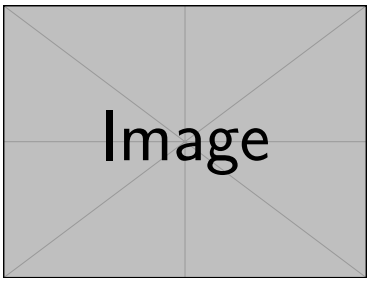
Cours Complet de Data Engineering

Appliqué au Projet Pipeline SIEM

Tous les outils, concepts et principes expliqués en détail



Projet Académique — Janvier 2026

Plateforme ETL d'Agrégation et Normalisation de Logs SIEM

Table des matières

Première partie

Principes Fondamentaux du Data Engineering

Chapitre 1

Qu'est-ce que le Data Engineering ?

1.1 Définition

Data Engineering — Définition

Le **Data Engineering** est la discipline qui consiste à concevoir, construire et maintenir les systèmes et infrastructures permettant de **collecter**, **stocker**, **transformer** et **distribuer** les données à grande échelle. Le Data Engineer construit les *pipelines* qui transportent les données depuis leurs sources brutes jusqu'à leur forme exploitable par les analystes, data scientists et applications métier.

1.2 Le Data Engineer vs les autres rôles Data

Rôle	Responsabilité principale	Outils typiques
Data Engineer	Construire et maintenir les pipelines de données, l'infrastructure, l'ingestion et la transformation	Kafka, Spark, Airflow, SQL, Docker
Data Analyst	Analyser les données pour en extraire des insights business	SQL, Excel, Tableau, Power BI
Data Scientist	Créer des modèles prédictifs et algorithmes ML	Python, scikit-learn, TensorFlow
Data Architect	Concevoir l'architecture globale des systèmes de données	Diagrammes, modélisation, gouvernance
ML Engineer	Déployer et maintenir les modèles ML en production	MLflow, Kubernetes, Docker

1.3 Les 5 piliers du Data Engineering

1. **Ingestion** — Collecter les données depuis les sources (APIs, fichiers, bases de données, flux temps réel)
2. **Stockage** — Stocker les données de manière fiable, scalable et organisée
3. **Transformation** — Nettoyer, enrichir, agréger et normaliser les données
4. **Orchestration** — Automatiser et planifier l'exécution des tâches du pipeline
5. **Restitution** — Rendre les données accessibles via dashboards, APIs ou requêtes

Chapitre 2

Le Pipeline de Données (Data Pipeline)

2.1 Définition

Pipeline de Données

Un **pipeline de données** (ou *data pipeline*) est une séquence automatisée d'étapes qui déplace et transforme les données depuis une ou plusieurs **sources** vers une ou plusieurs **destinations**. Chaque étape du pipeline effectue une opération spécifique : extraction, validation, transformation, chargement, etc.

On peut le comparer à une **chaîne de montage industrielle** : les matières premières (données brutes) entrent d'un côté, passent par différentes stations de traitement, et un produit fini (données exploitables) sort de l'autre côté.

2.2 Types de Pipelines

2.2.1 Pipeline Batch (par lots)

- Les données sont collectées pendant une période, puis traitées en un seul bloc
- Fréquence typique : toutes les heures, quotidien, hebdomadaire
- **Avantages** : Simple, efficient pour de gros volumes, facile à debugger
- **Inconvénients** : Latence élevée (les données ne sont pas disponibles en temps réel)
- **Exemple dans notre projet** : Le DAG Airflow `dag_batch_ingestion` qui récupère les fichiers de logs toutes les heures

2.2.2 Pipeline Streaming (temps réel)

- Les données sont traitées **au fil de l'eau**, dès qu'elles arrivent
- Latence typique : millisecondes à quelques secondes
- **Avantages** : Données disponibles quasi instantanément, réactivité
- **Inconvénients** : Plus complexe, gestion de l'ordre des événements, exactly-once delivery
- **Exemple dans notre projet** : Spark Structured Streaming qui lit depuis Kafka et écrit dans Elasticsearch en continu

2.2.3 Architecture Lambda

Architecture Lambda — Notre choix

L'**Architecture Lambda** combine *batch* et *streaming* en parallèle :

- **Speed Layer** (couche rapide) : traitement temps réel via Spark Streaming + Kafka pour des résultats immédiats mais potentiellement approximatifs
- **Batch Layer** (couche batch) : retraitement périodique complet des données pour des résultats exacts et corrigés
- **Serving Layer** (couche de service) : fusion des deux résultats pour la requête (Elasticsearch + Kibana)

Pourquoi ce choix pour notre SIEM ?

- Le streaming permet de détecter les menaces de sécurité **en temps réel**
- Le batch permet de **retraiter** et corriger les données quotidiennement
- Le SIEM a besoin des deux : alertes immédiates + analyses historiques

2.2.4 Architecture Kappa (alternative)

L'**Architecture Kappa** simplifie Lambda en utilisant **uniquement le streaming**. Toutes les données passent par le même chemin temps réel. Le retraitement se fait en rejouant les événements depuis Kafka.

- **Avantage** : Un seul code à maintenir
- **Inconvénient** : Kafka doit conserver tous les messages (coût de stockage élevé)
- Nous n'avons pas choisi cette approche car le batch offre plus de flexibilité pour les agrégations quotidiennes

Chapitre 3

ETL vs ELT

3.1 ETL — Extract, Transform, Load

ETL

ETL signifie **Extract-Transform-Load** :

1. **Extract** : Extraire les données des sources (APIs, fichiers, bases de données)
2. **Transform** : Transformer les données (nettoyage, normalisation, enrichissement) *avant* de les charger
3. **Load** : Charger les données transformées dans la destination (data warehouse, index, etc.)

Les transformations se font **en dehors** du système de stockage final, typiquement dans un moteur de calcul comme Spark.

ETL dans notre projet SIEM

1. **Extract** : Les logs bruts sont collectés depuis les 4 sources et publiés dans Kafka
2. **Transform** : Spark applique le parsing ECS, l'enrichissement GeoIP, le scoring de sévérité et la déduplication
3. **Load** : Les logs normalisés sont indexés dans Elasticsearch, les logs bruts archivés dans MinIO en Parquet

3.2 ELT — Extract, Load, Transform

ELT inverse l'ordre : les données brutes sont d'abord **chargées** dans le stockage (ex : data warehouse), puis transformées **sur place** grâce à la puissance du système de stockage.

— **Outils ELT typiques** : dbt, Dataform, BigQuery SQL

— **Avantage** : Le data warehouse fait le calcul, pas besoin d'un moteur externe

— **Inconvénient** : Le stockage doit être puissant (Snowflake, BigQuery, etc.)

Pourquoi ETL et non ELT pour notre projet ?

Notre projet utilise **ETL** car :

- Elasticsearch n'est **pas** un moteur de transformation puissant (ce n'est pas un data warehouse)
- Les transformations de parsing multi-format nécessitent la puissance de **Spark**
- Le streaming impose de transformer les données *avant* de les indexer (sinon la latence serait trop élevée)

Chapitre 4

Ingestion de Données

4.1 Définition

Ingestion de Données

L'**ingestion de données** est le processus de collecte et d'importation des données depuis des sources externes vers le système de traitement. C'est la **première étape** de tout pipeline.

