GCP Data Orchestration - Cloud Composer, Pub/Sub et Dataflow/DataProc

m2iformation.fr

Programme de la Formation

- Introduction écosystème Data GCP
- Cloud Composer (Airflow managé)
  - Architecture et concepts
  - DAGs et opérateurs
  - Scheduling et monitoring
- Pub/Sub (Messaging)
  - Topics et subscriptions
  - Patterns publish/subscribe
  - Intégration avec pipelines
- **Exercice 1** : Premier DAG Composer
- **Exercice 2** : Pipeline Pub/Sub
- Dataflow (Apache Beam)
  - Architecture streaming/batch
  - Pipelines et transformations
  - PCollections et ParDo
  - Windows et triggers
- DataProc (Spark/Hadoop)
  - Clusters éphémères
  - Jobs Spark et intégrations
- Architecture de référence
- **Exercice 3** : Pipeline Beam
- **TP Final** : Pipeline complet orchestré

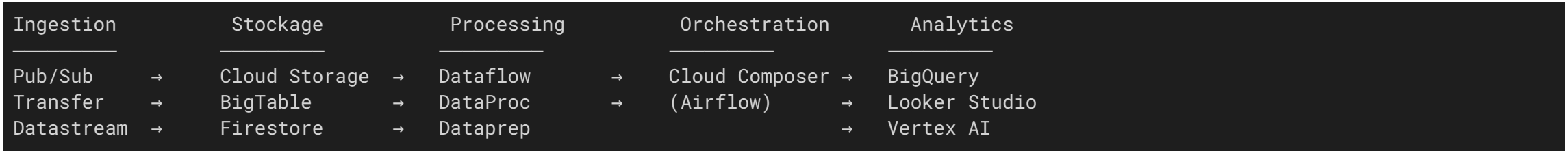
UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

Écosystème Data Engineering GCP

Architecture Data sur GCP

Stack Data GCP - Vue d'ensemble



Positionnement des services

- **Cloud Composer** : Orchestration et scheduling de workflows complexes
- **Pub/Sub** : Messaging asynchrone et streaming en temps réel
- **Dataflow** : Traitement unifié streaming/batch (Apache Beam)
- **DataProc** : Clusters Spark/Hadoop managés pour Big Data

Cas d'usage typiques

- ETL/ELT batch et temps réel
- Data Lakes et Data Warehouses
- Machine Learning pipelines
- Event-driven architectures

Pourquoi ces services ?

Cloud Composer

✓ Airflow managé (sans gestion infra)
✓ Orchestration de workflows complexes
✓ DAGs versionnés et testables
✓ Intégration native GCP
✓ Monitoring et alerting intégrés
✓ Scheduling avancé

Pub/Sub

✓ Messaging at scale (millions msg/sec)
✓ Découplage producteurs/consommateurs
✓ Garantie de livraison at-least-once
✓ Rétention jusqu'à 31 jours
✓ Ordering des messages (optionnel)
✓ Intégration native Dataflow

Dataflow

✓ Auto-scaling automatique
✓ Code unifié batch/streaming
✓ Gestion état et fenêtrage
✓ Exactly-once processing
✓ Support Python et Java
✓ Templates réutilisables

DataProc

✓ Clusters Spark/Hadoop en secondes
✓ Clusters éphémères (coût optimisé)
✓ Compatibilité OSS (Spark, Hive, Presto)
✓ Autoscaling des workers
✓ Intégration BigQuery/GCS
✓ Notebooks interactifs

