple concret

- Données collectées toutes les 5 secondes depuis des machines industrielles.
- Spark Streaming nettoie les données (valeurs aberrantes, timestamps manquants).
- Gold = moyennes horaires et alertes automatiques.

“ **Bénéfices** : maintenance prédictive, continuité de service, réduction des coûts d’arrêt. ”

Introduction au Data Lake

Cas d'usage

4 Log Analytics – Observabilité et sécurité

🔍 Objectif

Centraliser et analyser les logs applicatifs, systèmes et de sécurité pour :

- identifier les erreurs récurrentes,
- détecter des attaques,
- améliorer la performance des applications.

⚙ Flux typique

```
[Applications / Serveurs / Firewalls]
          ↓ (Filebeat / Fluentd)
       [Kafka ou direct S3 API]
          ↓
    [MinIO - Bronze : logs bruts]
          ↓
    [Spark / Elastic ingestion]
          ↓
    [MinIO - Silver : logs parsés]
          ↓
    [Metabase / Kibana / Grafana]
```

Introduction au Data Lake

Cas d’usage

4 Log Analytics – Observabilité et sécurité

Exemple concret

- **Bronze** : stockage des logs JSON journaliers dans MinIO.
- **Silver** : parsing automatique des timestamps et niveaux de log.
- **Gold** : tableaux de bord Grafana pour alertes d’erreur et charge CPU.

“ ✅ **Bénéfices** : observabilité globale, corrélation d’événements, sécurité proactive. ”

Introduction au Data Lake

Comparatif synthétique

Domaine	Sources de données	Objectif principal	Couches typiques	Outils clés
Banque	Transactions, clients, agents	Détection de fraudes, conformité	Bronze → Silver → Gold	Kafka, Spark, MinIO, Metabase
E-commerce	Logs web, commandes, CRM	Personnalisation, KPI ventes	Bronze → Silver → Gold	NiFi, Spark, dbt, Power BI
IoT	Capteurs, flux MQTT	Supervision, prédiction	Bronze → Silver → Gold	Kafka, Spark Streaming, Grafana
Log Analytics	Logs système / app	Monitoring, sécurité	Bronze → Silver → Gold	Fluentd, Spark, Kibana

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Architecture du stockage objet

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

1 Le stockage en bloc (Block Storage)

Principe

- Les données sont découpées en **blocs de taille fixe** (souvent 4 Ko à 8 Ko).
- Chaque bloc reçoit un **identifiant unique** et est **géré par le système d'exploitation**.
- Le disque (physique ou virtuel) est vu comme un **volume brut**.

Exemples d'usage

- Disques système, bases de données, VM, SAN (Storage Area Network).
- Technologies : iSCSI, Fibre Channel, Amazon EBS, Azure Managed Disks.

Avantages

- Très rapide en lecture/écriture aléatoire.
- Excellente performance pour les BDD transactionnelles.

Limites

- Ne gère **aucune métadonnée** (juste des blocs bruts).
- **Pas adapté** au stockage de millions de petits fichiers ou objets variés.

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

2 Le stockage en fichier (File Storage)

Principe

- Les fichiers sont stockés dans une **hiérarchie de répertoires**.
- Le système de fichiers (NTFS, EXT4, NFS) gère les noms, droits et métadonnées de base (taille, date, propriétaire).
- L'accès se fait via des **protocoles de fichiers** : SMB, NFS, CIFS.

Exemples d'usage

- Serveurs de fichiers d'entreprise, NAS, partage réseau.
- Technologies : Windows File Server, NFS, Azure Files, Amazon EFS.

Avantages

- Simple à utiliser et à monter sur un OS.
- Bon compromis pour les environnements partagés (bureautique, projets).

Limites

- Difficulté à **scaler horizontalement**.
- Moins performant et moins économique à très grande échelle.

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

3 Le stockage objet (Object Storage)

Principe

- Les données sont stockées sous forme **d'objets indépendants**, chacun contenant :
  - le **contenu binaire** (data),
  - un **identifiant unique (UUID)**,
  - un ensemble de **métadonnées personnalisées** (auteur, type, tags, version).
- Les objets sont regroupés dans des **buckets** (conteneurs logiques).
- L'accès se fait via une **API REST (S3)** plutôt que par montage de volume.

Exemples d'usage

- Data Lakes, sauvegardes, archivage, stockage de médias, logs massifs.
- Technologies : Amazon S3, Azure Data Lake, Google Cloud Storage, **MinIO**.

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

3 Le stockage objet (Object Storage)

Avantages

- Scalabilité quasi infinie.
- Haute durabilité et résilience (réplication, erasure coding).
- Métadonnées riches et extensibles.
- Compatible avec les environnements cloud et multi-tenant.

Limites

- Non adapté aux besoins de **latence ultra-faible** (par ex. bases transactionnelles).
- Accès via API (pas directement monté comme un disque local sans passerelle).

Architecture du stockage objet

Critère	Bloc	Fichier	Objet
Unité de stockage	Bloc (4–8 Ko)	Fichier	Objet (données + métadonnées)
Gestion par	Système d'exploitation	Système de fichiers	Application / API
Structure	Volume brut	Arborescence	Espace plat (buckets)
Protocole	iSCSI, Fibre Channel	NFS, SMB	REST (HTTP, S3 API)
Scalabilité	Limitée	Moyenne	Très élevée
Métadonnées	Minimales	Standard	Riches et personnalisables
Cas d'usage	VM, BDD	Partage réseau	Data Lake, Backup, Cloud
Exemple	EBS, SAN	EFS, NFS	S3, MinIO, ADLS

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

Schéma illustratif

```
            +--------------------------------------+
            |         Comparaison des modèles      |
+------------+-------------+-------------+------------+
|   Type     | Structure   | Accès       | Exemple    |
+------------+-------------+-------------+------------+
| Bloc       | Volume brut | OS/Kernel   | EBS, SAN   |
| Fichier    | Hiérarchie  | NFS/SMB     | NAS, EFS   |
| Objet      | Bucket plat | REST (S3)   | S3, MinIO  |
+------------+-------------+-------------+------------+

Représentation visuelle :

Bloc :  [ ][ ][ ][ ]  (données découpées)
Fichier : /docs/2025/report.pdf  (arborescence)
Objet : bucket:report -> { id, data, metadata } (API REST)
```

Architecture du stockage objet

Principe du stockage objet vs bloc / fichier

💡 Exemple concret avec MinIO

```
# Création d'un bucket (équivalent d'un répertoire logique)
mc mb local/datalake-bronze

# Envoi d'un objet (fichier CSV)
mc cp clients_raw.csv local/datalake-bronze/

# Récupération des métadonnées de l'objet
mc stat local/datalake-bronze/clients_raw.csv
```

Résultat :

```
Name      : clients_raw.csv
Size      : 4.3 MB
ETag      : 9b5f1f…
Metadata  : Content-Type=text/csv, Department=Sales, Zone=Bronze
```

> 🔎 **Remarque :**
> Contrairement à un partage NFS, le stockage objet **ne "voit" pas d'arborescence réelle** : tout est géré via des **clés d'objet** et des **métadonnées**, offrant une **souplesse totale** pour les architectures Big Data.

Architecture du stockage objet

Concepts fondamentaux du stockage objet

1 Le Bucket

Définition

Un **bucket** est l'**espace logique de stockage** dans lequel sont placés les objets.
C'est l'équivalent d'un **répertoire racine** (folder) dans un système de fichiers classique, mais à **plat** (sans hiérarchie réelle).
Chaque bucket :

- appartient à un compte utilisateur,
- possède un **nom unique** dans l'instance MinIO ou le compte S3,
- peut contenir **des millions d'objets**,
- dispose de **politiques d'accès** (ACL, IAM).

Architecture du stockage objet

Concepts fondamentaux du stockage objet

1️⃣ Le Bucket

Exemples concrets

```
# Création d'un bucket
mc mb local/datalake-bronze

# Liste des buckets existants
mc ls local/
```

Résultat :

```
[2025-11-07 12:15:00 UTC]  0B  datalake-bronze/
[2025-11-07 12:15:10 UTC]  0B  datalake-silver/
[2025-11-07 12:15:20 UTC]  0B  datalake-gold/
```

“ ⚙️ Les buckets sont souvent utilisés pour représenter les **zones logiques d'un Data Lake** :

- `datalake-bronze` → données brutes
- `datalake-silver` → données nettoyées
- `datalake-gold` → données agrégées ”

Architecture du stockage objet

Concepts fondamentaux du stockage objet

2 L'Object

Définition

Un **object** est l'**unité de stockage fondamentale**.
Chaque objet regroupe :

- le **contenu binaire** (fichier, image, JSON, Parquet, etc.),
- des **métadonnées associées**,
- un **identifiant unique (key)** qui remplace le chemin de fichier.

> Dans le stockage objet, il n'y a **pas de hiérarchie réelle** :
> L'"arborescence" est simulée par des **préfixes** dans les clés.