Deux modes principaux :

- **Ingestion Batch** : collecte périodique (fichiers, dump de BDD, API paginée)
- **Ingestion Streaming** : collecte continue en temps réel (Kafka, Kinesis, Pub/-Sub)

4.2 Ingestion Batch

Dans notre projet, l'ingestion batch est gérée par un **DAG Airflow** qui s'exécute **toutes les heures** :

- Il scanne un répertoire de fichiers de logs accumulés
- Il lit les fichiers, les valide, et les publie dans les topics Kafka correspondants
- Cela permet de traiter les logs historiques ou les logs qui n'ont pas pu être envoyés en streaming

4.3 Ingestion Streaming

Dans notre projet, l'ingestion streaming se fait via **Apache Kafka** :

- Les scripts `generators/` simulent des sources de logs et publient les événements en temps réel dans Kafka
- Kafka agit comme un **buffer distribué** entre les producteurs (sources) et les consommateurs (Spark)
- Les événements sont disponibles pour le traitement en **quelques millisecondes**

4.4 Patterns d'ingestion importants

4.4.1 Change Data Capture (CDC)

CDC — Change Data Capture

Le **CDC** est une technique qui capture les **changements** (INSERT, UPDATE, DELETE) dans une base de données et les propage en temps réel vers un système cible. L'outil le plus connu est **Debezium**, qui lit le *Write-Ahead Log* (WAL) de bases comme PostgreSQL.

Non utilisé dans notre projet car nos sources sont des fichiers de logs, pas des bases de données. Mais le CDC est fondamental en Data Engineering.

4.4.2 Pub/Sub (Publication/Souscription)

C'est le pattern utilisé par Kafka : les **producteurs** publient des messages dans des *topics*, et les **consommateurs** s'abonnent à ces topics pour recevoir les messages.

Chapitre 5

Stockage de Données

5.1 Les architectures de stockage

5.1.1 Data Lake

Data Lake

Un **Data Lake** est un système de stockage centralisé qui permet de stocker **toutes les données** dans leur **format brut** (structuré, semi-structuré, non structuré). Les données sont stockées « telles quelles » et ne sont transformées que lorsqu'on en a besoin (principe du *schema-on-read*).

Caractéristiques :

- Stockage objet (fichiers) — pas de schéma imposé à l'écriture
- Formats typiques : Parquet, ORC, Avro, JSON, CSV
- Peu coûteux, très scalable
- Technologies : AWS S3, Google Cloud Storage, Azure ADLS, **MinIO**

Dans notre projet : MinIO sert de Data Lake pour archiver les logs bruts au format Parquet, partitionnés par date et type de source.

5.1.2 Data Warehouse

Data Warehouse

Un **Data Warehouse** est un système de stockage optimisé pour l'**analyse** et les **requêtes analytiques** (OLAP). Les données y sont structurées, nettoyées et organisées selon un **schéma prédéfini** (*schema-on-write*).

Caractéristiques :

- Schéma rigide (tables, colonnes typées)
- Optimisé pour les requêtes SQL analytiques (agrégations, jointures)
- Technologies : Snowflake, BigQuery, Redshift, DuckDB

Dans notre projet : Elasticsearch joue un rôle similaire (index structuré, recherche rapide) mais n'est *pas* un data warehouse classique — c'est un moteur de recherche.

5.1.3 Data Lakehouse

Data Lakehouse

Le **Data Lakehouse** est une architecture hybride qui combine les avantages du Data Lake (stockage brut, peu coûteux, flexible) avec ceux du Data Warehouse (transactions ACID, schéma, performance analytique).

Technologies : Delta Lake, Apache Iceberg, Apache Hudi

C'est une tendance majeure depuis 2020. Le Lakehouse permet de faire de l'analytique directement sur le Data Lake sans avoir besoin de copier les données dans un warehouse séparé.

5.2 Schema-on-Read vs Schema-on-Write

Schema-on-Read (Data Lake)	**Schema-on-Write** (Warehouse)
Le schéma est appliqué **à la lecture**	Le schéma est imposé **à l'écriture**
Les données brutes sont stockées telles quelles	Les données doivent être conformes au schéma avant d'être écrites
Plus flexible, accepte tout format	Plus rigide mais plus fiable
Risque de « data swamp » (lac de données inutilisable)	Garantie de qualité des données
MinIO dans notre projet	**Elasticsearch dans notre projet**

5.3 Formats de stockage

5.3.1 Parquet

Apache Parquet

Parquet est un format de fichier **colonnaire** (columnar) optimisé pour l'analyse de données. Contrairement au CSV (ligne par ligne), Parquet stocke les données **colonne par colonne**.

Avantages :

- **Compression** : jusqu'à 80% de réduction par rapport au CSV (les valeurs similaires d'une colonne se compressent très bien)
- **Performance** : lecture sélective des colonnes (on ne lit que les colonnes nécessaires à la requête)
- **Schéma intégré** : le schéma est stocké dans le fichier lui-même
- **Compatible** : Spark, Pandas, DuckDB, Hive, etc.

Dans notre projet : Les logs bruts sont archivés dans MinIO au format Parquet, partitionnés par `date/source_type/`.

5.3.2 Partitionnement

Partitionnement des données

Le **partitionnement** consiste à organiser les fichiers dans des sous-répertoires basés sur une ou plusieurs colonnes (ex : date, source_type). Cela permet de :

- **Réduire la quantité de données lues** : si on requête un seul jour, on ne lit que le dossier de ce jour
- **Paralléliser le traitement** : chaque partition peut être traitée indépendamment

Exemple de structure :

```
raw-logs/
 date=2026-01-15/
    source_type=firewall/
       part-00000.parquet
       part-00001.parquet
    source_type=webserver/
    source_type=ids/
 date=2026-01-16/
    ...
```

Chapitre 6

Transformation de Données

6.1 Définition

Transformation de Données

La **transformation** est le processus de modification des données brutes pour les rendre exploitables. Cela inclut :

- **Nettoyage** : supprimer les doublons, corriger les erreurs, gérer les valeurs manquantes
- **Normalisation** : convertir les données vers un format/schéma unifié
- **Enrichissement** : ajouter des informations supplémentaires (géolocalisation, métadonnées)
- **Agrégation** : calculer des métriques résumées (sommes, moyennes, comptages)
- **Filtrage** : retirer les données non pertinentes

6.2 Les 4 transformations de notre projet

6.2.1 T1 — Parsing & Normalisation vers ECS

Elastic Common Schema (ECS)

L'**ECS** est un **schéma standard** créé par Elastic pour normaliser les données de sécurité. Il définit des noms de champs communs pour que toutes les sources de logs utilisent le même vocabulaire.

Exemples de mapping :

- Le champ `src_ip` (firewall) → `source.ip` (ECS)
- Le champ `dst_ip` (firewall) → `destination.ip` (ECS)
- Le champ `timestamp` → `@timestamp` (ECS, format UTC)
- Le champ `user` (Windows) → `user.name` (ECS)

Pourquoi normaliser ? Sans normalisation, chaque source a ses propres noms de champs, ses propres formats de date, ses propres conventions. Il est impossible de corréler les événements entre sources sans un schéma commun.

