Moderniser un projet *dbt* legacy

Tests, observabilité et migration sans risque

Henri-Maxime
Ducoulombier

Senior Data Engineer

ducoulombier.hm@sfeir.com

Certified dbt Developer

SÉCURITÉ — GPG — BONNES PRATIQUES

Sécuriser l'échange de données sensibles avec GPG

Chaque fois qu'un mot de passe ou un fichier sensible est partagé par mail, un chaton meurt quelque part dans le monde. Ça n'est pas une fatalité et il existe une solution pour stopper cette tragédie: le chiffrement des échanges avec GPG (Gnu Privacy Guard)

SFEIR ! DEV

AUTOMATION TEST — TIPS — BONNES PRATIQUES — SUCCESS STORY

Vieille flûte!

L'histoire vraie et incroyable de l'affaire "Vieille flûte!".

HENRI-MAXIME DUCOULOMBIER

21 février 2024 . 11:53 AM — 6 lecture min

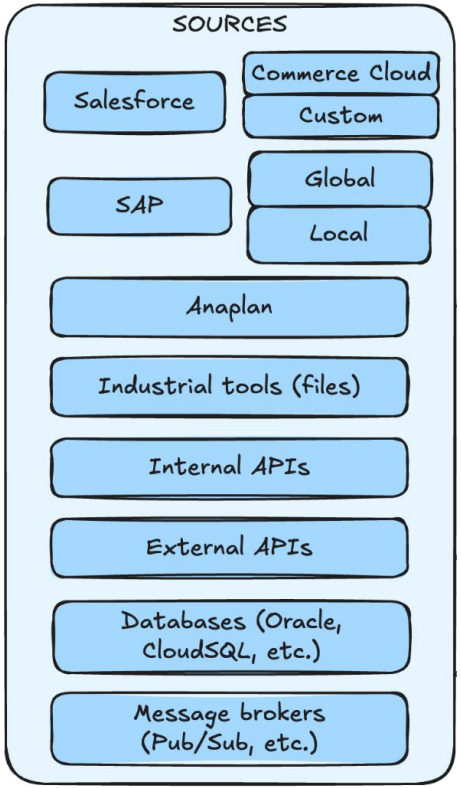
DATA — DBT — BONNES PRATIQUES — CODE — TIPS

DBT : des macros en cascade

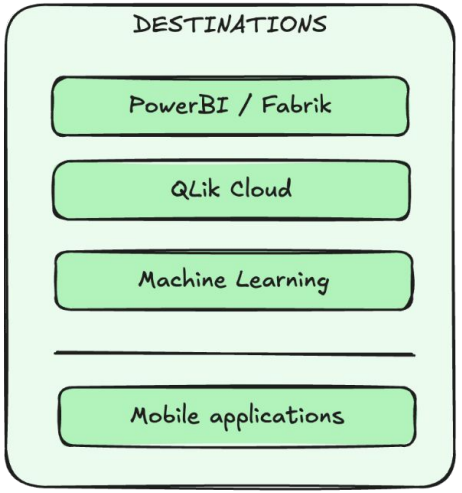
La gestion des packages dbt avec l'override des macros génériques et par domaine.