Exemples

```
# Envoi d'un fichier CSV vers un bucket
mc cp clients_raw.csv local/datalake-bronze/raw/clients_raw_2025-11-07.csv

# Liste des objets
mc ls local/datalake-bronze/raw/
```

Architecture du stockage objet

Concepts fondamentaux du stockage objet

2 L'Object

Résultat :

```
[2025-11-07 12:20:10 UTC]  4.3MiB  clients_raw_2025-11-07.csv
```

> L'objet est identifié par sa **clé complète** :
> `datalake-bronze/raw/clients_raw_2025-11-07.csv`

Architecture du stockage objet

Concepts fondamentaux du stockage objet

3 Les Metadata (Métadonnées)

Définition

Les **métadonnées** décrivent les **caractéristiques et le contexte** de chaque objet.
Elles peuvent être :

- **automatiques** : générées par le système (date, taille, hash, ETag, Content-Type),
- **personnalisées** : ajoutées par l'utilisateur (projet, auteur, classification, zone...).

Exemples

```
# Ajout d'un objet avec des métadonnées personnalisées
mc cp --attr "project=bank-analytics;zone=bronze;source=kafka" \
   transactions.json local/datalake-bronze/raw/

# Lecture des métadonnées
mc stat local/datalake-bronze/raw/transactions.json
```

Architecture du stockage objet

Concepts fondamentaux du stockage objet

3 Les Metadata (Métadonnées)

Résultat :

```
Name        : transactions.json
Size        : 8.1 MB
ETag        : 9b5f1f...
Metadata    :
  Content-Type   : application/json
  project        : bank-analytics
  zone           : bronze
  source         : kafka
```

> 💡 Ces métadonnées peuvent ensuite être **exploitées par Spark** ou un **catalogue de données** pour filtrer, tracer ou classifier les objets.

Architecture du stockage objet

Concepts fondamentaux du stockage objet

4 Le Versioning

Principe

Le **versioning** permet de **conserver plusieurs versions d'un même objet**, chaque mise à jour créant une **nouvelle révision** au lieu d'écraser la précédente.
C'est un **mécanisme essentiel pour les Data Lakes**, car il permet :

- d'assurer la **traçabilité**,
- d'éviter la **perte accidentelle**,
- de restaurer une version antérieure en cas d'erreur.

Exemple concret

```
# Activer le versioning sur un bucket
mc version enable local/datalake-bronze=
# Envoyer un fichier (v1)
mc cp data_v1.csv local/datalake-bronze/datasets/clients.csv
# Envoyer une nouvelle version du même objet
mc cp data_v2.csv local/datalake-bronze/datasets/clients.csv
# Liste des versions
mc ls --versions local/datalake-bronze/datasets/
```

Architecture du stockage objet

Concepts fondamentaux du stockage objet

4 Le Versioning

Résultat :

```
[VersionID: 4a91e1f]  data_v2.csv  (latest)
[VersionID: 9b2c8aa]  data_v1.csv
```

“ Chaque version possède un **VersionID unique**, visible dans la console MinIO. ”

Architecture du stockage objet

Concepts fondamentaux du stockage objet

Schéma visuel simplifié

Architecture du stockage objet

Concepts fondamentaux du stockage objet

Résumé comparatif

Concept	Rôle	Analogie	Exemple MinIO
Bucket	Conteneur logique	Dossier racine	`datalake-bronze`
Object	Donnée stockée + clé unique	Fichier	`raw/clients_2025.csv`
Metadata	Informations descriptives	Étiquettes / tags	`zone=bronze`
Versioning	Historique des modifications	Système de sauvegarde	`VersionID=4a91e1f`

Architecture du stockage objet

Concepts fondamentaux du stockage objet

Bonnes pratiques dans un Data Lake MinIO

- Organiser les buckets par **zone fonctionnelle** (Bronze / Silver / Gold).
- Ajouter des **métadonnées obligatoires** (auteur, source, zone, date).
- Activer le **versioning** sur les buckets critiques.
- Nettoyer régulièrement les anciennes versions via **lifecycle policies**.
- Utiliser **des conventions de clés uniformes** :

```
{zone}/{source}/{année}/{mois}/{jour}/{dataset}.parquet
```

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

1 Le protocole REST (REpresentational State Transfer)

Principe

Le protocole **REST** est une **architecture d'échange** basée sur **HTTP**.
Les clients envoient des **requêtes HTTP standard (GET, PUT, POST, DELETE)** vers une **URL unique**, et le serveur renvoie des **réponses JSON/XML**.
C'est le **socle historique** sur lequel repose la majorité des API web modernes — y compris **S3** et **MinIO**.

Exemple générique d'appel REST

```
# Requête pour récupérer un fichier (objet)
GET http://localhost:9000/datalake-bronze/raw/clients.csv
Authorization: Bearer <token>
```

Réponse :

```
200 OK
Content-Type: text/csv
```

Architecture du stockage objet

Protocoles d’accès : S3 API, REST, gRPC

1 Le protocole REST (REpresentational State Transfer)

Avantages

- Simple à utiliser (HTTP standard).
- Compatible avec n’importe quel langage ou outil.
- Très répandu (S3, Azure Blob, GCS reposent sur REST).

Limites

- Communication **textuelle** (JSON/XML → plus lourde).

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

2 L'API S3 (Amazon Simple Storage Service API)

Principe

L'**API S3** est une **spécification RESTful** standardisée par Amazon Web Services.
Elle est devenue **le protocole de facto du stockage objet** : la plupart des solutions (MinIO, Ceph, OpenIO, Cloudian, etc.) l'ont **implémentée à l'identique**.

> 💡 **MinIO** est **100 % compatible avec l'API S3** :
> Toute application conçue pour AWS S3 fonctionne sans modification sur MinIO.

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

2 L'API S3 (Amazon Simple Storage Service API)

Structure d'une requête S3

Les principales opérations sont :

Opération	Méthode HTTP	Description
ListBuckets	GET `/`	Liste tous les buckets
CreateBucket	PUT `/{bucket}`	Crée un bucket
PutObject	PUT `/{bucket}/{key}`	Ajoute un objet
GetObject	GET `/{bucket}/{key}`	Télécharge un objet
DeleteObject	DELETE `/{bucket}/{key}`	Supprime un objet

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

2️⃣ L'API S3 (Amazon Simple Storage Service API)

Exemple d'utilisation avec AWS CLI ou MinIO Client

```
# Création d'un bucket
mc mb local/datalake-bronze

# Téléversement d'un fichier
mc cp data.csv local/datalake-bronze/

# Téléchargement via l'API S3
curl -X GET \
  -u "minioadmin:minioadmin" \
  http://localhost:9000/datalake-bronze/data.csv \
  -o data.csv
```

“ ⚙️ En interne, cette commande **utilise le protocole S3 sur HTTP (REST)**. ”

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

2 L'API S3 (Amazon Simple Storage Service API)

Avantages

- Standard **interopérable et universel**.
- Compatible avec **tous les langages (Python boto3, Java SDK, Go, etc.)**.
- Support de fonctionnalités avancées : versioning, multipart upload, presigned URL.
- Facilement intégrable dans des outils Big Data (Spark, Airflow, Hive, etc.).

Limites

- Basé sur HTTP → latence un peu plus élevée que les protocoles binaires.
- Pas de streaming bidirectionnel (chaque appel = une requête complète).

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Principe

gRPC est un protocole moderne développé par **Google**, basé sur **HTTP/2** et le format binaire **Protocol Buffers (Protobuf)**.
Contrairement à REST, il **ne manipule pas de ressources (URL)** mais **appelle directement des méthodes de service**.

> gRPC est **plus performant que REST** pour les communications à haut débit, notamment entre microservices.

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Exemple de requête gRPC

Définition dans un fichier `.proto` :

```
service StorageService {
  rpc Upload (UploadRequest) returns (UploadResponse);
}

message UploadRequest {
  string bucket = 1;
  string filename = 2;
  bytes content = 3;
}
```

Appel depuis un client Java/Python :

```
stub.Upload(UploadRequest(bucket="bronze", filename="file.csv", content=data))
```

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Comparaison avec REST

Critère	REST / S3 API	gRPC
Format	Texte (JSON, XML)	Binaire (Protobuf)
Transport	HTTP/1.1	HTTP/2
Performance	Moyenne	Très élevée
Streaming	❌ Non	✅ Oui (client / serveur / bidirectionnel)
Compatibilité	Universelle	Spécifique aux clients gRPC
Usage typique	API Web, stockage	Microservices, transfert interne

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Avantages de gRPC

- Sérialisation **binaire compacte** (moins de bande passante).
- Support natif du **streaming bidirectionnel**.
- Parfait pour les **échanges entre microservices** (ex. ingestion → traitement).

Limites

- Moins accessible pour les intégrations externes (outils BI, ETL classiques).
- Besoin de générer des stubs à partir du `.proto`.
- Pas aussi universel que l'API S3 REST.