6.2.2 T2 — Enrichissement GeoIP

L'enrichissement consiste à **ajouter des informations** qui n'existaient pas dans les données brutes. Dans notre cas :

- On prend une adresse IP source (ex : `185.220.101.42`)
- On la cherche dans la base **MaxMind GeoIP2** (base locale)
- On obtient : pays (`Germany`), ville (`Berlin`), latitude/longitude
- On ajoute ces champs à l'événement : `geo.country`, `geo.city`, `geo.location`

Cela permet de visualiser les attaques sur une **carte mondiale** dans Kibana.

6.2.3 T3 — Calcul de Score de Sévérité

On crée un score **unifié de 1 à 10** qui permet de comparer la gravité des événements entre différentes sources :

Source	Événement	Score	Logique
Firewall	Connexion bloquée	3	Action défensive normale
Firewall	Port scan détecté	7	Comportement suspect
IDS	Alerte critique	9	Menace active détectée
Web Server	Erreur 404	1	Normal
Web Server	Injection SQL	8	Attaque détectée

Windows	Échec de connexion	5 Potentiellement suspect

6.2.4 T4 — Déduplication

La **déduplication** supprime les événements identiques. On calcule un **hash MD5/SHA256** du contenu de l'événement + son timestamp. Si deux événements ont le même hash, on ne garde que le premier.

Pourquoi des doublons ?

Les doublons peuvent apparaître à cause de :

- **Retransmissions réseau** : un paquet perdu est renvoyé
- **Retry de Kafka** : si le producteur ne reçoit pas d'acquittement, il renvoie le message
- **Retraitement batch** : si un job batch est relancé, les mêmes données sont traitées deux fois

Chapitre 7

Orchestration

7.1 Définition

Orchestration

L'**orchestration** est le mécanisme qui **planifie**, **coordonne** et **supervise** l'exécution des différentes tâches d'un pipeline de données. C'est le « chef d'orchestre » qui décide :

- **Quand** chaque tâche doit s'exécuter (scheduling)
- **Dans quel ordre** (gestion des dépendances)
- **Que faire en cas d'échec** (retry, alertes)
- **Comment monitorer** l'état du pipeline (logs, métriques)

7.2 DAG — Directed Acyclic Graph

DAG

Un **DAG** (Graphe Orienté Acyclique) est la représentation d'un pipeline sous forme de graphe :

- Chaque **nœud** est une tâche (extraction, transformation, chargement, etc.)
- Chaque **arête** est une dépendance (la tâche B ne peut commencer qu'après la tâche A)
- **Acyclique** signifie qu'il n'y a pas de boucle (pas de dépendance circulaire)

Exemple de DAG dans notre projet :
`extract_logs` → `parse_logs` → `enrich_geoip` → `compute_severity` → `deduplicate` → `load_elasticsearch` → `quality_check`

7.3 Gestion des dépendances

Dans notre projet, les DAGs Airflow gèrent des dépendances complexes :

- Le DAG `dag_batch_reprocessing` **dépend** de `dag_batch_ingestion`

— Le DAG `dag_data_quality` **dépend** de `dag_batch_reprocessing`
— Au sein de chaque DAG : `parse » enrich » load » quality_check`

Chapitre 8

Qualité des Données (Data Quality)

8.1 Définition

Data Quality

La **qualité des données** mesure dans quelle mesure les données sont **fiables**, **complètes**, **cohérentes** et **précises** pour leur usage prévu. Un pipeline sans contrôle qualité peut produire des résultats incorrects — *garbage in, garbage out.*

8.2 Dimensions de la qualité

Dimension	Description	Exemple dans notre projet
Complétude	Pas de valeurs manquantes dans les champs obligatoires	Vérifier que `source.ip` n'est jamais NULL
Unicité	Pas de doublons	Déduplication par hash
Validité	Les données respectent un format attendu	Les IPs sont au format IPv4/IPv6 valide
Cohérence	Les données sont logiquement cohérentes	Le timestamp d'ingestion est toujours après le timestamp de l'événement
Fraîcheur	Les données sont récentes	Pas de logs de plus de 24h dans l'index du jour
Exactitude	Les données reflètent la réalité	Le GeoIP renvoie un pays valide pour chaque IP

8.3 Dead Letter Queue (DLQ)

Dead Letter Queue

Une **Dead Letter Queue** (file d'attente des messages morts) est une file séparée où sont envoyés les messages/événements qui n'ont pas pu être traités avec succès. Plutôt que de perdre ces données, on les stocke à part pour investigation ultérieure.

Dans notre projet : les événements qui échouent au parsing ou à la validation sont stockés dans l'index Elasticsearch `siem-errors-*` pour être analysés manuellement.

Chapitre 9

Robustesse et Tolérance aux Pannes

9.1 Retry (Mécanisme de Relance)

Retry avec Backoff Exponentiel

Le **retry** est le mécanisme qui relance automatiquement une tâche en échec. Le **backoff exponentiel** consiste à augmenter progressivement le délai entre chaque tentative :

- 1ère tentative : attendre 5 secondes
- 2ème tentative : attendre 10 secondes
- 3ème tentative : attendre 20 secondes
- (puis abandonner et loguer l'erreur)

Pourquoi le backoff ? Si le service distant est surchargé, le bombarder de requêtes immédiatement ne fera qu'aggraver le problème. Le backoff lui laisse le temps de récupérer.

9.2 Idempotence

Idempotence

Une opération est **idempotente** si on peut l'exécuter plusieurs fois sans changer le résultat final. C'est essentiel quand on a des mécanismes de retry.

Exemple : Si le chargement dans Elasticsearch échoue et est relancé, les mêmes documents seront réinsérés. Grâce à un **ID unique** par document (basé sur le hash du contenu), Elasticsearch met à jour plutôt que de dupliquer.

9.3 Logging Structuré

Logs Structurés

Les **logs structurés** sont des messages de log au format **JSON** (plutôt que du texte libre). Chaque log contient des champs standards :

```
{
    "timestamp": "2026-01-15T14:32:01Z",
    "level": "ERROR",
    "component": "spark_streaming",
    "message": "Failed to index event in Elasticsearch",
    "correlation_id": "abc-123-def",
    "retry_count": 2,
    "error": "ConnectionTimeout"
}
```

Avantages : On peut rechercher, filtrer et agréger les logs facilement. Avec du texte libre, il faudrait parser chaque ligne avec des regex.

Dans notre projet : Nous utilisons la bibliothèque Python **structlog** pour générer des logs JSON structurés dans tous les composants.

Chapitre 10

Concepts Avancés

10.1 Exactly-Once vs At-Least-Once vs At-Most-Once

Garanties de livraison

Dans un système distribué, il existe 3 niveaux de garantie pour la livraison des messages :

- **At-Most-Once** : Le message est envoyé une fois. S'il est perdu, il n'est pas renvoyé. *Risque : perte de données.*
- **At-Least-Once** : Le message est envoyé jusqu'à confirmation de réception. S'il y a un doute, il est renvoyé. *Risque : doublons.*
- **Exactly-Once** : Le message est garanti d'être traité **exactement une fois**. *Le plus complexe à implémenter.*

Notre choix : **At-Least-Once** (Kafka par défaut) + **Déduplication** en aval (T4) pour simuler un comportement Exactly-Once.

10.2 Backpressure

Backpressure

Le **backpressure** (contre-pression) est un mécanisme qui ralentit le producteur lorsque le consommateur ne peut pas suivre le rythme de traitement. Sans backpressure, le consommateur serait submergé et crasherait.