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

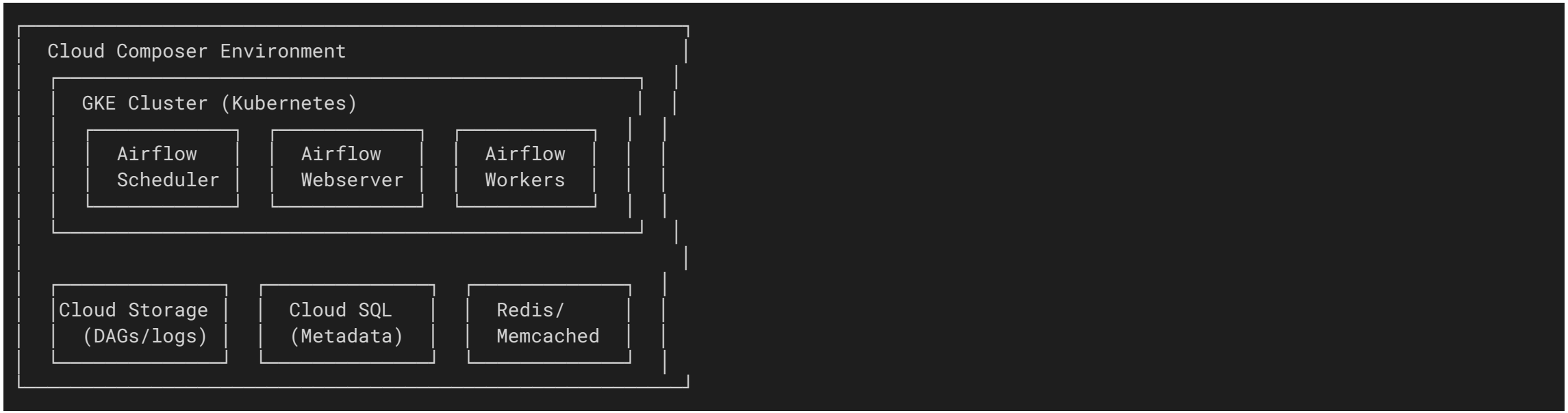
Cloud Composer (Apache Airflow Managé)

Qu'est-ce que Cloud Composer ?

Définition

Service GCP entièrement managé basé sur Apache Airflow pour orchestrer et planifier des workflows data complexes via des DAGs (Directed Acyclic Graphs).

Architecture Cloud Composer



Qu'est-ce que Cloud Composer ?

Composants principaux

- **Scheduler** : Déclenche les tâches selon le planning
- **Webserver** : Interface UI pour monitoring
- **Workers** : Exécutent les tâches (Celery Executor)
- **Metadata DB** : État des DAGs et runs (Cloud SQL)

Concepts Fondamentaux - DAG

DAG (Directed Acyclic Graph)

- Graphe orienté sans cycle définissant le workflow
- Composé de tâches (Tasks) et de leurs dépendances
- Possède un schedule et des paramètres de configuration

Création d'un Environnement Composer

Via gcloud CLI

```
# Créer un environnement Composer 2
gcloud composer environments create data-orchestration \
    --location europe-west1 \
    --image-version composer-2.5.0-airflow-2.5.3 \
    --environment-size small \
    --scheduler-cpu 2 \
    --scheduler-memory 4 \
    --scheduler-storage 2 \
    --scheduler-count 1 \
    --web-server-cpu 1 \
    --web-server-memory 2 \
    --web-server-storage 2 \
    --worker-cpu 2 \
    --worker-memory 7.5 \
    --worker-storage 2 \
    --min-workers 1 \
    --max-workers 3 \
    --network my-vpc-network \
    --subnetwork my-subnet

# Lister les environnements
gcloud composer environments list --locations europe-west1

# Obtenir les détails
gcloud composer environments describe data-orchestration \
    --location europe-west1
```

Déploiement de DAGs

Uploader un DAG vers Composer

```
# Récupérer le bucket GCS de l'environnement
BUCKET=$(gcloud composer environments describe data-orchestration \
    --location europe-west1 \
    --format="get(config.dagGcsPrefix)")

# Uploader le DAG
gsutil cp my_dag.py ${BUCKET}/dags/

# Uploader des dépendances Python
gsutil cp requirements.txt ${BUCKET}/

# Uploader des plugins personnalisés
gsutil cp -r plugins/* ${BUCKET}/plugins/

# Uploader des données de configuration
gsutil cp config.yaml ${BUCKET}/data/
```