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Résumé synthétique

Protocole	Nature	Format	Usage typique	Exemple d'outil
REST	HTTP	JSON / XML	API Web, CRUD simple	curl, Postman
S3 API	RESTful (HTTP/1.1)	XML / JSON	Stockage objet	AWS CLI, mc, boto3
gRPC	RPC (HTTP/2)	Protobuf (binaire)	Microservices, Data Streaming	grpcurl, Java/Python SDK

Architecture du stockage objet

Protocoles d’accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Schéma simplifié

Architecture du stockage objet

Protocoles d'accès : S3 API, REST, gRPC

3 Le protocole gRPC (Google Remote Procedure Call)

Exemple pratique mixte avec MinIO

Objectif	Outil	Protocole	Exemple
Upload manuel	`mc cp`	S3 API	`mc cp data.csv local/bronze/`
Intégration Spark	`spark.read.csv("s3a://...")`	S3 API (REST)	Lecture directe
Microservice ingestion	gRPC client	gRPC	Upload binaire en streaming

Architecture du stockage objet

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

Trois niveaux de protection

1. **Sécurité des accès** → qui peut faire quoi ?
2. **Sécurité des données** → comment sont-elles protégées ?
3. **Sécurité des opérations** → comment surveiller et auditer ?

Architecture du stockage objet

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

a. Contrôle d'accès : IAM et politiques

MinIO, comme AWS S3, repose sur un modèle **IAM (Identity and Access Management)**.
Chaque utilisateur ou application se voit attribuer :

- un **compte** (avec clés d'accès et secret key),
- une ou plusieurs **politiques (policies)** JSON définissant les droits.

💡 **Exemple d'une policy en JSON :**

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:PutObject"],
      "Resource": ["arn:aws:s3:::datalake-bronze/*"]
    }
  ]
}
```

Architecture du stockage objet

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

Commandes MinIO associées :

```
# Création d'un utilisateur
mc admin user add local dataeng secret12345

# Application d'une policy d'accès
mc admin policy set local readwrite user=dataeng
```

“ ✅ **Bonnes pratiques :**

- Un utilisateur = un usage ou une application.
- Politiques granulaires par **bucket** (ex : bronze = lecture seule).
- Rotation régulière des clés d'accès. ”

Architecture du stockage objet

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

b. Sécurité réseau et chiffrement

Chiffrement en transit (SSL/TLS)

- Les échanges entre clients et serveur MinIO se font via **HTTPS**.
- MinIO peut être configuré avec un certificat SSL (Let's Encrypt, interne...).

```
MINIO_SERVER_URL=https://minio.lab.local
MINIO_CERT_DIR=/etc/minio/certs/
```

Architecture du stockage objet

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

b. Sécurité réseau et chiffrement

Chiffrement au repos (Server-Side Encryption)

- Les objets sont **automatiquement chiffrés** avant d'être écrits sur disque.
- MinIO supporte :
  - le chiffrement via clé **KMS** (Key Management Service),
  - ou une clé locale dans `/root/.minio.sys/config`.

> Pour un Data Lake sensible (finance, santé), toujours **activer le chiffrement côté serveur**.

Architecture du stockage objet

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

c. Gestion des logs et audit

MinIO trace toutes les opérations via :

- **mc admin trace** (monitoring en temps réel),
- **mc admin audit** (historique complet des actions).

```
mc admin trace -a local
```

“ Ces logs peuvent être envoyés vers :

- un SIEM (Splunk, ELK),
- ou un outil d’audit interne pour la conformité RGPD / ISO 27001. ”

Architecture du stockage objet

Sécurité et gouvernance des données

2 Gouvernance des données

Définition

La **gouvernance des données** regroupe l'ensemble des **règles, processus et rôles** permettant :

- de garantir la **qualité**, la **sécurité** et la **traçabilité** des données,
- de définir **qui en est responsable**,
- et d'assurer la **conformité réglementaire**.

Architecture du stockage objet

Sécurité et gouvernance des données

2 Gouvernance des données

a. Traçabilité et cycle de vie

- **Versioning** : historique complet des modifications.
- **Lifecycle policies** : suppression automatique des versions anciennes.
- **Tags / metadata** : identification du propriétaire et du niveau de sensibilité.

```
# Exemple : politique de suppression après 90 jours
mc ilm add local/datalake-bronze --expire-days 90
```

> *Dans un Data Lake, on ne supprime pas arbitrairement : on planifie la rétention.*

Architecture du stockage objet

Sécurité et gouvernance des données

2 Gouvernance des données

b. Classification et sensibilité

Les données doivent être **classifiées** selon leur nature :

Niveau	Exemple	Protection requise
Publique	Données ouvertes, rapports anonymisés	Lecture libre
Interne	Logs techniques, statistiques anonymes	Authentification requise
Confidentielle	Données clients, RH	Chiffrement, IAM strict
Sensible / Réglementée	Données médicales, bancaires	Chiffrement + audit complet

Architecture du stockage objet

Sécurité et gouvernance des données

2 Gouvernance des données

c. Conformité et RGPD

Le Data Lake doit respecter les principes suivants :

- **Finalité** : ne collecter que ce qui est nécessaire.
- **Droit à l'oubli** : suppression sur demande.
- **Provenance et traçabilité** : savoir d'où vient la donnée.
- **Localisation** : hébergement dans des régions autorisées (UE).

> *MinIO peut être déployé on-premise pour garantir la souveraineté des données.*

Architecture du stockage objet

Sécurité et gouvernance des données

2 Gouvernance des données

d. Catalogue de données et qualité

- Intégrer un **catalogue de données** (Apache Atlas, DataHub, Amundsen).
- Définir des **rôles de gouvernance** :
  - *Data Owner* (propriétaire),
  - *Data Steward* (qualité),
  - *Data Engineer* (technicien).
- Suivre des **indicateurs de qualité** : complétude, cohérence, fraîcheur.

Architecture du stockage objet

Sécurité et gouvernance des données

Résumé synthétique

Domaine	Objectif	Exemple d'outil / pratique
IAM & policies	Gérer les accès et rôles	`mc admin user`, `mc admin policy`
Chiffrement	Protéger les données en transit / repos	TLS, SSE, KMS
Audit & logs	Tracer les actions	`mc admin trace`, Splunk
Lifecycle	Gérer la rétention / suppression	`mc ilm add`
Catalogue	Documenter les données	Apache Atlas, DataHub
Conformité	Respecter RGPD / ISO 27001	Politique interne, KMS

Architecture du stockage objet

Sécurité et gouvernance des données

Bonnes pratiques

✅ Activer le **chiffrement** dès le déploiement.
✅ Définir une **politique IAM stricte** dès la création du Data Lake.
✅ Utiliser un **KMS externe** pour la gestion des clés (HashiCorp Vault, AWS KMS).
✅ Conserver les **logs d'accès** sur un stockage séparé.
✅ Documenter chaque bucket dans un **catalogue de données**.
✅ Planifier la **rétention et suppression automatique** des objets obsolètes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à *MinIO*

Introduction à MinIO

Présentation de la solution open-source MinIO

1 Qu'est-ce que MinIO ?

Définition

MinIO est une solution **open-source** de **stockage objet distribué**, écrite en **Go**, conçue pour être **rapide, légère et entièrement compatible avec l'API S3** d'AWS.
Il permet de stocker **des fichiers de toute taille** (de quelques Ko à plusieurs To) dans des **buckets**, et d'y accéder via :

- une **console web** (interface graphique),
- une **API S3** standard,
- ou un **client en ligne de commande (`mc`)**.

Introduction à MinIO

Présentation de la solution open-source MinIO

1 Qu'est-ce que MinIO ?

Philosophie de MinIO

« Faire du S3, mais en mieux : rapide, portable, et libre ! »

MinIO vise la **simplicité et la performance**, tout en restant **cloud-agnostique**, c'est-à-dire utilisable :

- sur une **machine locale** (Docker, VM),
- en **cluster multi-nœuds on-premise**,
- ou dans un **environnement hybride / multi-cloud**.

2 Caractéristiques principales

Domaine	Description
Licence	Open-source sous **GNU AGPL v3** (possibilité commerciale via MinIO SUBNET)
Langage	Écrit en **Go** (GoLang) → exécutable unique, rapide et compact
Compatibilité	**100 % API S3** (Amazon S3, AWS CLI, boto3, Spark, etc.)
Déploiement	Standalone, Docker Compose, Kubernetes (Operator), ou bare-metal
Performance	Optimisé I/O, multi-thread, support **erasure coding**
Résilience	Réplication, versioning, haute disponibilité (HA)
Sécurité	Chiffrement, TLS/SSL, IAM, audit
Administration	Console Web UI + CLI (`mc admin`)
Écosystème	Intégration native avec Spark, Presto, Kafka, Airflow, etc.

Introduction à MinIO

Présentation de la solution open-source MinIO

3 Exemple d'installation rapide (standalone Docker)