Dans Kafka : si Spark Streaming ne consomme pas assez vite, les messages restent dans Kafka (qui sert de buffer). Kafka ne perd pas les messages et Spark les consomme à son rythme.

Dans Spark Structured Streaming : le paramètre `maxOffsetsPerTrigger` limite le nombre de messages lus à chaque micro-batch, évitant la surcharge.

10.3 Serialization / Deserialization (SerDe)

Sérialisation

La **sérialisation** est le processus de conversion d'un objet en mémoire vers un format stockable ou transmissible (bytes). La **désérialisation** est l'opération inverse.

Formats courants :

- **JSON** : lisible par l'humain, mais volumineux et lent à parser
- **Avro** : format binaire avec schéma, compact et rapide (populaire avec Kafka)
- **Protobuf** : format binaire de Google, très performant
- **Parquet** : format colonnaire pour le stockage analytique

Dans notre projet : Les messages Kafka sont en JSON (simplicité). Les archives MinIO sont en Parquet (performance analytique).

Deuxième partie

Les Outils en Détail

Chapitre 11

Apache Kafka

11.1 Qu'est-ce que Kafka ?

Apache Kafka

Apache Kafka est une plateforme de **streaming d'événements distribués** (distributed event streaming platform). Créé par LinkedIn en 2011 et donné à la fondation Apache, Kafka est devenu le **standard de facto** pour le transport de données en temps réel.

En termes simples, Kafka est un **système de messagerie** ultra-performant qui permet à des applications de publier et consommer des flux de données en temps réel, avec une garantie de **haute disponibilité**, de **durabilité** et de **scalabilité**.

11.2 Architecture de Kafka

11.2.1 Concepts clés

Concept	Description
Broker	Un serveur Kafka. Un cluster Kafka est composé de plusieurs brokers pour la haute disponibilité. Chaque broker stocke une partie des données.
Topic	Un « canal » ou « catégorie » de messages. Les producteurs publient dans un topic, les consommateurs lisent depuis un topic. Analogie : une chaîne de télévision.
Partition	Chaque topic est divisé en partitions. Les partitions permettent le parallélisme : chaque partition peut être lue par un consommateur différent.
Producer	L'application qui envoie des messages dans un topic Kafka.
Consumer	L'application qui lit des messages depuis un topic Kafka.

Consumer Group	Un ensemble de consommateurs qui se répartissent la lecture des partitions d'un topic. Chaque partition est lue par un seul consommateur du groupe.
Offset	La position d'un message dans une partition. Chaque message a un offset unique et croissant. Le consommateur suit son offset pour savoir où il en est.
Replication Factor	Le nombre de copies de chaque partition. Avec un facteur de 3, chaque partition existe sur 3 brokers différents — si un broker tombe, les données ne sont pas perdues.

11.2.2 Flux de données dans Kafka

```
Producteur                    Kafka Cluster                    Consommateur

    publish("firewall-logs",  Topic: firewall-logs
        {src_ip: "1.2.3.4"})      Partition 0
                                   Partition 1
                                   Partition 2

                                                      consume
                                                        (Spark)
```

11.3 Kafka dans notre projet

Configuration Kafka dans le projet SIEM

4 topics sont créés, un par type de source :

- `firewall-logs` — logs de pare-feu au format Syslog/CEF
- `web-logs` — logs de serveur web au format Apache/Nginx
- `windows-logs` — événements Windows au format JSON
- `ids-logs` — alertes IDS Suricata au format Eve JSON

Producteurs : les scripts Python dans `generators/` publient des événements simulés.

Consommateur : Spark Structured Streaming lit depuis ces 4 topics en parallèle.

Pourquoi Kafka plutôt que RabbitMQ ?

- Kafka est conçu pour le **haut débit** (millions de messages/sec)
- Kafka **persiste** les messages sur disque (RabbitMQ les supprime après consommation)
- Kafka permet de **rejouer** les messages (fondamental pour le retraitement batch)
- Kafka est le **standard** en Data Engineering

11.4 Garanties de Kafka

— **Durabilité** : les messages sont écrits sur disque et répliqués

— **Ordre** : les messages sont ordonnés **au sein d'une partition** (pas entre partitions)

— **Rétention** : les messages sont conservés pendant une durée configurable (ex : 7 jours), même après consommation

— **Scalabilité horizontale** : ajouter des brokers et des partitions pour augmenter le débit

Chapitre 12

Apache Spark

12.1 Qu'est-ce que Spark ?

Apache Spark

Apache Spark est un moteur de traitement de données distribué, conçu pour le traitement **massif** (Big Data) en mémoire. Créé à l'UC Berkeley en 2009, Spark est jusqu'à **100x plus rapide** que Hadoop MapReduce grâce au traitement en mémoire (in-memory computing).

Spark supporte :

- **Batch processing** : traitement de gros volumes de données stockés
- **Streaming** : traitement de flux de données en temps réel (Structured Streaming)
- **SQL** : requêtes SQL sur des données structurées (Spark SQL)
- **ML** : machine learning distribué (MLlib)
- **Graphes** : traitement de graphes (GraphX)

12.2 Architecture de Spark

12.2.1 Concepts clés

Concept	Description
Driver	Le programme principal qui orchestre le traitement. Il crée le SparkContext, définit les transformations et actions, et distribue le travail.
Executor	Un processus sur un nœud du cluster qui exécute les tâches assignées par le Driver. Chaque executor a sa propre mémoire et ses propres CPU.
Cluster Manager	Le gestionnaire de ressources (YARN, Mesos, Kubernetes, ou Standalone) qui alloue les executors.

RDD	*Resilient Distributed Dataset* — la structure de données fondamentale de Spark. Collection distribuée et immuable d'éléments.
DataFrame	Structure tabulaire (comme une table SQL) distribuée. Plus performant que les RDD grâce à l'optimiseur Catalyst. **C'est ce que nous utilisons.**
Transformation	Opération paresseuse (*lazy*) qui définit comment transformer un DataFrame (filter, map, groupBy, etc.). N'est exécutée que quand une action est appelée.
Action	Opération qui déclenche l'exécution réelle (count, show, write, collect).

12.2.2 Lazy Evaluation

Évaluation Paresseuse (Lazy Evaluation)

Spark utilise l'**évaluation paresseuse** : les transformations ne sont pas exécutées immédiatement. Spark construit un **plan d'exécution** (DAG de transformations) et ne l'exécute que lorsqu'une **action** est appelée.

Avantage : Spark peut optimiser le plan d'exécution global avant de l'exécuter. Par exemple, si vous filtrez puis projetez, Spark peut réorganiser les opérations pour lire moins de données.

12.3 Spark Structured Streaming

Structured Streaming

Structured Streaming est le moteur de streaming de Spark. Il traite les flux de données en temps réel en utilisant le même modèle de programmation que le batch (DataFrames et SQL).

Principe : le flux de données est traité comme une **table infinie** qui grandit continuellement. Chaque nouveau lot de données est un ensemble de nouvelles lignes ajoutées à cette table.

Modes de sortie :

- **Append** : seules les nouvelles lignes sont écrites (notre choix pour Elasticsearch)
- **Complete** : toute la table est réécrite à chaque itération (utile pour les agrégations)
- **Update** : seules les lignes modifiées sont écrites

12.4 PySpark

PySpark

PySpark est l'API Python de Spark. Elle permet d'écrire des jobs Spark en Python plutôt qu'en Scala ou Java. PySpark est devenu le standard en Data Engineering car la majorité des data engineers utilisent Python.