Déploiement de DAGs

Structure recommandée du projet

```
airflow-project/
├── dags/
│   ├── etl_sales_daily.py
│   ├── etl_customers_weekly.py
│   └── ml_training_pipeline.py
├── plugins/
│   ├── custom_operators/
│   └── custom_hooks/
├── data/
│   └── config.yaml
├── requirements.txt
└── tests/
    └── test_dags.py
```

Monitoring et Debugging

Interface Web Airflow

- Accès via l'URL fournie dans la console GCP
- Vue Graph : Visualisation du DAG
- Vue Tree : Historique des exécutions
- Vue Gantt : Timeline des tâches
- Logs détaillés par tâche

Monitoring et Debugging

Logs et métriques

```python
import logging

def my_task(**context):
    # Logger dans Airflow
    logging.info("Starting task execution")
    logging.warning("This might take a while")

    # Accéder aux métadonnées
    dag_run = context['dag_run']
    logging.info(f"DAG Run ID: {dag_run.run_id}")
    logging.info(f"Execution date: {context['execution_date']}")

    try:
        result = process_data()
        logging.info(f"Processed {result['count']} records")
    except Exception as e:
        logging.error(f"Error processing data: {e}")
        raise

# Monitoring via Cloud Logging
# Les logs sont automatiquement exportés vers Cloud Logging
# Recherche : resource.type="cloud_composer_environment"
```

Best Practices Cloud Composer

1. Design des DAGs

```
# ✓ Faire : DAGs idempotents
# ✓ Faire : Tâches atomiques et réutilisables
# ✓ Faire : Gestion des erreurs appropriée
# ✗ Éviter : DAGs trop complexes (>50 tâches)
# ✗ Éviter : Tâches de longue durée (>30 min)

# Exemple d'idempotence
task = BigQueryInsertJobOperator(
    task_id='load_data',
    configuration={
        "query": {
            "query": """
                CREATE OR REPLACE TABLE `dataset.table_{{ ds_nodash }}` AS
                SELECT * FROM source WHERE date = '{{ ds }}'
            """,
            "useLegacySql": False,
        }
    },
)
```

Best Practices Cloud Composer

2. Performance

- Utiliser des opérateurs natifs GCP (plus performants)
- Paralléliser les tâches indépendantes
- Éviter de passer de grosses données via XCom (max 48 KB)
- Utiliser Cloud Storage pour les gros fichiers

3. Sécurité

- Utiliser des secrets pour les credentials (Secret Manager)
- Appliquer le principe du moindre privilège (IAM)
- Chiffrer les données sensibles

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

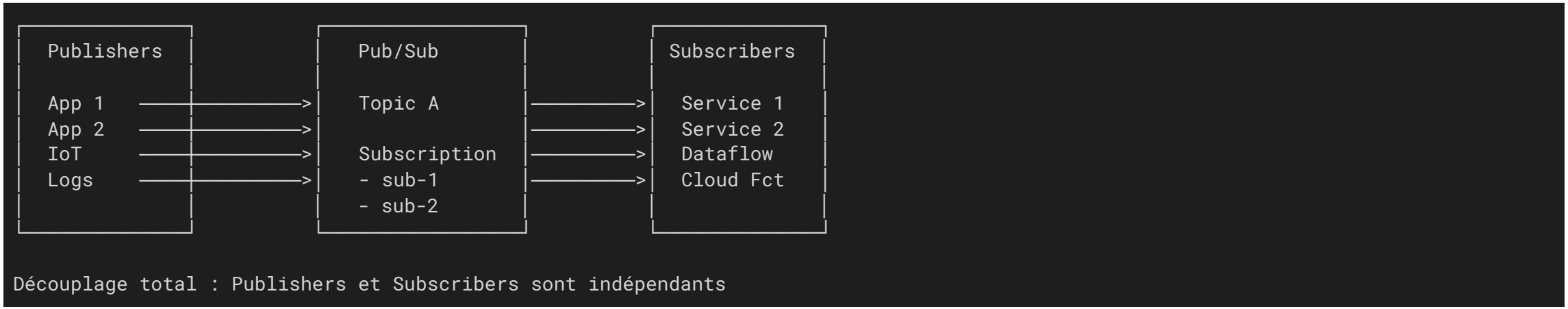
Google Cloud Pub/Sub

Qu'est-ce que Pub/Sub ?