```
docker run -d --name minio \
  -p 9000:9000 -p 9001:9001 \
  -e MINIO_ROOT_USER=minioadmin \
  -e MINIO_ROOT_PASSWORD=minioadmin \
  quay.io/minio/minio server /data --console-address ":9001"
```

Accès :

- Console Web → `http://localhost:9001`
- API S3 → `http://localhost:9000`
- Identifiants par défaut : `minioadmin / minioadmin`

Introduction à MinIO

Présentation de la solution open-source MinIO

3 Exemple d'installation rapide (standalone Docker)

Client en ligne de commande

MinIO fournit un outil officiel appelé `mc` **(MinIO Client)** :

```
mc alias set local http://localhost:9000 minioadmin minioadmin
mc mb local/datalake-bronze
mc cp data.csv local/datalake-bronze/
```

> 💡 `mc` est à MinIO ce que `aws cli` est à AWS : un outil de gestion complet (buckets, users, policies, audit…).

Introduction à MinIO

Présentation de la solution open-source MinIO

🌍 4 Positionnement de MinIO dans l'écosystème

Plateforme	Type	Points forts
AWS S3	Cloud propriétaire	Solution managée, intégrée AWS
Azure ADLS	Cloud propriétaire	Sécurité fine, intégration Azure
GCP Storage	Cloud propriétaire	Haute performance, simplicité
MinIO	Open-source / On-prem / multi-cloud	Portable, rapide, compatible S3, économique

“ 🔗 *MinIO s'impose comme une alternative « cloud privé S3 » :*
il apporte les mêmes fonctionnalités qu'AWS S3, mais dans ton propre datacenter. ”

Introduction à MinIO

Présentation de la solution open-source MinIO

5 Cas d'usage typiques

Domaine	Exemple d'utilisation de MinIO
Data Lake	Stockage des zones Bronze / Silver / Gold
Machine Learning	Stockage des datasets d'entraînement
Backup & Archivage	Cible de sauvegarde chiffrée (Veeam, Restic…)
Applications Cloud-Native	Stockage de fichiers pour microservices
Hybrid Cloud	Pont entre stockage local et S3/Azure/GCS

Introduction à MinIO

Présentation de la solution open-source MinIO

6 Schéma simplifié de l'architecture MinIO

Introduction à MinIO

Présentation de la solution open-source MinIO

7️⃣ Pourquoi choisir MinIO pour un Data Lake ?

✅ **Compatibilité totale** avec l'écosystème S3 (Kafka, Spark, Airflow, dbt…).
✅ **Contrôle complet** de la donnée : déploiement sur ton infrastructure.
✅ **Performances élevées** pour le streaming et les traitements distribués.
✅ **Sécurité intégrée** (chiffrement, IAM, audit).
✅ **Coût maîtrisé** et **open-source** (pas de dépendance fournisseur).

Introduction à MinIO

Avantages de MinIO

1️⃣ Haute disponibilité (High Availability – HA)

Principe

MinIO offre une **haute disponibilité native**, grâce à son **architecture distribuée** et à l'**erasure coding**.
Il peut être déployé sur plusieurs nœuds (machines physiques, VMs ou conteneurs) afin de garantir la **résilience** du stockage même en cas de panne partielle.

Fonctionnement

- Les objets sont **fragmentés en blocs (data + parity)**.
- Ces blocs sont répartis sur plusieurs disques et nœuds.
- En cas de défaillance d'un disque ou d'un serveur, MinIO **reconstruit les données manquantes** à partir des fragments restants.

Exemple d'architecture distribuée (4 nœuds)

```
docker run -d --name minio1 \
  -p 9001:9000 -e MINIO_ROOT_USER=minio -e MINIO_ROOT_PASSWORD=123456 \
  quay.io/minio/minio server http://minio{1...4}/data --console-address ":9001"
```

Introduction à MinIO

Avantages de MinIO

1️⃣ Haute disponibilité (High Availability – HA)

Avantages concrets

- ✅ Tolérance aux pannes : perte d'un disque ou d'un nœud sans perte de données.
- ✅ Répartition automatique de la charge.
- ✅ Réplication possible entre datacenters (site A → site B).
- ✅ Maintenance facilitée sans interruption de service.

Schéma simplifié :

Introduction à MinIO

Avantages de MinIO

2 Compatibilité totale avec l'API S3

Principe

MinIO implémente **intégralement l'API S3** d'Amazon Web Services, ce qui garantit la **compatibilité complète** avec :

- tous les SDK AWS (Java, Python boto3, Go, Node.js…),
- les outils DevOps (AWS CLI, Terraform, Ansible),
- et les frameworks Big Data (Spark, Airflow, Kafka, dbt, Trino…).

Introduction à MinIO

Avantages de MinIO

2 Compatibilité totale avec l'API S3

Exemples d'intégration

- **Spark** :

```
df = spark.read.csv("s3a://datalake-bronze/clients.csv")
```

- **Airflow** :

```
s3_hook = S3Hook(aws_conn_id="minio_conn")
s3_hook.load_file("data.csv", key="bronze/data.csv", bucket_name="datalake-bronze")
```

- **AWS CLI / MinIO Client (mc)** :

```
mc alias set local http://localhost:9000 minioadmin minioadmin
mc ls local/
```

> 💬 *Toute application compatible S3 fonctionne sans modification avec MinIO.*

Introduction à MinIO

Avantages de MinIO

2️⃣ Compatibilité totale avec l'API S3

✅ Avantages concrets

- Aucune dépendance à un fournisseur cloud.
- Portabilité complète entre AWS, Azure, GCP et on-premise.
- Migration fluide d'un environnement à un autre.
- Support universel de l'écosystème data.

Introduction à MinIO

Avantages de MinIO

3 Performance et scalabilité

Principe

MinIO est conçu pour être **ultra-performant** et **scalable horizontalement**.
Son moteur, écrit en **Go**, tire parti :

- du **multithreading natif**,
- de la **parallélisation I/O**,
- et de **l'optimisation réseau HTTP/2**.

Résultats typiques

- Performances comparables voire supérieures à AWS S3.
- “ 170 Go/s en lecture et > 120 Go/s en écriture sur cluster de 32 nœuds. ”
- Temps de réponse inférieur à 1 ms sur requêtes simples.

Introduction à MinIO

Avantages de MinIO

3 Performance et scalabilité

Exemple d'usage concret

Dans un **pipeline Spark** :

- Lecture simultanée de 500 fichiers Parquet depuis MinIO.
- Traitement parallèle sur 10 workers Spark.
- Résultat agrégé écrit en format Parquet compressé dans `s3a://lake-gold/`.

> **Gain de performance** : 3 à 5× plus rapide que NFS.

Introduction à MinIO

Avantages de MinIO

4 Simplicité de déploiement et d'administration

Principe

MinIO est **léger, autonome et facile à administrer**, que ce soit :

- en **mode standalone** (1 conteneur),
- ou en **mode cluster** (via Docker Compose, Kubernetes, Ansible).

Installation minimale

Un seul binaire suffit :

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
./minio server /data
```

Console d'administration intuitive

- Interface web moderne sur `http://localhost:9001`.
- Gestion graphique des buckets, policies, utilisateurs, versioning, etc.
- Audit et monitoring en temps réel.

Introduction à MinIO

Avantages de MinIO

4 Simplicité de déploiement et d'administration

Simplicité DevOps

- Fichiers YAML prêts à l'emploi pour Docker Compose ou Kubernetes.
- Intégration avec Terraform (`minio/minio` provider).
- Support API REST complet pour automatisation CI/CD.

Avantages concrets

- Installation en < 1 min.
- Aucun composant externe nécessaire.
- Administration centralisée (CLI + Web UI).
- Migration rapide (copie buckets via `mc mirror`).