Dans notre projet, tous les jobs Spark sont écrits en PySpark :

- `streaming_job.py` : lit depuis Kafka, applique les transformations, écrit dans ES et MinIO
- `batch_job.py` : retraite les données historiques depuis MinIO
- `parsers/` : fonctions de parsing pour chaque format de log

12.5 Spark dans notre projet

Listing 12.1 – Exemple simplifié du streaming job

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import from_json, col

spark = SparkSession.builder \
    .appName("SIEM-Streaming") \
    .getOrCreate()

# Lire depuis Kafka
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "firewall-logs") \
    .load()

# Parser les messages JSON
parsed = df.select(
    from_json(col("value").cast("string"), schema).alias("data")
).select("data.*")

# Appliquer les transformations
normalized = parse_firewall(parsed)        # T1
enriched = enrich_geoip(normalized)        # T2
scored = compute_severity(enriched)        # T3
deduped = deduplicate(scored)              # T4

# Ecrire dans Elasticsearch
deduped.writeStream \
    .format("es") \
    .option("es.resource", "siem-firewall-{@timestamp}") \
    .start()
```

Chapitre 13

Apache Airflow

13.1 Qu'est-ce qu'Airflow ?

Apache Airflow

Apache Airflow est une plateforme d'**orchestration de workflows** (flux de travail). Créé par Airbnb en 2014, Airflow permet de définir, planifier et surveiller des pipelines de données complexes via du **code Python** (et non via une interface graphique).

Philosophie : *« Workflows as Code »* — les pipelines sont définis en Python, versionnés dans Git, testables et reproductibles.

Airflow est le **standard de l'industrie** pour l'orchestration de pipelines Data Engineering.

13.2 Architecture d'Airflow

Composant	Description
Scheduler	Le planificateur qui décide quand exécuter les DAGs et les tâches. Il surveille les horaires et les dépendances.
Executor	Le moteur qui exécute les tâches. Types : LocalExecutor (un seul serveur), CeleryExecutor (distribué), KubernetesExecutor.
Webserver	L'interface web (UI) pour visualiser les DAGs, surveiller l'état des tâches, relancer des jobs, consulter les logs.
Metadata DB	Base de données (PostgreSQL ou MySQL) qui stocke l'état des DAGs, l'historique des exécutions, les variables, les connexions.
Worker	Le processus qui exécute réellement les tâches (quand on utilise CeleryExecutor ou KubernetesExecutor).

13.3 Concepts clés

13.3.1 DAG (Directed Acyclic Graph)

Un DAG Airflow est un fichier Python qui définit :

— Les **tâches** (tasks) à exécuter
— Les **dépendances** entre tâches
— Le **schedule** (fréquence d'exécution)
— Les **paramètres** de retry, timeout, etc.

13.3.2 Operators

Les **operators** sont les types de tâches disponibles :

— **PythonOperator** : exécute une fonction Python
— **BashOperator** : exécute une commande shell
— **SparkSubmitOperator** : soumet un job Spark
— **Sensor** : attend qu'une condition soit remplie (ex : un fichier existe)

13.3.3 Exemple de DAG

Listing 13.1 – Exemple de DAG Airflow

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime, timedelta

default_args = {
    'retries': 3,                          # Retry 3 fois en cas d'echec
    'retry_delay': timedelta(minutes=2),   # Attendre 2 min entre chaque retry
}

with DAG(
    'dag_batch_ingestion',
    schedule_interval='@hourly',           # Execution toutes les heures
    start_date=datetime(2026, 1, 1),
    default_args=default_args,
    catchup=False,
) as dag:

    extract = PythonOperator(
        task_id='extract_log_files',
        python_callable=extract_logs_from_directory,
    )

    publish = PythonOperator(
```

```
        task_id='publish_to_kafka',
        python_callable=publish_logs_to_kafka,
    )

    validate = PythonOperator(
        task_id='validate_ingestion',
        python_callable=validate_kafka_messages,
    )

    # Dependances
    extract >> publish >> validate
```

13.4 DAGs dans notre projet

DAG	Fréquence	Description
`dag_batch_ingestion`	Horaire	Récupère les fichiers de logs accumulés et les publie dans Kafka
`dag_batch_reprocessing`	Quotidien (02h)	Relance Spark batch pour corriger/compléter les données de la veille
`dag_data_quality`	Quotidien (04h)	Exécute les checks Great Expectations et génère un rapport qualité
`dag_retention_cleanup`	Hebdomadaire	Supprime les anciens index ES (>30j) et archive les données MinIO (>1an)

Chapitre 14

Elasticsearch

14.1 Qu'est-ce qu'Elasticsearch ?

Elasticsearch

Elasticsearch est un moteur de recherche et d'analyse distribué, open-source, basé sur la bibliothèque **Apache Lucene**. Il est conçu pour :

- La **recherche full-text** ultra-rapide (recherche dans des millions de documents en millisecondes)
- L'**analyse de logs** et de données temporelles
- L'**agrégation** en temps réel (comptages, moyennes, histogrammes)

Elasticsearch est le **cœur de la Elastic Stack** (anciennement ELK Stack) et le **standard** pour les solutions SIEM.

14.2 Concepts clés

Concept	Description
Index	Équivalent d'une « table » en base de données relationnelle. Un index est une collection de documents ayant une structure similaire. Ex : `siem-firewall-2026.01.15`
Document	Équivalent d'une « ligne » en base relationnelle. Un document est un objet JSON stocké dans un index. Chaque événement de log est un document.
Mapping	Équivalent du « schéma ». Définit les types de champs (text, keyword, date, ip, geo_point, etc.) pour un index.
Shard	Un index est divisé en shards (fragments) distribués sur les nœuds du cluster. Permet la scalabilité et le parallélisme.

Replica	Copie d'un shard sur un autre nœud. Assure la haute disponibilité (si un nœud tombe, les replicas prennent le relais).
ILM (Index Lifecycle Management)	Politique de gestion du cycle de vie des index : hot → warm → cold → delete. Permet de gérer la rétention automatiquement.

14.3 ECS — Elastic Common Schema

Elastic Common Schema (ECS)

L'**ECS** est un **standard de nommage** des champs créé par Elastic. Il définit un vocabulaire commun pour normaliser les données de sources différentes.

Catégories principales d'ECS :

- `@timestamp` — horodatage de l'événement
- `source.*` — informations sur la source (IP, port, geo)
- `destination.*` — informations sur la destination
- `event.*` — type d'événement, action, sévérité
- `user.*` — informations utilisateur
- `host.*` — machine hôte
- `network.*` — protocole, bytes transférés

Avantage : Permet de corréler des événements entre sources différentes. Si tous les logs ont `source.ip`, on peut facilement chercher « toutes les activités de l'IP 1.2.3.4 » à travers firewalls, serveurs web et IDS.

14.4 ILM — Index Lifecycle Management

ILM

ILM permet de gérer automatiquement le cycle de vie des index Elasticsearch :

1. **Hot Phase** : Index actif, recevant des écritures. Stocké sur des disques SSD rapides.
2. **Warm Phase** : Index en lecture seule, encore requêtable. Peut être déplacé vers des disques moins coûteux.
3. **Cold Phase** : Index rarement consulté, fortement compressé.
4. **Delete Phase** : Index supprimé après la période de rétention.