Définition

Service de messaging asynchrone, fully-managed et hautement scalable pour des architectures événementielles et du streaming de données en temps réel.

Architecture Pub/Sub



Qu'est-ce que Pub/Sub ?

Concepts clés

- **Topic** : Canal de messages nommé
- **Subscription** : Endpoint de livraison des messages
- **Message** : Données + attributs (metadata)
- **Publisher** : Producteur de messages
- **Subscriber** : Consommateur de messages

Modèles de Messaging

1. **Push Subscription** - Pub/Sub pousse vers le subscriber

```
Topic → Subscription (Push) → HTTP(S) Endpoint (webhook)

Avantages :
✓ Pas de gestion de pull par l'application
✓ Idéal pour Cloud Functions, Cloud Run
✓ Latence faible

Configuration :
gcloud pubsub subscriptions create my-push-sub \
    --topic=my-topic \
    --push-endpoint=https://my-service.run.app/webhook
```

2. **Pull Subscription** - Le subscriber récupère les messages

```
Topic → Subscription (Pull) ← Application (polling)

Avantages :
✓ Contrôle du débit par le subscriber
✓ Batch processing efficace
✓ Idéal pour batch et ETL

Configuration :
gcloud pubsub subscriptions create my-pull-sub \
    --topic=my-topic
```

Garanties de Livraison

At-least-once delivery

- Chaque message est livré au moins une fois
- Possible de recevoir des duplicatas
- Le subscriber doit gérer l'idempotence

Message acknowledgment

```
from google.cloud import pubsub_v1

subscriber = pubsub_v1.SubscriberClient()
subscription_path = subscriber.subscription_path(project_id, subscription_id)

def callback(message):
    print(f"Received: {message.data}")

    try:
        process_message(message)
        message.ack()  # Acquitter le message (supprimé de la queue)
    except Exception as e:
        print(f"Error: {e}")
        message.nack()  # Ré-enqueuer le message pour nouvel essai

# Subscribe
streaming_pull_future = subscriber.subscribe(subscription_path, callback=callback)
```

Publishing Messages

Publisher Python

```python
from google.cloud import pubsub_v1
import json

project_id = "my-project"
topic_id = "sales-events"

publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path(project_id, topic_id)

# 1. Message simple
data = json.dumps({"order_id": "12345", "amount": 99.99}).encode("utf-8")
future = publisher.publish(topic_path, data)
message_id = future.result()
print(f"Published message ID: {message_id}")

# 2. Message avec attributs (metadata)
data = json.dumps({
    "transaction_id": "TX-789",
    "product": "laptop",
    "quantity": 2
}).encode("utf-8")

attributes = {
    "source": "web-app",
    "country": "FR",
    "priority": "high"
}

future = publisher.publish(topic_path, data, **attributes)
message_id = future.result()

# 3. Batch publishing (meilleure performance)
futures = []
for i in range(100):
    data = json.dumps({"id": i, "value": i * 10}).encode("utf-8")
```