Introduction à MinIO

Avantages de MinIO

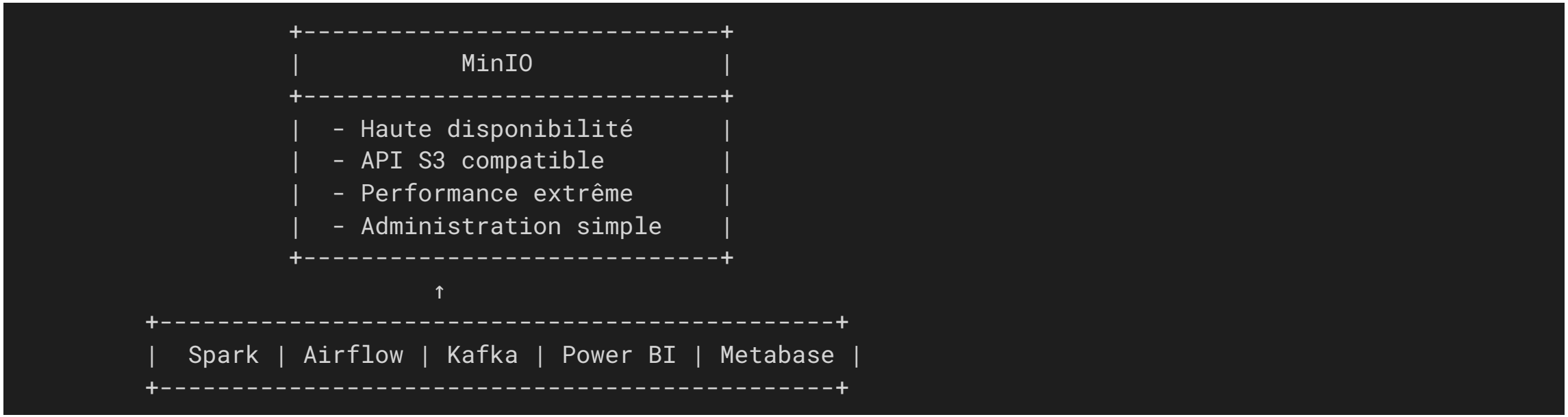
Résumé global des avantages

Atout	Description	Bénéfice concret
Haute disponibilité (HA)	Architecture distribuée + Erasure Coding	Résilience et tolérance aux pannes
Compatibilité S3	API standard du marché	Intégration immédiate avec tout l'écosystème
Performance	Moteur Go optimisé, I/O parallèle	Rapidité et scalabilité
Simplicité	Déploiement léger, console web	Facilité d'usage et administration rapide

Introduction à MinIO

Avantages de MinIO

Schéma visuel de synthèse



Introduction à MinIO

Installation de MinIO en mode Standalone

🐳 1️⃣ Installation avec Docker

Prérequis

- Docker installé (`docker -v`)
- Port 9000 (API) et 9001 (console Web) disponibles
- 1 répertoire local pour les données, par exemple `/data` ou `~/minio-data`

Commande simple

```
docker run -d --name minio \
  -p 9000:9000 -p 9001:9001 \
  -e MINIO_ROOT_USER=minioadmin \
  -e MINIO_ROOT_PASSWORD=minioadmin \
  -v ~/minio-data:/data \
  quay.io/minio/minio server /data --console-address ":9001"
```

Introduction à MinIO

Installation de MinIO en mode Standalone

🐋 1️⃣ Installation avec Docker

Explications des options

Option	Description
`-p 9000:9000`	Expose l'API S3 (accès programmatique)
`-p 9001:9001`	Expose la console d'administration web
`-e MINIO_ROOT_USER`	Nom d'utilisateur administrateur
`-e MINIO_ROOT_PASSWORD`	Mot de passe administrateur
`-v ~/minio-data:/data`	Volume local où seront stockés les objets
`server /data`	Lance le serveur sur le répertoire `/data`
`--console-address ":9001"`	Définit le port pour la console graphique

Introduction à MinIO

Installation de MinIO en mode Standalone

🐳 1️⃣ Installation avec Docker

✅ Vérification

```
docker ps
```

→ Doit afficher :

```
CONTAINER ID   IMAGE                 PORTS
abcd1234       quay.io/minio/minio   0.0.0.0:9000->9000/tcp, 0.0.0.0:9001->9001/tcp
```

Accès

- Interface Web (console d'administration) →
  👉 `http://localhost:9001`
  Login : `minioadmin / minioadmin`
- API S3 (accès programmatique) →
  👉 `http://localhost:9000`

Introduction à MinIO

Installation de MinIO en mode Standalone

2️⃣ Installation via le binaire CLI (Linux ou VM)

Téléchargement du binaire

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
sudo mv minio /usr/local/bin/
```

Création d'un répertoire de données

```
sudo mkdir -p /data/minio
sudo chown $USER:$USER /data/minio
```

Lancement du serveur MinIO

```
export MINIO_ROOT_USER=minioadmin
export MINIO_ROOT_PASSWORD=minioadmin
minio server /data/minio --console-address ":9001"
```

Résultat attendu :

```
API: http://127.0.0.1:9000  Console: http://127.0.0.1:9001
RootUser: minioadmin  RootPass: minioadmin
```

Introduction à MinIO

Installation de MinIO en mode Standalone

2️⃣ Installation via le binaire CLI (Linux ou VM)

Accès identique

- **Console web** : http://localhost:9001
- **API S3** : http://localhost:9000

Arborescence créée

```
/data/minio/
 ├── .minio.sys/          (métadonnées internes)
 └── datalake-bronze/     (bucket créé via mc)
      └── data.csv        (objet importé)
```

Introduction à MinIO

Installation de MinIO en mode Standalone

3 Configuration complémentaire (optionnelle)

Activer le HTTPS

Créer un dossier `/etc/minio/certs/` et y placer un certificat TLS (`public.crt`, `private.key`).

```
sudo mkdir -p /etc/minio/certs
sudo cp public.crt private.key /etc/minio/certs/
```

Lancer ensuite :

```
minio server /data/minio --certs-dir /etc/minio/certs --console-address ":9001"
```

Intégrer le monitoring Prometheus

```
mc admin config set local/ prometheus auth_type=public
```

→ Les métriques seront disponibles sur `http://localhost:9000/minio/v2/metrics/cluster`.

Introduction à MinIO

Installation de MinIO en mode Standalone

3 Configuration complémentaire (optionnelle)

Résumé comparatif des deux approches

Méthode	Environnement	Avantage principal	Idéal pour
Docker	Conteneurisé	Rapide, isolé, réutilisable	Lab local, démo
Binaire CLI	Système nu (VM, Linux, WSL)	Plus proche de la prod	Environnements on-premise

Introduction à MinIO

Installation de MinIO en mode Standalone

3 Configuration complémentaire (optionnelle)

Schéma d'installation standalone

Introduction à MinIO

Installation de MinIO en mode Standalone

Bonnes pratiques de démarrage

✅ Modifier les identifiants par défaut.
✅ Monter un volume persistant (`-v ~/minio-data:/data`).
✅ Tester la compatibilité S3 avec un script Python boto3.
✅ Documenter l'URL, les ports et les utilisateurs dans ton environnement.

Introduction à MinIO

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

Trois niveaux de protection

1. **Sécurité des accès** → qui peut faire quoi ?
2. **Sécurité des données** → comment sont-elles protégées ?
3. **Sécurité des opérations** → comment surveiller et auditer ?

a. Contrôle d'accès : IAM et politiques

MinIO, comme AWS S3, repose sur un modèle **IAM (Identity and Access Management)**.
Chaque utilisateur ou application se voit attribuer :

- un **compte** (avec clés d'accès et secret key),
- une ou plusieurs **politiques (policies)** JSON définissant les droits.

Introduction à MinIO

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

💡 Exemple d'une policy en JSON :

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:PutObject"],
      "Resource": ["arn:aws:s3:::datalake-bronze/*"]
    }
  ]
}
```

Introduction à MinIO

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

Commandes MinIO associées :

```
# Création d'un utilisateur
mc admin user add local dataeng secret12345

# Application d'une policy d'accès
mc admin policy set local readwrite user=dataeng
```

“ ✅ **Bonnes pratiques :**

- Un utilisateur = un usage ou une application.
- Politiques granulaires par **bucket** (ex : bronze = lecture seule).
- Rotation régulière des clés d'accès. ”

Introduction à MinIO

Sécurité et gouvernance des données

1 Sécurité des données dans un stockage objet

b. Sécurité réseau et chiffrement

Chiffrement en transit (SSL/TLS)

- Les échanges entre clients et serveur MinIO se font via **HTTPS**.
- MinIO peut être configuré avec un certificat SSL (Let's Encrypt, interne...).

```
MINIO_SERVER_URL=https://minio.lab.local
MINIO_CERT_DIR=/etc/minio/certs/
```

Introduction à MinIO

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

b. Sécurité réseau et chiffrement

Chiffrement au repos (Server-Side Encryption)

- Les objets sont **automatiquement chiffrés** avant d'être écrits sur disque.
- MinIO supporte :
  - le chiffrement via clé **KMS** (Key Management Service),
  - ou une clé locale dans `/root/.minio.sys/config`.

> Pour un Data Lake sensible (finance, santé), toujours **activer le chiffrement côté serveur**.

Introduction à MinIO

Sécurité et gouvernance des données

1️⃣ Sécurité des données dans un stockage objet

🔍 c. Gestion des logs et audit

MinIO trace toutes les opérations via :

- **mc admin trace** (monitoring en temps réel),
- **mc admin audit** (historique complet des actions).

```
mc admin trace -a local
```

“ 🔎 Ces logs peuvent être envoyés vers :

- un SIEM (Splunk, ELK),
- ou un outil d’audit interne pour la conformité RGPD / ISO 27001. ”

Introduction à MinIO

Sécurité et gouvernance des données

2 Gouvernance des données

Définition

La **gouvernance des données** regroupe l'ensemble des **règles, processus et rôles** permettant :

- de garantir la **qualité**, la **sécurité** et la **traçabilité** des données,
- de définir **qui en est responsable**,
- et d'assurer la **conformité réglementaire**.

