ChatGPT

Conception d’une application ETL personnalisée avec Django

Ce rapport détaille une architecture ETL complète basée sur **Django**, inspirée de Talend, Azure Data Factory, Fivetran et Airbyte. Il présente les modules clés (connecteurs, exécution/orchestration, planification, transformations, monitoring), des exemples de modèles Django, les technologies complémentaires recommandées, ainsi qu’une approche de sécurité (authentification, RBAC, gestion des identifiants). Enfin, une proposition de schéma relationnel est donnée sous forme de tables Markdown.

Modules techniques clés

- **Connecteurs (sources/destinations)** – Chaque source ou destination est modélisée comme un « connecteur » réutilisable. Un connecteur représente typiquement une **API**, un **fichier**, une **base de données** ou un **data warehouse** [1]. Par exemple, on peut prévoir des connecteurs pour PostgreSQL, MySQL, API REST, fichiers CSV, S3, etc. La configuration de chaque connecteur (URL, login/mot de passe, paramètres) peut être stockée en base (JSON ou champs dédiés), idéalement de façon chiffrée ou via un gestionnaire de secrets.
- **Orchestration et exécution** – Les pipelines ETL sont orchestrés via des tâches asynchrones. Django fournit l’interface web et le modèle de données, tandis que Celery gère l’exécution parallèle des tâches ETL. Chaque *Pipeline* se décompose en étapes (`PipelineStep`), chacune pouvant déclencher une tâche Celery (extraction, transformation, chargement). Celery permet de composer des workflows complexes (chaînes, groupes, accords) et de déboguer les échecs. Comme le note un blog technique, « Celery est un outil essentiel qui permet de gérer des tâches asynchrones et de les exécuter de manière planifiée dans Django » [2]. On utilisera plusieurs workers Celery pour la scalabilité et la tolérance aux pannes.
- **Planification (Scheduling)** – La planification périodique des pipelines peut s’appuyer sur **Celery Beat** ou sur des tâches Cron. Avec **django-celery-beat**, on définit dynamiquement des tâches périodiques en base de données. Ainsi on peut, par exemple, planifier un pipeline d’extraction chaque nuit ou toutes les heures. Comme illustré dans un tutoriel, on peut créer un objet `PeriodicTask` lié à une tâche Celery pour automatiser l’exécution [2]. Le système doit aussi prévoir le déclenchement manuel de pipeline via l’interface ou des appels API.
- **Transformations** – Les transformations de données s’effectuent en code Python (p. ex. avec **Pandas** pour du traitement en mémoire) ou via des scripts SQL/NoSQL selon la source et la volumétrie. On peut également intégrer **Spark** ou des ETL basés sur SQL pour de gros volumes. Les étapes de transformation sont définies dans les `PipelineStep` et peuvent être réalisées par des fonctions Python réutilisables. Le code de transformation (ou une référence à un script) peut être stocké en base ou versionné dans le code source.
- **Monitoring et logging** – Chaque exécution de pipeline est journalisée pour le suivi. On conserve les logs et le statut (`succès` / `échec`) dans un modèle `TaskRun` (par pipeline) et, si besoin, `StepRun` (par étape). Pour la supervision temps réel, on peut intégrer **Flower**, la UI de Celery, qui expose des métriques prêtes à l’emploi pour Prometheus/Grafana [3]. Flower exporte notamment « plusieurs métriques de worker et de tâche Celery au format Prometheus » [3]. Des alertes ou notifications (par e-mail/SMS/Sentry) sont prévues en cas d’échec. Enfin, des vues d’administration (Django Admin ou interface custom) permettent de consulter l’historique des exécutions et l’état des pipelines.