Dans notre projet : Les index SIEM sont en phase Hot pendant 7 jours, puis passent en Warm pendant 23 jours, puis sont supprimés (rétention totale : 30 jours).

Chapitre 15

Kibana

15.1 Qu'est-ce que Kibana ?

Kibana

Kibana est la plateforme de **visualisation** de la Elastic Stack. C'est l'interface web qui permet de :

- **Explorer** les données d'Elasticsearch (onglet Discover)
- **Créer des dashboards** interactifs avec graphiques, tableaux, cartes
- **Analyser les logs** en temps réel
- **Configurer des alertes** basées sur des conditions
- **Utiliser les fonctionnalités SIEM** intégrées (Security app)

Kibana est couplé **nativement** à Elasticsearch — il n'y a pas de configuration de connexion complexe.

15.2 Dashboards SIEM dans notre projet

Notre dashboard Kibana comprend 6 visualisations :

1. **Top 10 Source IPs** — Diagramme en barres des IPs les plus actives (permet d'identifier les attaquants les plus persistants)
2. **Timeline des événements** — Histogramme temporel par type de source (montre les pics d'activité)
3. **Alertes IDS par sévérité** — Pie chart des alertes Suricata (priorisation des menaces)
4. **Carte de géolocalisation** — Carte mondiale des IPs sources (visualisation géographique des attaques)
5. **Statistiques par type de log** — Compteurs et tendances par source
6. **Top actions bloquées** — Tableau des règles firewall les plus déclenchées

Chapitre 16

MinIO (Data Lake)

16.1 Qu'est-ce que MinIO ?

MinIO

MinIO est un serveur de **stockage objet** haute performance, **compatible avec l'API Amazon S3**. C'est une solution open-source qui permet de créer un **Data Lake privé** sans dépendre d'un fournisseur cloud.

Stockage objet vs stockage fichier :

- **Stockage fichier** (NFS, HDFS) : les fichiers sont organisés en répertoires hiérarchiques
- **Stockage objet** (S3, MinIO) : les « objets » (fichiers) sont stockés dans des « buckets » (conteneurs) avec des métadonnées. Pas de hiérarchie réelle, mais on simule des dossiers avec des préfixes dans les noms de clés.

Pourquoi MinIO plutôt que S3 ?

- Déployable **en local** via Docker (pas de compte AWS nécessaire)
- **Gratuit** et open-source
- **100% compatible S3** : le même code fonctionne avec MinIO et avec AWS S3
- Parfait pour le **développement local** et les projets académiques

16.2 MinIO dans notre projet

Utilisation de MinIO

Bucket : `raw-logs`

Organisation :

```
raw-logs/
 firewall/
    date=2026-01-15/
       part-00000.parquet
       part-00001.parquet
    date=2026-01-16/
 webserver/
 windows/
 ids/
```

Format : Apache Parquet (colonnaire, compressé)

Rétention : 1 an (les fichiers de plus d'un an sont supprimés par le DAG `dag_retention_cleanup`)

Rôle : Archivage des logs bruts pour le retraitement batch et l'analyse historique.

Chapitre 17

Grafana

17.1 Qu'est-ce que Grafana ?

Grafana

Grafana est une plateforme open-source de **monitoring et visualisation**. Elle permet de créer des dashboards temps réel connectés à de multiples sources de données (Prometheus, Elasticsearch, InfluxDB, PostgreSQL, etc.).

Différence avec Kibana :

- **Kibana** est spécialisé pour Elasticsearch et l'analyse de logs/SIEM
- **Grafana** est généraliste et multi-sources, idéal pour le monitoring d'infrastructure

Dans notre projet, Grafana sert à monitorer **le pipeline lui-même** (pas les logs de sécurité).

17.2 Métriques monitorées

Métrique	Source	Utilité
Throughput Kafka	Kafka JMX / Prometheus	Messages/sec par topic — détecte les ralentissements
Latence Spark	Spark UI / Prometheus	Temps de traitement par micro-batch — détecte les goulots
Taux de parsing	Application logs	% d'événements parsés avec succès — détecte les problèmes de format
Taux d'erreurs	Application logs	% d'échecs par composant — alerte sur les pannes
Lag Kafka	Kafka Consumer metrics	Retard du consommateur — détecte si Spark ne suit plus

Chapitre 18

Docker & Docker Compose

18.1 Qu'est-ce que Docker ?

Docker

Docker est une plateforme de **conteneurisation** qui permet d'empaqueter une application avec toutes ses dépendances (librairies, runtime, configuration) dans un **conteneur** isolé et portable.

Analogie : Un conteneur Docker est comme un **colis** qui contient tout ce dont l'application a besoin pour fonctionner. Peu importe le système hôte, le colis fonctionne toujours de la même manière.

Conteneur vs Machine Virtuelle :

- **VM** : virtualise le hardware complet + OS complet. Lourd (Go de RAM), démarrage lent (minutes).
- **Conteneur** : partage le noyau de l'OS hôte, isole uniquement l'application. Léger (Mo de RAM), démarrage instantané (secondes).

18.2 Concepts clés

Concept	Description
Image	Un « template » en lecture seule qui contient l'OS, les dépendances et l'application. Ex : `elasticsearch:8.12.0`
Conteneur	Une instance en cours d'exécution d'une image. Comme un processus isolé sur l'hôte.
Dockerfile	Le fichier de recette qui décrit comment construire une image (base image, commandes, ports, etc.)
Volume	Un mécanisme de persistance des données. Sans volume, les données du conteneur sont perdues quand il s'arrête.
Network	Un réseau virtuel qui permet aux conteneurs de communiquer entre eux par nom de service.

Registry	Un dépôt d'images (Docker Hub est le registry public par défaut).

18.3 Docker Compose

Docker Compose

Docker Compose est un outil qui permet de définir et lancer une application **multi-conteneurs** à partir d'un seul fichier `docker-compose.yml`. Il orchestre le démarrage de tous les services, leurs réseaux, leurs volumes et leurs dépendances.

Commande magique :

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

Cette commande lance **tous les services** du projet en arrière-plan : Kafka, Elasticsearch, Kibana, MinIO, Airflow, Grafana, etc.

18.4 Services Docker dans notre projet

Service	Image	Port	Rôle
Zookeeper	confluentinc/cp-zookeeper	2181	Coordination du cluster Kafka
Kafka	confluentinc/cp-kafka	9092	Broker de messages
Elasticsearch	elasticsearch :8.x	9200	Indexation des logs
Kibana	kibana :8.x	5601	Dashboards SIEM
MinIO	minio/minio	9000/9001	Data Lake (stockage objet)
Airflow	apache/airflow :2.x	8080	Orchestration
Spark	bitnami/spark	7077	Traitement des données
Grafana	grafana/grafana	3000	Monitoring
Kafka UI	provectuslabs/kafka-ui	8081	Interface web Kafka

Chapitre 19

Python et ses Bibliothèques

19.1 Bibliothèques utilisées

Bibliothèque	Rôle dans le projet
kafka-python	Client Python pour Kafka. Permet de publier des messages (KafkaProducer) et de les consommer (KafkaConsumer).
pyspark	API Python de Apache Spark. Permet d'écrire des jobs de traitement distribué.
elasticsearch-py	Client Python officiel pour Elasticsearch. Permet d'indexer et de rechercher des documents.
geoip2	Client Python pour la base MaxMind GeoIP2. Permet de géolocaliser les adresses IP.
structlog	Bibliothèque de logging structuré. Génère des logs au format JSON avec des champs standardisés.
boto3	SDK AWS pour Python. Utilisé pour interagir avec MinIO (compatible S3).
pytest	Framework de tests unitaires. Permet d'écrire et d'exécuter des tests automatisés.
great-expectations	Framework de validation de la qualité des données. Définit des « expectations » (attentes) sur les données.
faker	Génération de données réalistes (IPs, noms d'utilisateurs, timestamps aléatoires).
apache-airflow	Framework d'orchestration. Les DAGs sont des scripts Python importés par Airflow.