L’architecture et la stack data actuelle

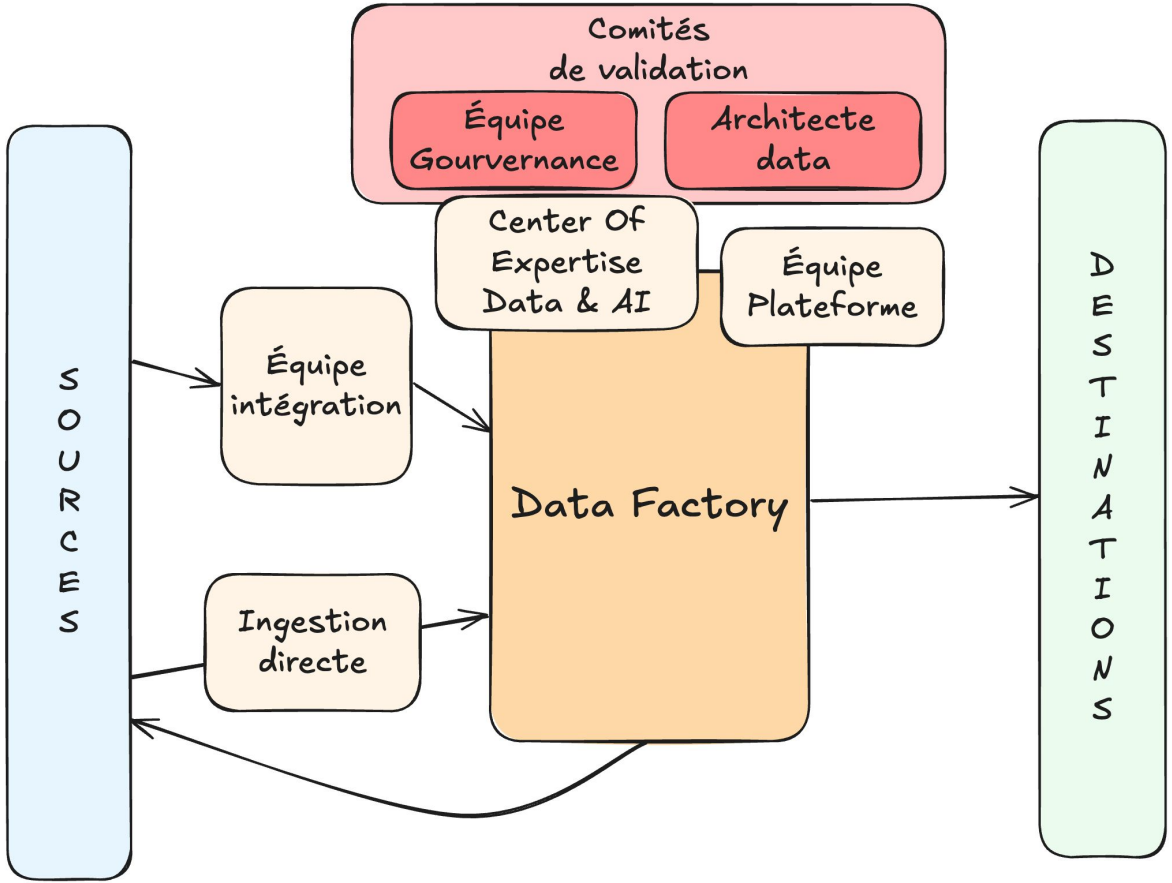
Architecture globale → sources



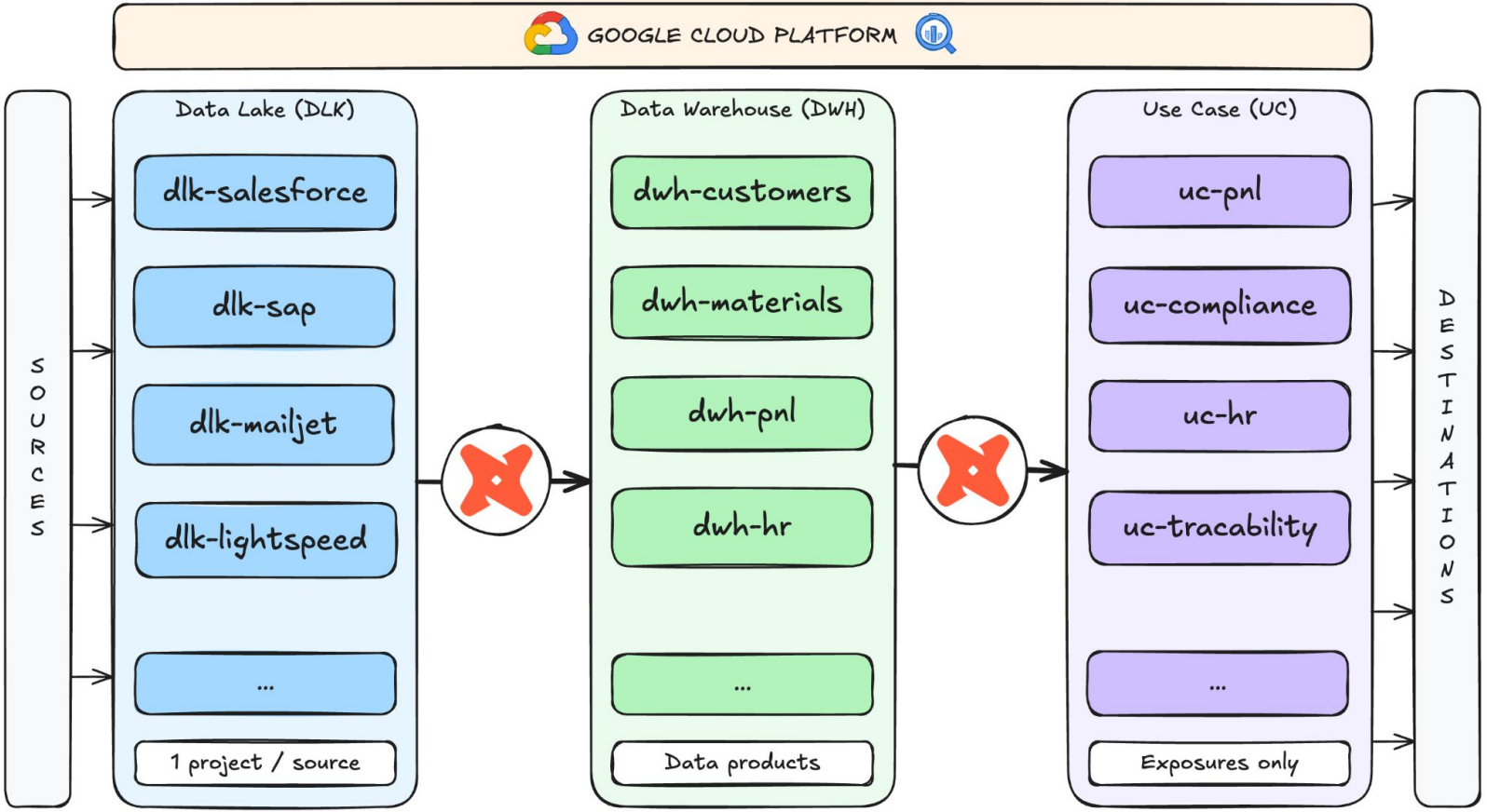
Architecture globale → destinations



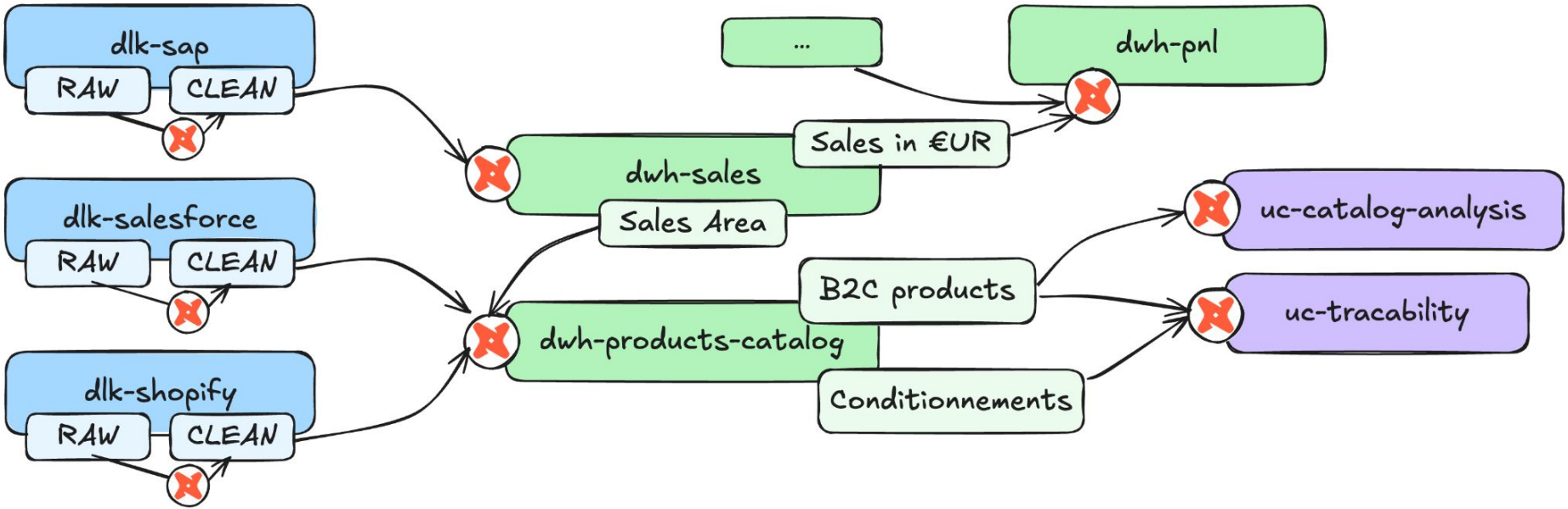
Architecture globale → data factory



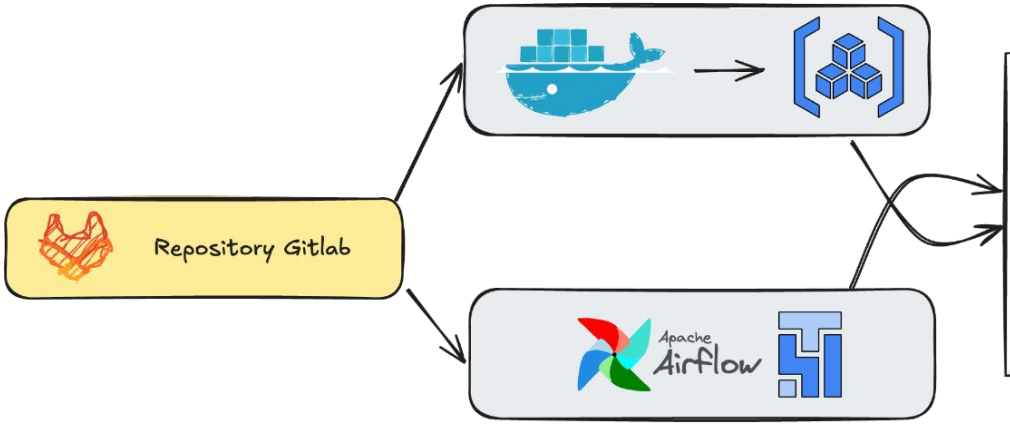
DLK → DWH → UC


GOOGLE CLOUD PLATFORM
SOURCES
Data Lake (DLK)
dlk-salesforce
dlk-sap
dlk-mailjet
dlk-lightspeed
...
1 project / source
Data Warehouse (DWH)
dwh-customers
dwh-materials
dwh-pnl
dwh-hr
...
Data products
Use Case (UC)
uc-pnl
uc-compliance
uc-hr
uc-tracability
...
Exposures only
DESTINATIONS