Exemples de modèles Django (`models.py`)

Pour stocker les entités ETL, on définit des modèles Django clairs et extensibles. Par exemple :

```
from django.db import models
from django.contrib.auth.models import User

class Connector(models.Model):
    """
    Représente un connecteur de données (source ou destination).
    """
    TYPE_CHOICES = [
        ('database', 'Base de données'),
        ('api', 'API'),
        ('file', 'Fichier'),
        ('storage', 'Stockage cloud'),
        # etc.
    ]
    name = models.CharField(max_length=200, unique=True)
    connector_type = models.CharField(max_length=50, choices=TYPE_CHOICES)
    config = models.JSONField(blank=True)  # Paramètres (URL, port, etc.)
    # Les identifiants (login, mot de passe) peuvent être chiffrés ou gérés
via Vault.
```

```
class Pipeline(models.Model):
    """
    Représente un pipeline ETL complet.
    """
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
    name = models.CharField(max_length=200)
    description = models.TextField(blank=True)
    schedule_cron = models.CharField(
        max_length=100, blank=True,
        help_text="Expression CRON ou champ pour planification (ex: '0 0 * *
*')"
    )
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

```
class PipelineStep(models.Model):
    """
    Représente une étape d'un pipeline (extract, transform ou load).
    """
    STEP_TYPE_CHOICES = [
        ('extract', 'Extraction'),
        ('transform', 'Transformation'),
        ('load', 'Chargement'),
```

```
    ]
    pipeline = models.ForeignKey(
        Pipeline, related_name='steps', on_delete=models.CASCADE
    )
    order = models.PositiveIntegerField(help_text="Ordre d'exécution")
    name = models.CharField(max_length=200)
    step_type = models.CharField(max_length=20, choices=STEP_TYPE_CHOICES)
    connector = models.ForeignKey(
        Connector, null=True, blank=True,
        on_delete=models.SET_NULL,
        help_text="Connecteur utilisé si extraction ou chargement"
    )
    transformation_code = models.TextField(
        blank=True,
        help_text="Code Python/SQL pour l'étape de transformation (si
applicable)"
    )
```

```
class TaskRun(models.Model):
    """
    Journalise une exécution d'un pipeline.
    """
    pipeline = models.ForeignKey(Pipeline, on_delete=models.CASCADE)
    start_time = models.DateTimeField(auto_now_add=True)
    end_time = models.DateTimeField(null=True, blank=True)
    status = models.CharField(
        max_length=20,
        choices=[('pending','En attente'), ('running','En cours'),
('success','Succès'), ('failure','Échec')]
    )
    logs = models.TextField(blank=True, help_text="Journaux d'exécution et
erreurs")
```