Chapitre 20

pytest — Framework de Tests

20.1 Qu'est-ce que pytest ?

pytest

pytest est le framework de tests unitaires le plus populaire en Python. Il permet d'écrire des tests simples et lisibles pour valider le comportement des fonctions.

Convention : Les fichiers de tests commencent par `test_` et les fonctions de tests commencent aussi par `test_`.

20.2 Exemple de test dans notre projet

Listing 20.1 – Test du parser Firewall

```
import pytest
from spark_jobs.parsers.firewall_parser import parse_firewall_log

def test_parse_firewall_valid_cef():
    """Test parsing d'un log CEF valide"""
    raw_log = 'CEF:0|Fortinet|FortiGate|6.4|traffic|allowed|3|src
       =192.168.1.100 dst=10.0.0.1 spt=54321 dpt=443 act=allow'

    result = parse_firewall_log(raw_log)

    assert result['source.ip'] == '192.168.1.100'
    assert result['destination.ip'] == '10.0.0.1'
    assert result['destination.port'] == 443
    assert result['event.action'] == 'allow'

def test_parse_firewall_invalid_format():
    """Test parsing d'un log avec format invalide"""
    raw_log = 'INVALID LOG FORMAT'

    result = parse_firewall_log(raw_log)

```

```
21      assert result is None  # Le parser retourne None pour les logs
            invalides
```

Chapitre 21

Great Expectations — Tests de Qualité

21.1 Qu'est-ce que Great Expectations ?

Great Expectations

Great Expectations est un framework Python de **validation de la qualité des données**. Il permet de définir des « expectations » (attentes) sur les données :

- Cette colonne ne doit **jamais** être NULL
- Les valeurs de cette colonne doivent être dans un **ensemble donné**
- Cette colonne doit contenir des **IPs valides**
- Le nombre de lignes doit être **supérieur à** 1000

Great Expectations génère des **rapports HTML** détaillant quelles expectations ont réussi ou échoué.

21.2 Exemple dans notre projet

Listing 21.1 – Test de qualité des données

```
import great_expectations as gx

def test_mandatory_fields_not_null():
    """Verifie que les champs obligatoires ne sont jamais NULL"""
    context = gx.get_context()

    validator = context.sources.pandas_default.read_csv(
        "output/normalized_logs.csv"
    )

    # Le champ source.ip ne doit jamais etre null
    result = validator.expect_column_values_to_not_be_null("source.
       ip")
    assert result.success

    # Le champ @timestamp ne doit jamais etre null
```

```
    result = validator.expect_column_values_to_not_be_null("
        @timestamp")
    assert result.success

def test_valid_ip_format():
    """Verifie que les IPs sont au format valide"""
    result = validator.expect_column_values_to_match_regex(
        "source.ip",
        r"^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$"
    )
    assert result.success
```

Chapitre 22

structlog — Logging Structuré

22.1 Qu'est-ce que structlog ?

structlog

structlog est une bibliothèque Python qui remplace le module `logging` standard par un système de **logs structurés**. Au lieu de produire des lignes de texte libre, structlog produit des **objets JSON** avec des champs standards.

Avant (texte libre) :

```
2026-01-15 14:32:01 ERROR Failed to index event in ES
```

Après (structuré avec structlog) :

```
{"timestamp": "2026-01-15T14:32:01Z", "level": "error",
 "component": "elasticsearch_loader",
 "message": "Failed to index event",
 "retry_count": 2, "error": "ConnectionTimeout",
 "correlation_id": "abc-123"}
```

Avantage : Les logs structurés sont **searchable** (on peut filtrer par `component`, `level`, etc.) et intégrables dans Elasticsearch pour le monitoring.

Troisième partie

Concepts Transversaux

Chapitre 23

Le SIEM et la Cybersécurité

23.1 Qu'est-ce qu'un SIEM ?

SIEM — Security Information and Event Management

Un **SIEM** est un système qui collecte, normalise, corrèle et analyse les événements de sécurité provenant de toute l'infrastructure informatique d'une organisation. Son objectif est de :

- **Détecter** les incidents de sécurité (intrusions, malware, fuites de données)
- **Alerter** les analystes SOC en temps réel
- **Investiguer** les incidents grâce à l'historique des événements
- **Se conformer** aux réglementations (RGPD, PCI-DSS, SOX)

Solutions SIEM du marché : Splunk, IBM QRadar, Microsoft Sentinel, Elastic SIEM (notre choix).

23.2 Types de logs de sécurité

23.2.1 Logs Firewall (Syslog/CEF)

Format CEF — Common Event Format

Le **CEF** est un format standard pour les logs de sécurité, créé par ArcSight. Structure :

```
CEF:Version|Device Vendor|Device Product|Device Version|
    Signature ID|Name|Severity|Extension
```

Exemple :

```
CEF:0|Fortinet|FortiGate|6.4|traffic|allowed|3|
    src=192.168.1.100 dst=10.0.0.1 spt=54321
    dpt=443 act=allow proto=TCP
```

23.2.2 Logs Web Server (Apache/Nginx)

Format **Combined Log Format** :

```
192.168.1.100 - admin [15/Jan/2026:14:32:01 +0000]
    "GET /admin/login HTTP/1.1" 200 4523
    "https://example.com" "Mozilla/5.0"
```

23.2.3 Windows Events (JSON)

```
{
    "EventID": 4625,
    "TimeCreated": "2026-01-15T14:32:01Z",
    "Computer": "DC01.corp.local",
    "TargetUserName": "admin",
    "LogonType": 3,
    "IpAddress": "192.168.1.100",
    "Status": "0xC000006D"
}
```

L'EventID 4625 signifie « échec de connexion » — un indicateur potentiel de brute force.

23.2.4 Logs IDS Suricata (Eve JSON)

```
{
    "timestamp": "2026-01-15T14:32:01Z",
    "event_type": "alert",
    "src_ip": "185.220.101.42",
    "dest_ip": "192.168.1.10",
    "alert": {
        "signature": "ET SCAN Nmap SYN Scan",
        "severity": 2,
        "category": "Attempted Information Leak"
    }
}
```

Chapitre 24

Scalabilité et Performance

24.1 Scalabilité Horizontale vs Verticale

Scalabilité

— **Scalabilité verticale** (Scale Up) : Ajouter plus de ressources à une seule machine (plus de RAM, CPU, disque). Limité par la capacité maximale d'une machine.

— **Scalabilité horizontale** (Scale Out) : Ajouter plus de machines au cluster. Théoriquement illimitée.