Introduction à MinIO

Sécurité et gouvernance des données

2 Gouvernance des données

a. Traçabilité et cycle de vie

- **Versioning** : historique complet des modifications.
- **Lifecycle policies** : suppression automatique des versions anciennes.
- **Tags / metadata** : identification du propriétaire et du niveau de sensibilité.

```
# Exemple : politique de suppression après 90 jours
mc ilm add local/datalake-bronze --expire-days 90
```

> 💡 *Dans un Data Lake, on ne supprime pas arbitrairement : on planifie la rétention.*

Introduction à MinIO

Sécurité et gouvernance des données

2 Gouvernance des données

b. Classification et sensibilité

Les données doivent être **classifiées** selon leur nature :

Niveau	Exemple	Protection requise
Publique	Données ouvertes, rapports anonymisés	Lecture libre
Interne	Logs techniques, statistiques anonymes	Authentification requise
Confidentielle	Données clients, RH	Chiffrement, IAM strict
Sensible / Réglementée	Données médicales, bancaires	Chiffrement + audit complet

Introduction à MinIO

Sécurité et gouvernance des données

2 Gouvernance des données

c. Conformité et RGPD

Le Data Lake doit respecter les principes suivants :

- **Finalité** : ne collecter que ce qui est nécessaire.
- **Droit à l'oubli** : suppression sur demande.
- **Provenance et traçabilité** : savoir d'où vient la donnée.
- **Localisation** : hébergement dans des régions autorisées (UE).

> *MinIO peut être déployé on-premise pour garantir la souveraineté des données.*

Introduction à MinIO

Sécurité et gouvernance des données

2 Gouvernance des données

d. Catalogue de données et qualité

- Intégrer un **catalogue de données** (Apache Atlas, DataHub, Amundsen).
- Définir des **rôles de gouvernance** :
  - *Data Owner* (propriétaire),
  - *Data Steward* (qualité),
  - *Data Engineer* (technicien).
- Suivre des **indicateurs de qualité** : complétude, cohérence, fraîcheur.

Introduction à MinIO

Sécurité et gouvernance des données

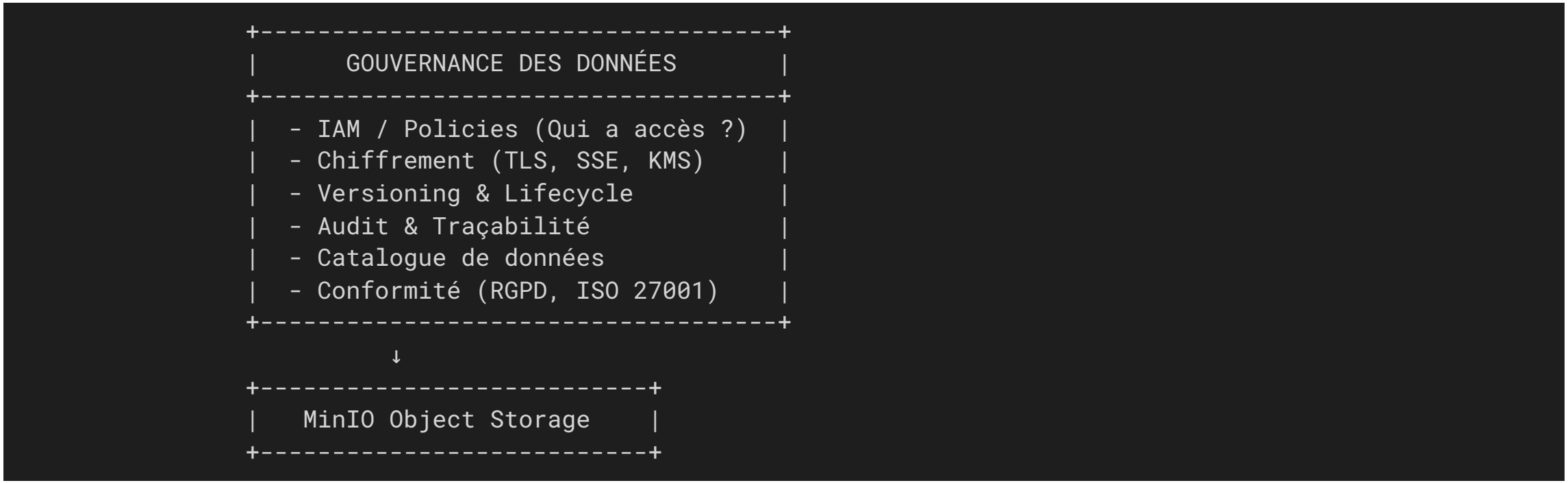
Résumé synthétique

Domaine	Objectif	Exemple d'outil / pratique
IAM & policies	Gérer les accès et rôles	`mc admin user`, `mc admin policy`
Chiffrement	Protéger les données en transit / repos	TLS, SSE, KMS
Audit & logs	Tracer les actions	`mc admin trace`, Splunk
Lifecycle	Gérer la rétention / suppression	`mc ilm add`
Catalogue	Documenter les données	Apache Atlas, DataHub
Conformité	Respecter RGPD / ISO 27001	Politique interne, KMS

Introduction à MinIO

Sécurité et gouvernance des données

Schéma synthétique



Introduction à MinIO

Sécurité et gouvernance des données

Bonnes pratiques

✅ Activer le **chiffrement** dès le déploiement.
✅ Définir une **politique IAM stricte** dès la création du Data Lake.
✅ Utiliser un **KMS externe** pour la gestion des clés (HashiCorp Vault, AWS KMS).
✅ Conserver les **logs d'accès** sur un stockage séparé.
✅ Documenter chaque bucket dans un **catalogue de données**.
✅ Planifier la **rétention et suppression automatique** des objets obsolètes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

MinIO distribué

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

1 Principe général du cluster MinIO

MinIO peut fonctionner :

- en **mode standalone** : un seul serveur / disque,
- ou en **mode distribué (cluster)** : plusieurs nœuds et plusieurs disques.

> Le mode distribué permet à MinIO d'assurer la **haute disponibilité**, la **répartition de charge** et la **tolérance aux pannes** grâce à la technologie d'**erasure coding**.

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

1 Principe général du cluster MinIO

Architecture logique d'un cluster

Chaque nœud **héberge un ou plusieurs disques**, et MinIO considère **l'ensemble des volumes** comme **un seul espace de stockage unifié**.

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

2 L'erasure coding (codage d'effacement)

Définition

L'**erasure coding** est une méthode de **protection des données par redondance intelligente**.
Plutôt que de faire une **simple copie (réplication)**, MinIO **découpe les objets en fragments (shards)**, puis **ajoute des fragments de parité**.

> 📦 Exemple :
> Un fichier de 100 Mo peut être découpé en **8 fragments de données** et **8 fragments de parité**, distribués sur plusieurs disques.

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

2 L'erasure coding (codage d'effacement)

Principe mathématique

MinIO utilise un schéma *(k, m)* :

- **k** = nombre de fragments de données,
- **m** = nombre de fragments de parité.

Un objet est **reconstruit tant que k fragments sont disponibles**,
ce qui permet de **survivre à la perte de m disques ou nœuds**.

“ Exemple : (k=4, m=2)

- 4 blocs de données + 2 blocs de parité.
- Le cluster peut perdre **2 disques** sans perte d'information. ”

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

2️⃣ L'erasure coding (codage d'effacement)

Exemple visuel (Erasure Coding 4+2)

```
Fichier original → découpé en 4 blocs de données :
  D1  D2  D3  D4

MinIO génère 2 blocs de parité :
  P1  P2

Distribution sur 6 disques :
  Disk1 : D1
  Disk2 : D2
  Disk3 : D3
  Disk4 : D4
  Disk5 : P1
  Disk6 : P2
```

> ✅ Si Disk3 et Disk5 tombent en panne, MinIO **reconstruit D3 et P1** automatiquement à partir des autres fragments.

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

2 L'erasure coding (codage d'effacement)

Avantage sur la réplication classique

Méthode	Description	Espace perdu	Tolérance
Réplication	Copie intégrale des fichiers (x3, x5…)	66 % (pour x3)	N-1
Erasure Coding	Découpage en blocs + parité (k+m)	20–40 %	m disques

Résultat : **meilleure protection** avec **moins d'espace gaspillé**.

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

3 Les shards : découpage et distribution

Chaque objet est **découpé en plusieurs shards (fragments)**, qui sont :

- répartis sur les disques disponibles,
- stockés avec des métadonnées précises (index, version, hash),
- vérifiés régulièrement pour détecter toute corruption.

Exemple :

```
Objet : transactions_2025_11_07.parquet
→ 8 shards (6 data + 2 parity)
→ distribués sur 8 disques différents
```

Les shards sont ensuite **réassemblés à la lecture** :

```
Read request → MinIO recompose l'objet à partir des fragments nécessaires.
```

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

4 Comportement du cluster

Écriture

- Lorsqu'un client écrit un fichier, MinIO découpe → encode → distribue les shards.
- Chaque disque reçoit une portion unique de l'objet.