```
class StepRun(models.Model):
    """
    Journalise l'exécution d'une étape au sein d'un pipeline en cours.
    """
    task_run = models.ForeignKey(TaskRun, related_name='step_runs',
on_delete=models.CASCADE)
    step = models.ForeignKey(PipelineStep, on_delete=models.CASCADE)
    start_time = models.DateTimeField(auto_now_add=True)
    end_time = models.DateTimeField(null=True, blank=True)
    status = models.CharField(
        max_length=20,
        choices=[('pending','En attente'), ('running','En cours'),
('success','Succès'), ('failure','Échec')]
    )
    logs = models.TextField(blank=True)
```

Ces modèles permettent de configurer dynamiquement des **pipelines** et leurs étapes. Lorsqu'un pipeline est lancé (via un appel API ou la planification), une instance `TaskRun` est créée, puis chaque étape déclenche une tâche Celery qui crée un `StepRun` avec le statut et les logs correspondants.

Technologies complémentaires

- **Celery** – Pour l'exécution asynchrone et l'orchestration des tâches (extraction, transformations, chargement). Comme noté précédemment, Celery est essentiel dans le contexte Django pour gérer des workflows ETL complexes [2]. On utilisera **Celery Beat** pour la planification périodique.
- **Broker Redis/RabbitMQ** – Celery nécessite un *message broker* pour la file d'attente. **RabbitMQ** et **Redis** sont les options les plus courantes. RabbitMQ gère bien les gros volumes de messages et permet le contrôle à distance des workers, tandis que Redis est simple et rapide pour de petits messages [4]. En pratique, on peut combiner *RabbitMQ (broker)* + *Redis (backend)* [4]. Pour la persistance des résultats à long terme (état des tâches), PostgreSQL (via SQLAlchemy) ou un stockage personnalisé peuvent être préférés [4].
- **Base de données (PostgreSQL)** – **PostgreSQL** est recommandée en production pour Django, grâce à sa robustesse et ses fonctionnalités avancées (JSONB, transactions, scalabilité) [4]. Elle stocke les modèles (Pipeline, TaskRun…) et peut servir de backend de résultats pour Celery.
- **Caches / Filesystems** – Un système de cache comme **Redis** (ou **Memcached**) peut optimiser le tri des tâches ou stocker des résultats intermédiaires. Pour les gros volumes de logs, on peut utiliser un stockage de fichiers ou d'objets (ex. S3) couplé à un lien stocké en base.
- **Conteneurs Docker** – Il est fortement conseillé de conteneuriser l'application (Django, Celery workers, broker, DB, etc.). Les conteneurs offrent un environnement stable et reproductible, avec toutes les dépendances encapsulées [5]. Un fichier `Dockerfile` + `docker-compose.yml` (ou Kubernetes) facilite le déploiement multi-serveurs. En particulier, « les conteneurs fournissent un environnement stable avec toutes les dépendances installées » [5], ce qui simplifie le déploiement et la montée en charge horizontale (ajout de workers).
- **Surveillance (Prometheus/Grafana, Sentry)** – Pour la scalabilité et le support, on déploie Prometheus+Grafana (récupérant les métriques de Flower/Celery) et des outils de logs centralisés. Des alertes Sentry ou Papertrail peuvent notifier en cas d'erreurs critiques.

Sécurité et gestion des utilisateurs

- **Authentification et autorisation** – Django intègre un système d'authentification complet [6]. Par défaut, les utilisateurs (`auth_user`), groupes et permissions sont gérés par `django.contrib.auth` [6]. On peut utiliser ce système natif pour contrôler l'accès aux pipelines (ex: chaque pipeline a un propriétaire, et on vérifie `request.user`). Pour une API REST, on peut utiliser **Django REST Framework** avec JWT ou OAuth2. Pour un modèle RBAC plus avancé (permissions dynamiques ou niveau objet), on peut compléter avec des packages tels que **django-guardian** (permissions par instance) ou **django-role-permissions**.
- **Stockage des identifiants** – Les credentials des connecteurs (mots de passe, tokens) ne doivent jamais être en clair dans le code. On utilise des variables d'environnement ou un **gestionnaire de secrets**. Par exemple, l'intégration de HashiCorp Vault via le client `hvac` permet de stocker et chiffrer dynamiquement les secrets [7]. Comme le souligne un article récent, « Vault fournit une façon plus sûre de gérer ces informations sensibles… en stockant de manière sécurisée les clés API et les identifiants via son moteur KV, et en générant des identifiants dynamiques pour les bases de données » [7]. Ainsi, la DB ou le code n'auront jamais les identifiants bruts.
- **Chiffrement et communications sécurisées** – Les échanges avec les sources cibles utilisent TLS/SSL. On chiffre les données sensibles au repos au besoin (ex: champs chiffrés en base ou