Notre stack est conçu pour la scalabilité horizontale :

— **Kafka** : ajouter des brokers et des partitions

— **Spark** : ajouter des workers/executors

— **Elasticsearch** : ajouter des nœuds et des shards

— **MinIO** : ajouter des serveurs de stockage

24.2 Latence vs Throughput

Latence	**Throughput (Débit)**
Temps pour traiter **un seul** événement	Nombre d'événements traités **par seconde**
Notre objectif : < 5 secondes	Notre objectif : 100K+ events/jour
Mesuré en millisecondes/secondes	Mesuré en events/sec ou records/sec
Important pour les alertes temps réel	Important pour le traitement batch

Chapitre 25

Containerisation et Reproductibilité

25.1 Pourquoi containeriser un pipeline ?

Infrastructure as Code

La containerisation via Docker Compose permet de :

1. **Reproductibilité** : tout le monde obtient le même environnement avec `docker-compose up`
2. **Isolation** : chaque service a ses propres dépendances, pas de conflits
3. **Portabilité** : fonctionne sur Linux, macOS, Windows
4. **Versionnement** : le `docker-compose.yml` est versionné dans Git
5. **Facilité de déploiement** : un seul fichier pour démarrer 9+ services

C'est le principe de l'**Infrastructure as Code** (IaC) : l'infrastructure est décrite dans des fichiers de configuration versionnés.

Chapitre 26

Récapitulatif : Comment Tout S'Assemble

26.1 Le flux complet d'un événement

Suivons le parcours d'un **log firewall** depuis sa génération jusqu'à sa visualisation :

Étape 1 : **Génération** — Le script `firewall_generator.py` crée un log CEF simulé avec `faker`

Étape 2 : **Publication Kafka** — Le log est sérialisé en JSON et publié dans le topic `firewall-logs` via `kafka-python`

Étape 3 : **Consommation Spark** — Spark Structured Streaming lit le message depuis Kafka en quelques millisecondes

Étape 4 : **T1 - Parsing** — `firewall_parser.py` extrait les champs CEF et les mappe vers le schéma ECS

Étape 5 : **T2 - Enrichissement** — `geoip_enricher.py` géolocalise l'IP source : `185.220.101.42` → Allemagne, Berlin

Étape 6 : **T3 - Scoring** — `severity_scorer.py` calcule un score de 7/10 (port scan détecté)

Étape 7 : **T4 - Déduplication** — `deduplicator.py` vérifie que cet événement n'a pas déjà été traité

Étape 8 : **Archivage MinIO** — Le log brut est écrit en Parquet dans `raw-logs/firewall/date=2026-01-1`

Étape 9 : **Indexation Elasticsearch** — Le log normalisé est indexé dans `siem-firewall-2026.01.15`

Étape 10 : **Visualisation Kibana** — Le point apparaît sur la carte mondiale, l'IP monte dans le top 10, l'histogramme temporel se met à jour

Étape 11 : **Monitoring Grafana** — Le compteur de throughput Kafka s'incrémente, la latence end-to-end est mesurée

Temps total : < 5 secondes de bout en bout.

Annexe A

Glossaire

Terme	Définition
API	Application Programming Interface — interface pour communiquer entre applications
Batch	Traitement par lots de données accumulées
Broker	Serveur intermédiaire (Kafka broker = serveur Kafka)
CDC	Change Data Capture — capture des changements en BDD
CEF	Common Event Format — format standard de logs de sécurité
Cluster	Ensemble de machines travaillant ensemble
Consumer	Application qui lit des messages depuis Kafka
DAG	Directed Acyclic Graph — graphe orienté sans cycle
Data Lake	Stockage centralisé de données brutes
Data Warehouse	Stockage optimisé pour l'analyse (OLAP)
DLQ	Dead Letter Queue — file pour messages en erreur
ECS	Elastic Common Schema — schéma standard Elastic
ELT	Extract-Load-Transform
ETL	Extract-Transform-Load
GeoIP	Géolocalisation par adresse IP
Idempotent	Opération donnant le même résultat si exécutée plusieurs fois
IDS	Intrusion Detection System
ILM	Index Lifecycle Management
Offset	Position d'un message dans une partition Kafka
OLAP	Online Analytical Processing
Parquet	Format de fichier colonnaire pour l'analytique
Partition	Subdivision d'un topic Kafka pour le parallélisme
Producer	Application qui envoie des messages dans Kafka
Schema	Structure définissant les champs et types d'un dataset
SerDe	Serialization/Deserialization

SIEM	Security Information and Event Management
SOC	Security Operations Center
Streaming	Traitement de données en temps réel
Topic	Canal/catégorie de messages dans Kafka

Annexe B

Commandes Utiles

B.1 Docker

```
# Lancer tous les services
docker-compose -f docker/docker-compose.yml up -d

# Voir les logs d'un service
docker-compose logs -f kafka

# Arreter tous les services
docker-compose down

# Arreter et supprimer les volumes (reset complet)
docker-compose down -v
```

B.2 Kafka

```
# Lister les topics
kafka-topics --bootstrap-server localhost:9092 --list

# Consommer les messages d'un topic
kafka-console-consumer --bootstrap-server localhost:9092 \
    --topic firewall-logs --from-beginning

# Voir les consumer groups
kafka-consumer-groups --bootstrap-server localhost:9092 --list
```

B.3 Elasticsearch

```
# Verifier la sante du cluster
curl -X GET "localhost:9200/_cluster/health?pretty"

# Lister les index
curl -X GET "localhost:9200/_cat/indices?v"
```

```
# Rechercher dans un index
curl -X GET "localhost:9200/siem-firewall-*/_search?pretty" \
    -H 'Content-Type: application/json' \
    -d '{"query": {"match": {"source.ip": "185.220.101.42"}}}'
```

B.4 Spark

```
# Soumettre un job streaming
spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2
   .12:3.5.0 \
    spark_jobs/streaming_job.py

# Soumettre un job batch
spark-submit spark_jobs/batch_job.py
```

B.5 Tests

```
# Executer tous les tests
pytest tests/ -v

# Executer avec couverture de code
pytest tests/ -v --cov=spark_jobs --cov-report=html

# Executer un test specifique
pytest tests/test_firewall_parser.py -v
```

Annexe C

Références et Ressources

1. **Apache Kafka Documentation** — https://kafka.apache.org/documentation/
2. **Apache Spark Documentation** — https://spark.apache.org/docs/latest/
3. **Apache Airflow Documentation** — https://airflow.apache.org/docs/
4. **Elasticsearch Guide** — https://www.elastic.co/guide/en/elasticsearch/reference/current/
5. **Elastic Common Schema (ECS)** — https://www.elastic.co/guide/en/ecs/current/
6. **MinIO Documentation** — https://min.io/docs/minio/linux/
7. **Kibana Guide** — https://www.elastic.co/guide/en/kibana/current/
8. **Grafana Documentation** — https://grafana.com/docs/grafana/latest/
9. **Docker Documentation** — https://docs.docker.com/
10. **pytest Documentation** — https://docs.pytest.org/
11. **Great Expectations** — https://greatexpectations.io/
12. **structlog** — https://www.structlog.org/
13. **Designing Data-Intensive Applications** — Martin Kleppmann (O'Reilly)
14. **Fundamentals of Data Engineering** — Joe Reis & Matt Housley (O'Reilly)
15. **CICIDS2017 Dataset** — https://www.unb.ca/cic/datasets/ids-2017.html