Dépendances entre les 3 layers



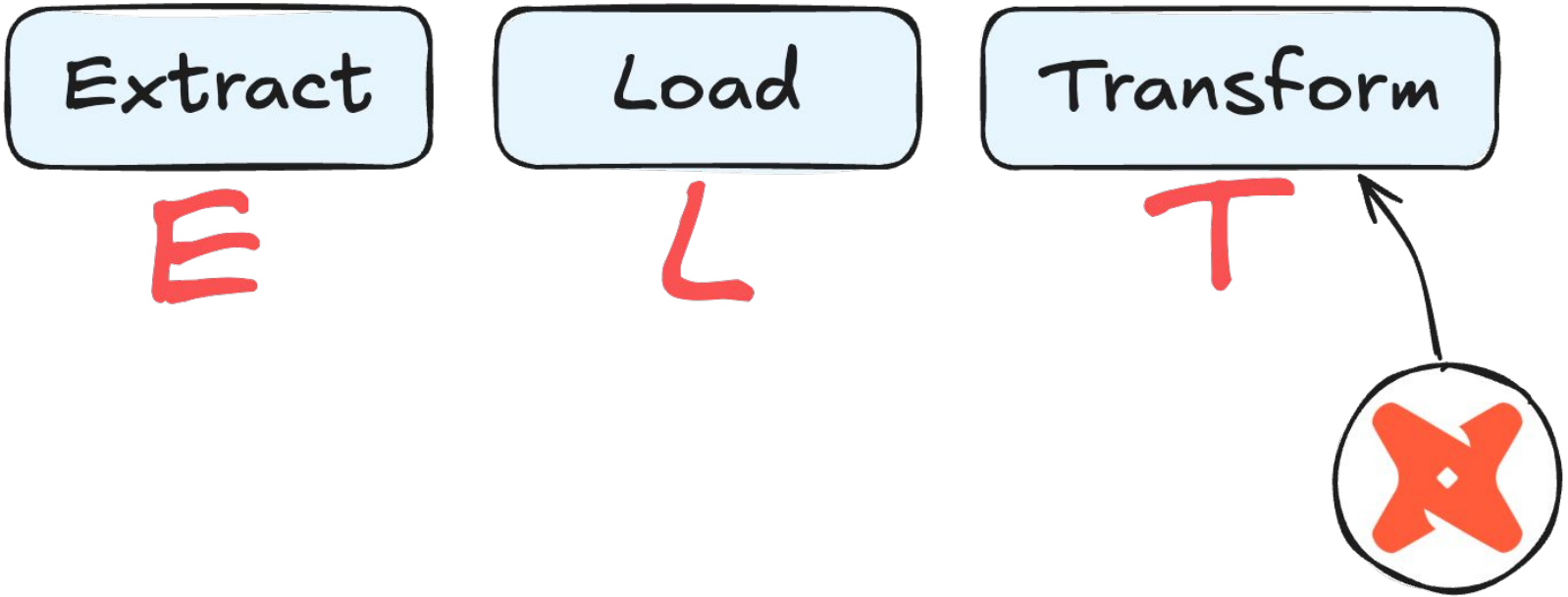
Orchestration des tâches



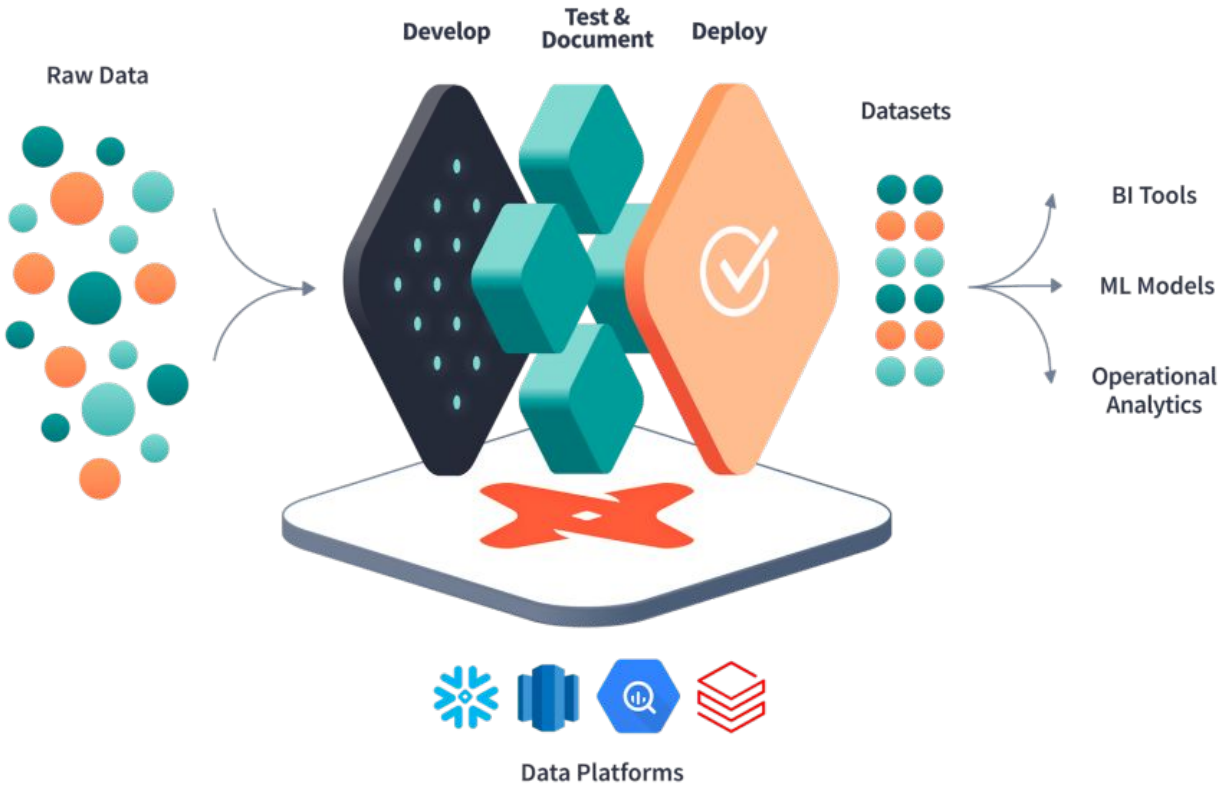
Time-based + data aware orchestration

dbt

Data Build Tool



Principe de fonctionnement



source: dbt labs

dbt c'est...