Lecture

- À la lecture, MinIO assemble les fragments disponibles.
- Si un disque est manquant, les blocs manquants sont reconstruits en temps réel.

Reconstruction automatique

Lorsqu'un disque est remplacé :

1. MinIO détecte le nouveau disque.
2. Les fragments manquants sont automatiquement régénérés.
3. Le cluster revient en état "healthy".

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

4 Comportement du cluster

Exemple concret : déploiement 4 nœuds, 4 disques chacun

```
docker run -d --name minio1 \
  -p 9001:9000 \
  -e MINIO_ROOT_USER=minio \
  -e MINIO_ROOT_PASSWORD=minio123 \
  quay.io/minio/minio server \
  http://minio{1...4}/data{1...4} --console-address ":9001"
```

Explication :

- `minio{1...4}` → 4 conteneurs / nœuds.
- `data{1...4}` → 4 disques par nœud.
  Total = 16 disques (ou volumes).

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

5 Tolérance aux pannes selon le nombre de disques

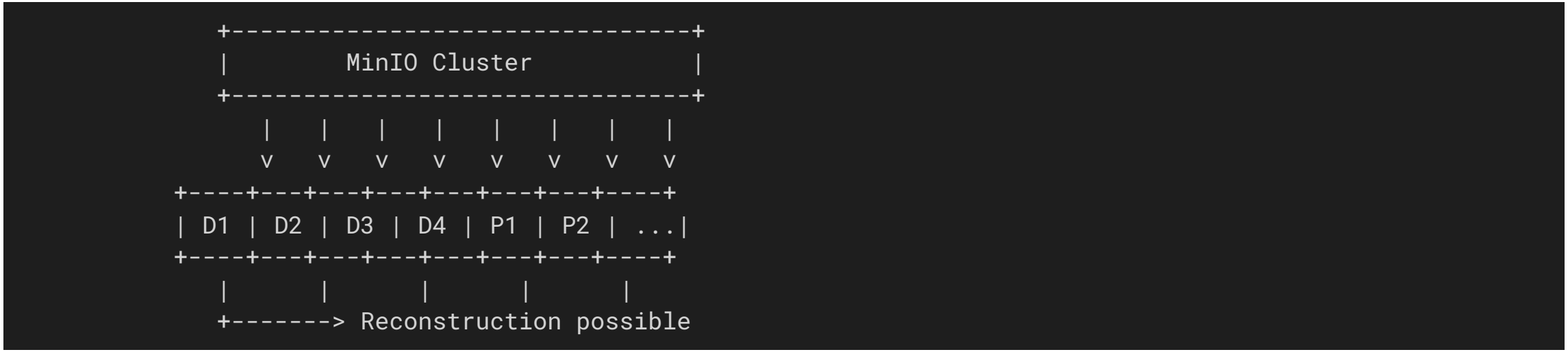
Nombre total de disques	Fragments de données	Fragments de parité	Tolérance aux pannes
4	2	2	2 disques
6	4	2	2 disques
8	6	2	2 disques
12	8	4	4 disques
16	12	4	4 disques

“ 🔐 Plus le cluster est grand, plus la **résilience** et la **scalabilité** augmentent. ”

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

6 Schéma synthétique de l'architecture Erasure Coding



> 💬 *L'erasure coding agit comme une "assurance" : même si certains disques tombent, le cluster reste fonctionnel.*

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

Résumé clé

Concept	Description	Objectif
Erasure Coding	Découpage en blocs + parité	Assure la redondance intelligente
Shards	Fragments physiques de chaque objet	Permettent la distribution et la reconstruction
Cluster distribué	Ensemble de nœuds et disques	Offre scalabilité et haute disponibilité
Self-healing	Reconstruction automatique	Garantit la continuité de service

Introduction à MinIO

Architecture du cluster MinIO (Erasure Coding, Shards)

Résumé clé

Concept	Description	Objectif
Erasure Coding	Découpage en blocs + parité	Assure la redondance intelligente
Shards	Fragments physiques de chaque objet	Permettent la distribution et la reconstruction
Cluster distribué	Ensemble de nœuds et disques	Offre scalabilité et haute disponibilité
Self-healing	Reconstruction automatique	Garantit la continuité de service

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

1️⃣ Déploiement MinIO distribué avec Docker Compose

Contexte

- En environnement de test ou formation, on peut simuler un cluster MinIO complet sur une seule machine grâce à **Docker Compose**.
- Chaque conteneur représente un **nœud MinIO** avec son propre disque virtuel.

Structure du projet

```
minio-cluster/
  ├── docker-compose.yml
  ├── data1/
  ├── data2/
  ├── data3/
  └── data4/
```

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

1 Déploiement MinIO distribué avec Docker Compose

Fichier `docker-compose.yml`

```yaml
version: '3.8'

services:
  minio1:
    image: quay.io/minio/minio:latest
    container_name: minio1
    volumes:
      - ./data1:/data
    environment:
      MINIO_ROOT_USER: minio
      MINIO_ROOT_PASSWORD: minio123
    command: server http://minio{1...4}/data --console-address ":9001"
    ports:
      - "9001:9001"
      - "9000:9000"
    networks:
      - minio_net
```

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

1️⃣ Déploiement MinIO distribué avec Docker Compose

Lancement du cluster

```
docker-compose up -d
```

Vérification

```
docker ps
```

Doit afficher 4 conteneurs MinIO actifs (`minio1` à `minio4`).

Accès

- Console Web : http://localhost:9001
  (ou ports 9002, 9003, 9004 pour chaque nœud)

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

1 Déploiement MinIO distribué avec Docker Compose

Remarques pédagogiques

- Cette configuration crée un cluster **4-nœuds / 4-disques** simulé localement.
- Le cluster peut tolérer **jusqu'à 2 pannes de nœud** grâce à l'erasure coding.
- Chaque service partage le même "namespace" `/data`.
- Les buckets et objets sont accessibles de manière uniforme depuis n'importe quel nœud.

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Contexte

Pour un environnement de production ou d'entreprise, MinIO recommande le **MinIO Kubernetes Operator** ou un **Helm Chart officiel**.
Cela permet de bénéficier :

- de la **scalabilité automatique**,
- du **redémarrage automatique des pods**,
- et de l'**intégration réseau / stockage (PersistentVolumeClaim)**.

Option 1 : déploiement via Helm Chart

📦 Installation du dépôt Helm

```
helm repo add minio https://charts.min.io/
helm repo update
```

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Déploiement du cluster

```
helm install minio-cluster minio/minio \
  --namespace minio --create-namespace \
  --set rootUser=minio,rootPassword=minio123 \
  --set replicas=4 \
  --set persistence.enabled=true \
  --set persistence.size=10Gi
```

Accès

- Console : `kubectl port-forward svc/minio-cluster 9001:9001`
- API : `kubectl port-forward svc/minio-cluster 9000:9000`

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Option 2 : déploiement via le MinIO Operator

Installation

```
kubectl apply -k "github.com/minio/operator/?ref=v6.0.5"
```

🚀 Création d'un tenant (cluster logique MinIO)

```
kubectl minio tenant create minio-tenant1 \
  --servers 4 \
  --volumes 16 \
  --capacity 20Ti \
  --namespace minio-tenant1
```

Visualisation

- Dashboard Operator :
  👉 `kubectl minio proxy` → http://localhost:9090

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Option 2 : déploiement via le MinIO Operator

Structure typique dans Kubernetes

```
Namespace: minio
├── StatefulSet/minio-cluster
├── Service/minio-cluster (port 9000/9001)
├── PVC/minio-pv-claim-0..3
├── Secret/minio-creds
└── ConfigMap/minio-env
```

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Bonnes pratiques Kubernetes

- Toujours utiliser des **PersistentVolumes (PV)** reliés à un **StorageClass** (NFS, Ceph, Longhorn, EBS...).
- Gérer les identifiants via **Secrets** (`kubectl create secret generic`).
- Configurer les probes :

```
livenessProbe:
  httpGet:
    path: /minio/health/live
    port: 9000
readinessProbe:
  httpGet:
    path: /minio/health/ready
    port: 9000
```

- Exposer via un **Ingress** (TLS activé).

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

Comparatif Docker Compose vs Kubernetes

Critère	Docker Compose	Kubernetes
Complexité	Faible	Élevée
Usage typique	Lab, test, développement	Production, cloud
Scalabilité	Limitée	Dynamique (pods & volumes)
Résilience	Manuelle (redémarrage docker)	Automatique (self-healing)
Stockage	Volumes locaux	PersistentVolumeClaim
Monitoring	`mc admin info`	Prometheus + Grafana
Mise à jour	Rebuild containers	Rolling updates

Introduction à MinIO

Déploiement de MinIO distribué : Docker Compose & K8s

✅ Synthèse des acquis

Compétence	Objectif atteint
Déployer un cluster MinIO avec Docker Compose	✓
Déployer un cluster MinIO sur Kubernetes	✓
Comprendre la configuration réseau, volumes, et credentials	✓
Accéder via console web et API S3	✓

Introduction à MinIO

Load Balancing et Scaling dans MinIO

1 Load Balancing (Répartition de charge)

Principe

Dans une architecture **MinIO distribuée**, plusieurs nœuds (instances MinIO) partagent :

- le même **espace de nommage (namespace)**,
- les mêmes **buckets et objets**,
- et le même **point d'accès API S3**.