PostgreSQL chiffré). Les connexions à la base de données de l'application et au broker passent en mode sécurisé.

- **Contrôle d'accès (RBAC)** – On définit des rôles (par exemple *admin*, *ingénieur data*, *lecteur*) avec permissions associées (créer/éditer pipelines, exécuter/visualiser). En pratique, on peut associer des permissions Django aux modèles (`view_pipeline`, `change_pipeline`, etc.) et les grouper dans des rôles. Lorsqu'un utilisateur déclenche un pipeline, on vérifie ses permissions sur l'entité concernée. Les logs d'accès et d'exécution sont conservés pour audit.

Modèle relationnel (schéma de base de données)

Voici un exemple simplifié de schéma relationnel pour l'application ETL, avec les principales tables, champs clés, types et descriptions :

Table Connector

Champ	Type	Description
id (PK)	SERIAL	Identifiant unique auto-incrémenté
name	VARCHAR(200)	Nom du connecteur
connector_type	VARCHAR(50)	Type de connecteur (base, API, fichier, etc.)
config	JSONB	Paramètres de connexion (hôte, port, etc.)
created_at	TIMESTAMP	Date de création du connecteur
updated_at	TIMESTAMP	Date de dernière mise à jour

Table Pipeline

Champ	Type	Description
id (PK)	SERIAL	Identifiant du pipeline
owner_id (FK)	INT	Référence vers l'utilisateur propriétaire (auth_user.id)
name	VARCHAR(200)	Nom du pipeline
description	TEXT	Description détaillée du pipeline
schedule_cron	VARCHAR(100)	Expression CRON pour la planification (vide si manuel)
is_active	BOOLEAN	Indique si le pipeline est actif
created_at	TIMESTAMP	Date de création
updated_at	TIMESTAMP	Date de dernière modification

Table PipelineStep

Champ	Type	Description
id (PK)	SERIAL	Identifiant de l'étape
pipeline_id (FK)	INT	Référence vers `Pipeline.id` (cascade delete)

Champ	Type	Description
order	INT	Position d'exécution dans le pipeline
name	VARCHAR(200)	Nom de l'étape (p. ex. "Extract Customers")
step_type	VARCHAR(20)	Type d'étape (`extract` / `transform` / `load`)
connector_id (FK)	INT	Si extraction ou chargement : référence au connecteur associé (nullable)
transformation_code	TEXT	Code de transformation à exécuter (pour les étapes `transform`)

Table TaskRun

Champ	Type	Description
id (PK)	SERIAL	Identifiant de l'exécution de pipeline
pipeline_id (FK)	INT	Référence vers le `Pipeline` exécuté
start_time	TIMESTAMP	Date/heure de début d'exécution
end_time	TIMESTAMP	Date/heure de fin d'exécution
status	VARCHAR(20)	Statut (`pending`, `running`, `success`, `failure`)
logs	TEXT	Journal d'exécution (sorties standard, erreurs)

Table StepRun

Champ	Type	Description
id (PK)	SERIAL	Identifiant de l'exécution d'étape
task_run_id (FK)	INT	Référence vers `TaskRun` parent
step_id (FK)	INT	Référence vers l'étape `PipelineStep` associée
start_time	TIMESTAMP	Date/heure de début de l'étape
end_time	TIMESTAMP	Date/heure de fin de l'étape
status	VARCHAR(20)	Statut (`pending`, `running`, `success`, `failure`)
logs	TEXT	Journaux spécifiques à l'étape

Ces tables illustrent la structure relationnelle générale. On peut ajouter des tables auxiliaires (ex. **Group**, **Permission**, etc.) en utilisant les tables natives de Django pour la gestion des utilisateurs. Le champ `config` de `Connector` contient idéalement des références chiffrées aux identifiants d'accès plutôt que de les stocker en clair.

Sources : Documentation Airbyte sur les connecteurs [1], articles sur l'orchestration Celery [2] et la sécurité avec Vault [7], documentation Celery sur RabbitMQ/Redis [4], documentation Django sur le système d'authentification [6], et blog Docker sur la containerisation des apps [5]. Ces ressources montrent les bonnes pratiques pour construire un ETL modulaire, évolutif et sécurisé.

1 Connectors | Airbyte Docs
https://docs.airbyte.com/integrations/

2 Django-Celery-Beat : comment créer des tâches planifiées
https://blog.mikihands.com/fr/whitedec/2025/2/3/django-celery-beat-scheduled-task/

3 Prometheus Integration — Flower 2.0.0 documentation
https://flower.readthedocs.io/en/latest/prometheus-integration.html

4 Backends and Brokers — Celery 5.5.2 documentation
https://docs.celeryq.dev/en/stable/getting-started/backends-and-brokers/index.html

5 Dockerize a Django App: Step-by-Step Guide for Beginners | Docker
https://www.docker.com/blog/how-to-dockerize-django-app/

6 User authentication in Django | Django documentation | Django
https://docs.djangoproject.com/en/5.2/topics/auth/

7 Securing Django Applications with HashiCorp Vault (hvac): Concepts and Practical Examples - Simplico
https://simplico.net/2024/10/23/securing-django-applications-with-hashicorp-vault-hvac-concepts-and-practical-examples/