- **Transformations** de données décrites en **SQL**
- **Références et dépendances automatiques** entre modèles
- **Versionning** du code et **collaboration** facilitée
- **Tests, documentation** et **lineage**
- **Contrats** et **versions** de modèles
- **Modularité** et personnalisation

Killer feature: tests

Data tests

→ Valider la donnée dans les **sources**
→ Valider la donnée une fois **transformée**
→ Permet d'**interrompre le pipeline** en cas de problème

Unit tests

→ Valider le code **en dev et dans la CI/CD**
→ Valider des règles de calcul, du code métier
→ Permet de ne pas déployer du code défectueux

Gouvernance

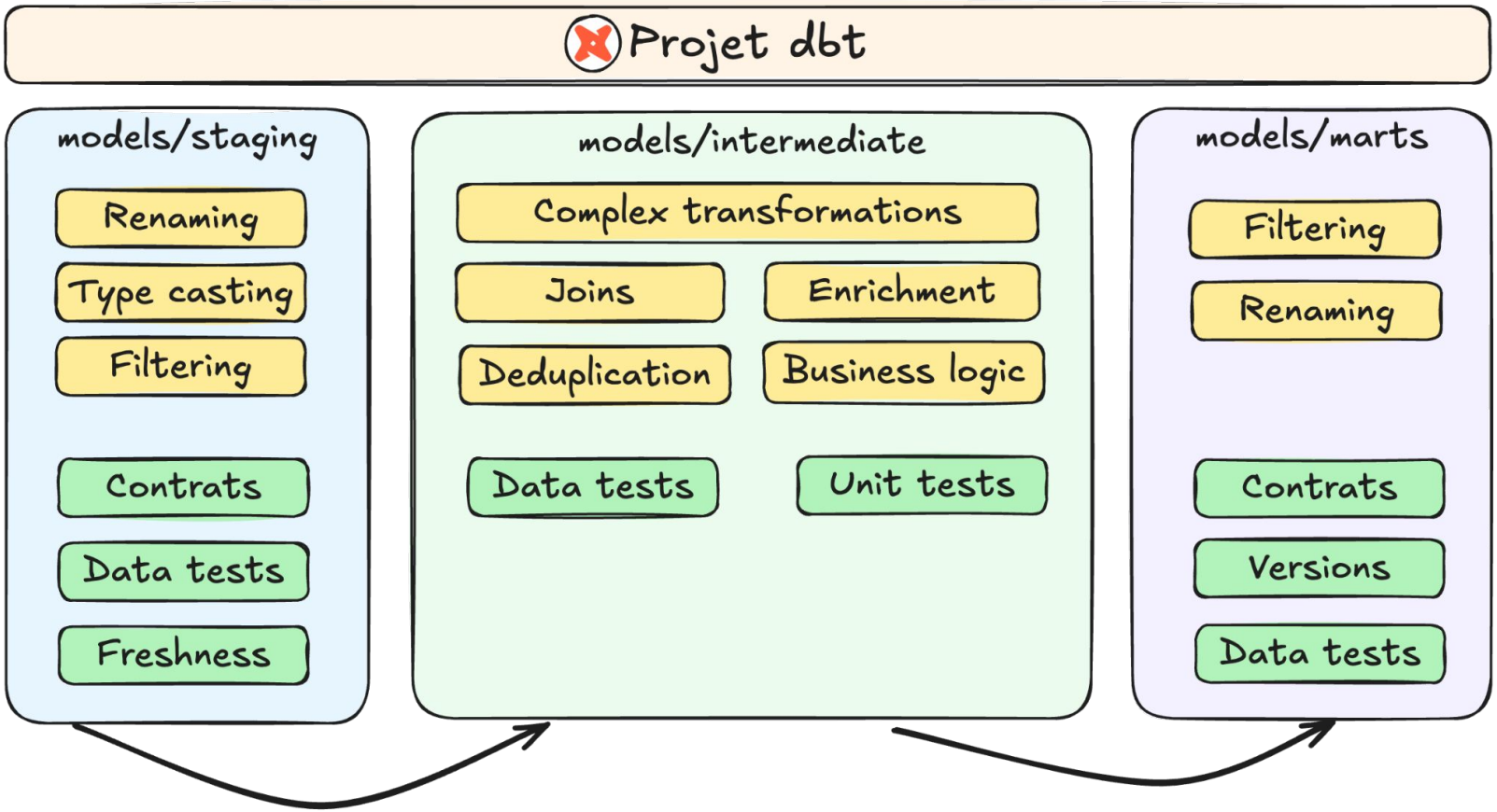
Data contracts

→ Garantir la **validité** des schémas de données
→ Ajouter des **contraintes**, déclarer des **clés**