Subscribing to Messages

Pull Subscriber Python

```python
from google.cloud import pubsub_v1
from concurrent.futures import TimeoutError
import json

project_id = "my-project"
subscription_id = "sales-events-sub"
timeout = 60.0

subscriber = pubsub_v1.SubscriberClient()
subscription_path = subscriber.subscription_path(project_id, subscription_id)

def callback(message):
    """Traiter chaque message"""
    print(f"Received message ID: {message.message_id}")
    print(f"Data: {message.data.decode('utf-8')}")
    print(f"Attributes: {message.attributes}")

    try:
        # Parser et traiter
        data = json.loads(message.data.decode('utf-8'))
        process_transaction(data)

        # Acquitter
        message.ack()
        print(f"Message {message.message_id} acknowledged.")

    except Exception as e:
        print(f"Error processing message: {e}")
        message.nack()  # Réessayer plus tard

# Subscriber asynchrone avec streaming pull
streaming_pull_future = subscriber.subscribe(
    subscription_path,
    callback=callback,
    flow_control=pubsub_v1.types.FlowControl(
        max_messages=100,  # Max messages en parallèle
        max_bytes=10 * 1024 * 1024,  # 10 MB buffer
    )
)

print(f"Listening for messages on {subscription_path}...")
```

Patterns d'Architecture Pub/Sub

1. **Fan-out Pattern** - Un message vers plusieurs consumers

```
                        ┌─> Subscription-1 → Service Analytics
Topic (Orders) ─────────┼─> Subscription-2 → Service Notifications
                        └─> Subscription-3 → Service Audit
```

2. **Work Queue Pattern** - Distribution de charge

```
Topic (Tasks) ──┬──> Subscription (multiple workers pulling)
                │        Worker-1
                │        Worker-2
                └──────> Worker-N
```

3. Event-driven ETL

```
Source DB → CDC → Pub/Sub Topic → Dataflow → BigQuery
                        │
                        └──────────> Cloud Function → Alerting
```

4. IoT Data Ingestion

```
IoT Devices → Pub/Sub → Dataflow → BigTable (real-time)
                  │                    ↓
                  └──────────────> Cloud Storage (archiving)
```

Filtering et Ordering

Message Filtering - Filtrer côté subscription

```
# Créer une subscription avec filtre
gcloud pubsub subscriptions create filtered-sub \
    --topic=sales-events \
    --message-filter='attributes.country="FR" AND attributes.priority="high"'

# Seuls les messages matchant le filtre sont livrés à cette subscription
```

```
# Publishing avec attributs pour filtrage
publisher.publish(
    topic_path,
    data=message_data,
    country="FR",       # Attribut filtrable
    priority="high",    # Attribut filtrable
    source="web"
)
```

Message Ordering - Garantir l'ordre des messages

```
# Enable ordering pour un topic
from google.cloud import pubsub_v1

publisher = pubsub_v1.PublisherClient()

# Publier avec ordering key
```

Pub/Sub avec Dataflow

Intégration native - Pub/Sub comme source streaming

```python
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
import json

# Options Dataflow
options = PipelineOptions(
    project='my-project',
    runner='DataflowRunner',
    region='europe-west1',
    temp_location='gs://my-bucket/temp',
    streaming=True
)

# Pipeline
with beam.Pipeline(options=options) as pipeline:
    messages = (
        pipeline
        | 'Read from Pub/Sub' >> beam.io.ReadFromPubSub(
            subscription='projects/my-project/subscriptions/my-sub'
        )
        | 'Decode' >> beam.Map(lambda x: json.loads(x.decode('utf-8')))
        | 'Transform' >> beam.Map(lambda x: {
            'id': x['id'],
            'amount': x['amount'],
            'processed_at': beam.window.TimestampedValue.now()
        })
        | 'Write to BigQuery' >> beam.io.WriteToBigQuery(
```

Démonstration 2 : Pipeline Pub/Sub

Messages en temps réel

Objectif : Créer un topic, publier des événements simulés, et les consommer

```
# 1. Créer le topic
gcloud pubsub topics create demo-events

# 2. Créer les subscriptions
gcloud pubsub subscriptions create demo-pull-sub --topic=demo-events
gcloud pubsub subscriptions create demo-analytics-sub --topic=demo-events
```

Publisher

```
# publisher.py
from google.cloud import pubsub_v1
import json
import time
import random

project_id = "my-project"
topic_id = "demo-events"

publisher = pubsub_v1.PublisherClient()
topic_path = publisher.topic_path(project_id, topic_id)

# Simuler des événements
events = ['page_view', 'click', 'purchase', 'signup']
```