Le **load balancer** (interne ou externe) a pour rôle de :

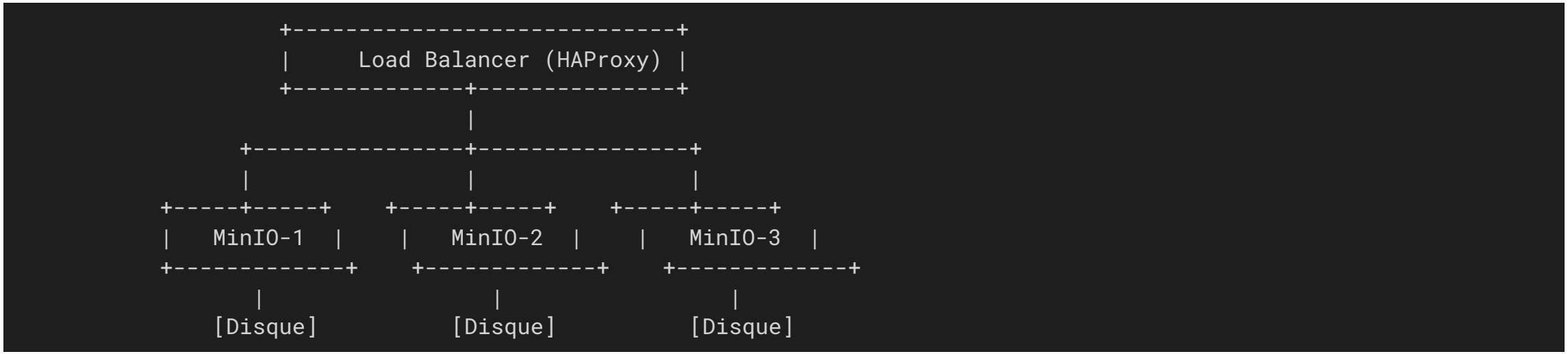
- **distribuer les requêtes clients** vers les différents nœuds du cluster,
- **équilibrer la charge CPU / I/O / réseau**,
- et **gérer la tolérance aux pannes** (failover).

Introduction à MinIO

Load Balancing et Scaling dans MinIO

1 Load Balancing (Répartition de charge)

Schéma conceptuel



Introduction à MinIO

Load Balancing et Scaling dans MinIO

1️⃣ Load Balancing (Répartition de charge)

Implémentation typique du Load Balancer

Option 1 : HAProxy (local ou Docker)

```
# Exemple de configuration haproxy.cfg
frontend minio_front
    bind *:9000
    default_backend minio_backend

backend minio_backend
    balance roundrobin
    server minio1 172.18.0.2:9000 check
    server minio2 172.18.0.3:9000 check
    server minio3 172.18.0.4:9000 check
    server minio4 172.18.0.5:9000 check
```

“ 💡 L’algorithme **round-robin** répartit les requêtes de manière équilibrée sur tous les nœuds. ”

Introduction à MinIO

Load Balancing et Scaling dans MinIO

1 Load Balancing (Répartition de charge)

Implémentation typique du Load Balancer

Option 2 : NGINX (proxy HTTP/REST)

```
upstream minio_backend {
    server minio1:9000;
    server minio2:9000;
    server minio3:9000;
    server minio4:9000;
}

server {
    listen 9000;
    location / {
        proxy_pass http://minio_backend;
    }
}
```

Introduction à MinIO

Load Balancing et Scaling dans MinIO

1 Load Balancing (Répartition de charge)

Implémentation typique du Load Balancer

Option 3 : Kubernetes Service

En environnement Kubernetes :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: minio-cluster
spec:
  type: LoadBalancer
  ports:
    - port: 9000
      targetPort: 9000
  selector:
    app: minio
```

Introduction à MinIO

Load Balancing et Scaling dans MinIO

1️⃣ Load Balancing (Répartition de charge)

✅ Avantages du load balancing

Avantage	Description
Disponibilité	Si un nœud tombe, les autres prennent le relais.
Performance	Répartition équilibrée des requêtes entre serveurs.
Scalabilité	Facilité d'ajouter ou retirer des nœuds sans interruption.
Uniformité	Un seul point d'entrée (API/console) pour les clients.

Introduction à MinIO

Load Balancing et Scaling dans MinIO

2 Scaling (mise à l'échelle)

Principe

Le **scaling** permet d'**augmenter la capacité** du cluster MinIO
en ajoutant de nouveaux **nœuds** ou **disques**, sans interrompre le service.

> MinIO supporte le **scaling horizontal** : tu ajoutes des serveurs supplémentaires dans le cluster.

Introduction à MinIO

Load Balancing et Scaling dans MinIO

2 Scaling (mise à l'échelle)

Types de scaling

Type	Description	Exemple
Horizontal (scale-out)	Ajout de nouveaux nœuds MinIO	Ajouter `minio5`, `minio6`
Vertical (scale-up)	Ajout de disques ou augmentation de taille	Étendre `/data` d'un nœud

Introduction à MinIO

Load Balancing et Scaling dans MinIO

2️⃣ Scaling (mise à l'échelle)

Exemple de scaling horizontal

Avant :

```
minio server http://minio{1...4}/data{1...4}
```

Après (ajout de 2 nœuds supplémentaires) :

```
minio server http://minio{1...6}/data{1...4}
```

Chaque nouveau nœud :

- rejoint automatiquement le cluster,
- synchronise la configuration IAM et buckets,
- rééquilibre les données avec les anciens nœuds.

Introduction à MinIO

Load Balancing et Scaling dans MinIO

2 Scaling (mise à l'échelle)

Architecture distribuée avec scaling

```
Avant :
4 serveurs (16 disques)  →  tolérance 4 pannes

Après scaling :
6 serveurs (24 disques)  →  tolérance 8 pannes
```

> Les performances et la résilience augmentent proportionnellement au nombre de nœuds et disques.

Introduction à MinIO

Load Balancing et Scaling dans MinIO

2️⃣ Scaling (mise à l'échelle)

Bonnes pratiques de scaling

- Toujours ajouter des nœuds **par paires** pour équilibrer l'erasure coding.
- Vérifier la **compatibilité de version** entre les instances.
- Synchroniser les **certificats et fichiers de configuration** (`/root/.minio.sys`).
- Sur Kubernetes : simplement **augmenter le nombre de replicas** dans le StatefulSet :

```
kubectl scale statefulset minio --replicas=6
```

Introduction à MinIO

Load Balancing et Scaling dans MinIO

3 Auto-scaling et supervision

Auto-scaling Kubernetes

MinIO s'intègre avec les contrôleurs Kubernetes :

- **Horizontal Pod Autoscaler (HPA)** pour ajuster les pods selon la charge CPU.
- **Cluster Autoscaler** pour ajouter des nœuds lorsque la capacité de stockage devient critique.

```
kubectl autoscale statefulset minio --min=4 --max=8 --cpu-percent=70
```

Introduction à MinIO

Load Balancing et Scaling dans MinIO

3 Auto-scaling et supervision

Monitoring et équilibrage dynamique

MinIO expose des métriques Prometheus :

```
/minio/v2/metrics/cluster
```

Métriques clés :

- `minio_http_requests_total` → nombre de requêtes par nœud
- `minio_disk_storage_used_bytes` → espace disque utilisé
- `minio_node_online_total` → disponibilité des nœuds

> Ces indicateurs permettent d'**anticiper la saturation** et de **scaler avant incident**.

Introduction à MinIO

Load Balancing et Scaling dans MinIO

3 Auto-scaling et supervision

Résumé comparatif

Fonction	Docker Compose	Kubernetes
Load Balancing	HAProxy / Nginx manuel	Service / Ingress automatique
Scaling horizontal	Ajout manuel de conteneurs	`kubectl scale` dynamique
Monitoring	`mc admin info`	Prometheus + Grafana
Failover	Rebuild containers	Pods redémarrés automatiquement

Introduction à MinIO

Load Balancing et Scaling dans MinIO

3 Auto-scaling et supervision

Schéma synthétique : load balancing & scaling

Introduction à MinIO

Load Balancing et Scaling dans MinIO

3 Auto-scaling et supervision

✅ Synthèse pédagogique

Compétence	Objectif atteint
Comprendre le rôle du Load Balancer	✓
Configurer un équilibrage de charge pour MinIO	✓
Mettre à l'échelle un cluster MinIO (scale-out)	✓
Surveiller et anticiper la montée en charge	✓