Versions de modèles

→ Gérer les ***breaking changes*** dans les modèles
→ Introduire la notion de **cycle de vie** des modèles

Organisation d'un projet *dbt*



Rome ne s’est pas faite en un jour…

De vieux projets…

Premiers projets *dbt* datent de **2021**

- Version de dbt actuelle : **dbt 1.9** (1.10 en beta)
- Early adopters = essuyer les plâtres
- Ancien projet “*dataplatform*” qui ne devrait plus exister

```
docker pull fishtownanalytics/dbt:0.21.1
```

Arrivée tardive de la gouvernance

Mise en place réelle depuis début 2024

- *Policies difficiles à appliquer sur l'existant*
- *Beaucoup + de contraintes qu'auparavant*

Empilement et duplication de données

Difficulté à cataloguer et cartographier

- *Beaucoup de données, beaucoup de doublons*
- *Les policies historiques pas toujours respectées*
- *Shadow IT*

Maintenir l'existant

Criticité des applications existantes

- *Nécessité **absolue** de maintenir les use cases utilisés*
- *Pas de budget pour tout refaire*
- *Pas de volonté de tout refaire*

Error in SQL line 3871

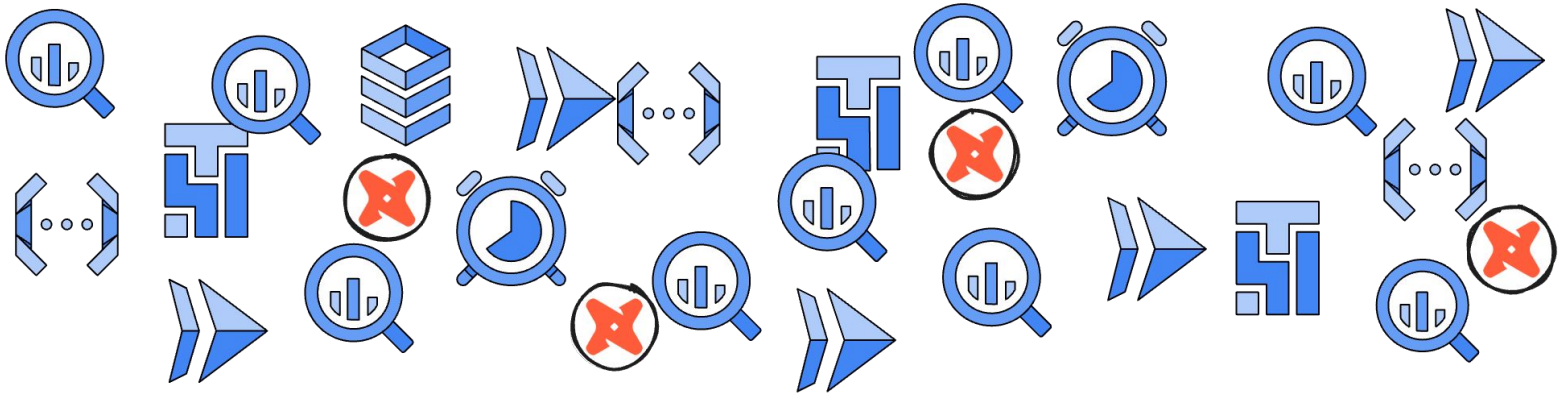
Le périmètre concerné

B2C

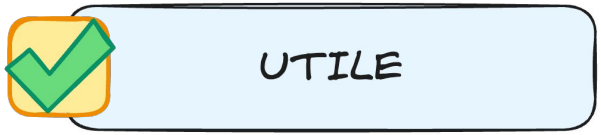
Base client direct

Marketing Intelligence

Media & performance



La règle des 3U



Pain points

Principaux problèmes sur le périmètre:

- *Non respect de l'architecture DLK -> DWH -> UC*
- *Vieilles versions des outils et packages*
- *Non respect des patterns et templates CI/CD*
- *Code mort caché un peu partout*
- *Opérations manuelles régulières (RUN x 1000)*

Ne pas nourrir le monstre

Besoins d'évolution

- *Supporter les nouveautés des **sources existantes***
- *Ajouter de **nouvelles sources** de données*
- ***Améliorer / corriger** les KPIs existants*
- *Ajouter de **nouveaux KPIs***
- *Appliquer de la **compliance** réglementaire*

Liste des courses

Comment procéder ?

- **Étape n°1 : Identifier le code mort et le désactiver**

  → Ne pas migrer ce qui est inutile
  → Transformer ce qui est inutilement complexe

- **Étape n°2 : Migration technique des outils**

  → Uniformiser les versions de ***dbt***
  → Corriger les configurations (auth, packages, etc.)
  → Mettre en place ***elementary data*** *sur le legacy*

Liste des courses

- **Étape n°3 : Réorganisation de la couche DLK**

  → Appliquer le pattern “1 source = 1 dlk”
  → Mise en place de tests et de contrats *dbt*

- **Étape n°4 : Réorganisation des projets *dbt DWH et UC***

  → Respecter la structure recommandée stg / int / mart
  → Mise en place de tests data / unitaires

- **Étape n°5 : Damage control**

Et elementary dans tout ça ?

Constat et besoins

- Les **journaux** ne sont pas suffisants
- Besoin **technique** et **métier**
- **Intégration** forte avec *dbt*
- **Compatibilité** avec tous types de tests
- Si possible avec une **interface sympa**

Le candidat idéal

+

Fonctionnement d'elementary

- Package *python* **ET** package *dbt*
- *elementary* **surcharge TOUS** les tests dbt
- **pre-hook** et **post-hook** à chaque exécution *dbt*
- Génération **asynchrone** des rapports
- **Possibilité** d'envoyer des alertes

Setup d'elementary

- À l'installation et à chaque montée de version
  → Init / upgrade des modèles *elementary*

```
dbt run --select elementary
```

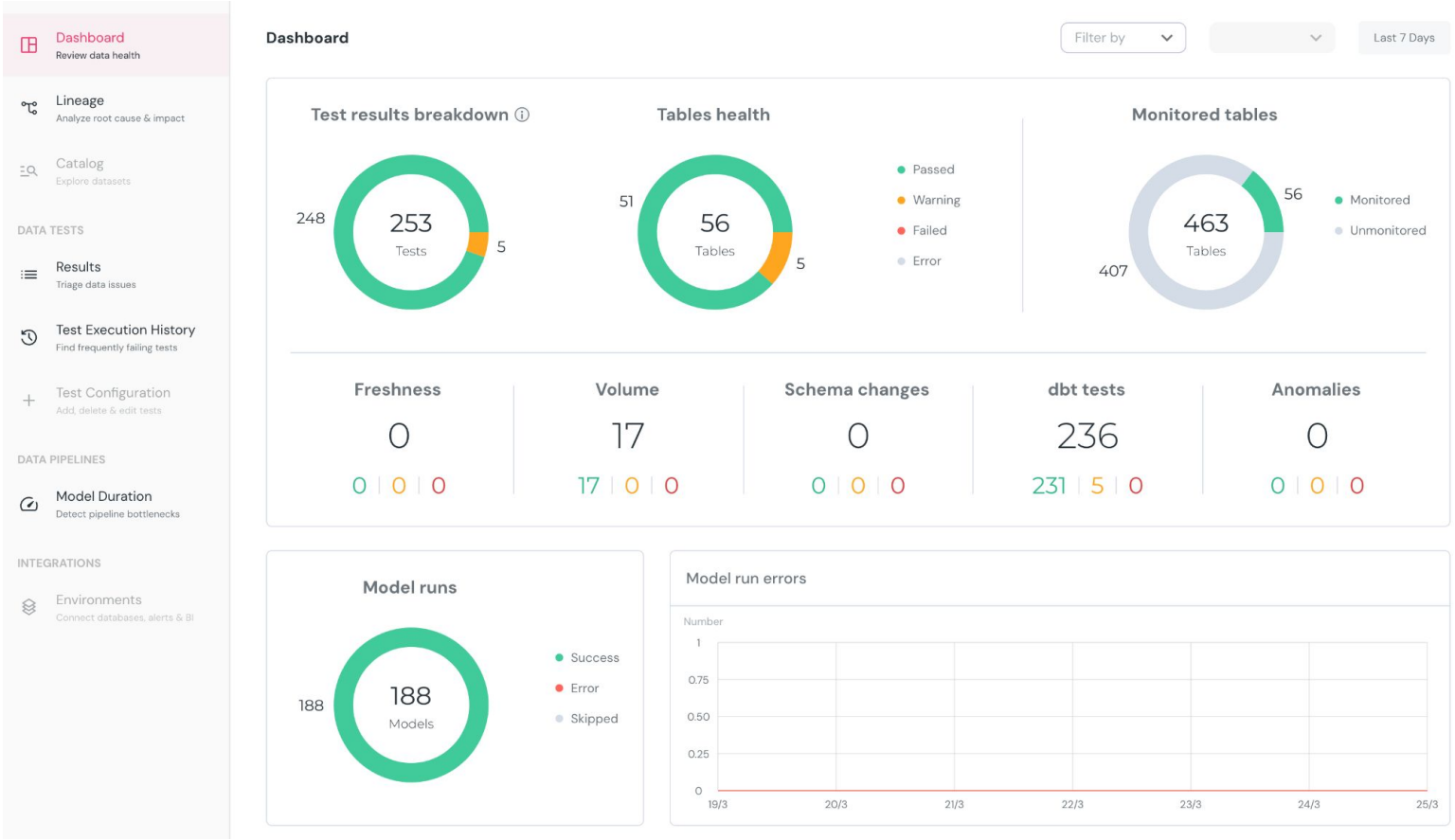
- Création **obligatoire** d'un profile *dbt* dédié
  → L'outil cherche un profil nommé "elementary"

- Ajout d'un "selector" par défaut qui ignore *elementary*
  → Se fait dans le *selectors.yml*
    → Ne pas lancer le setup à chaque fois
  → Désactiver les hooks facilement via une variable d'env

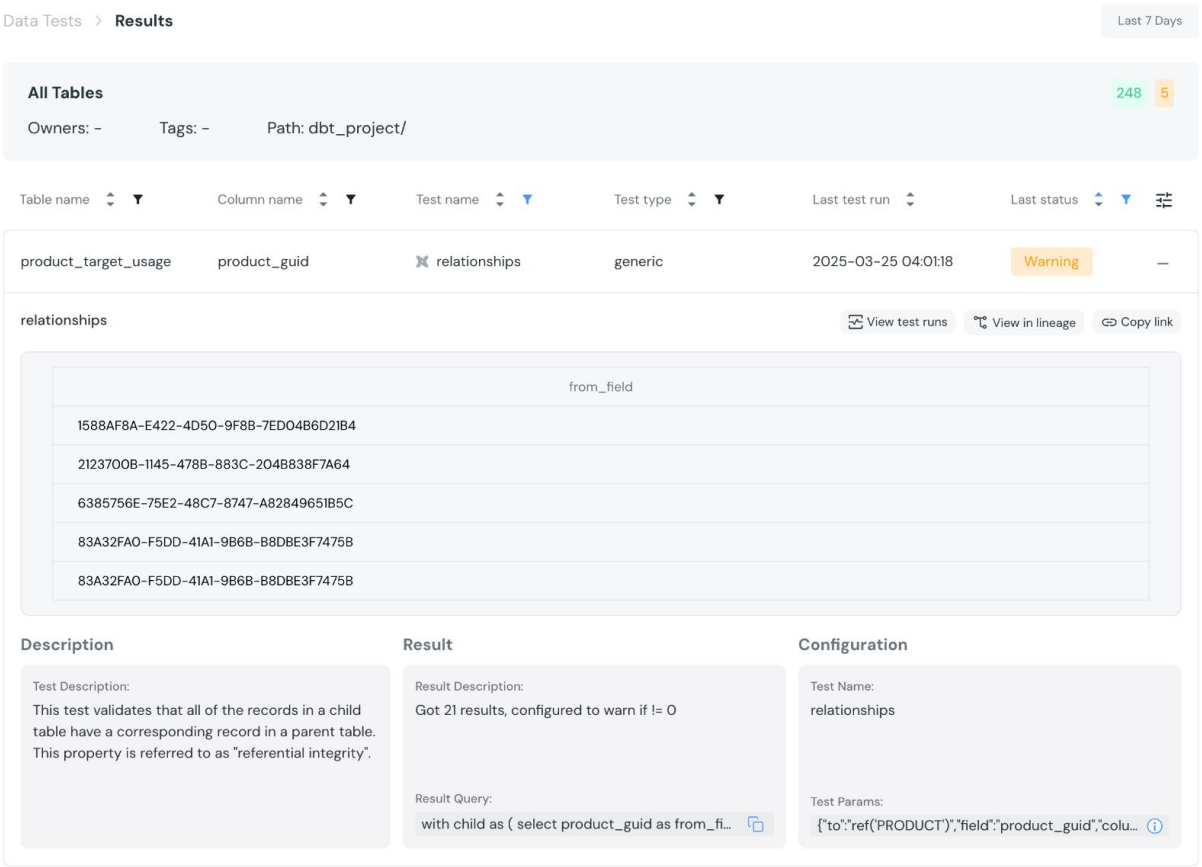
Configuration de l'orchestration

- *elementary* qui plante → non critique
- générer les rapports = longue opération
  → le faire en asynchrone
- génération de rapport peu importe le statut
  → rapport d'erreurs / warnings

Exemple de dashboard - homepage



Exemple de dashboard - tests results



Exemple de dashboard - temps d'exécution

Data Pipelines > **Model Duration**

Last 7 Days

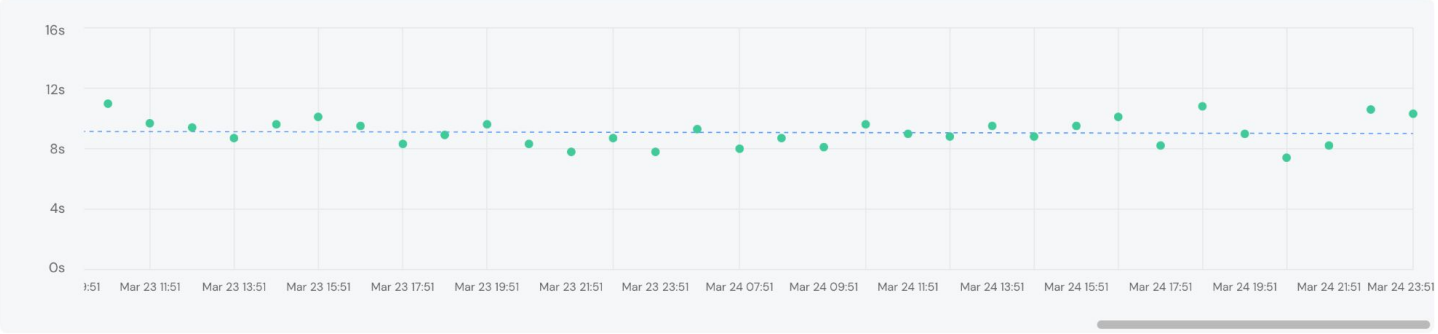
All Tables

Owners: - Tags: - Path: dbt_project/

Schema	Name	Last exec. time	Median exec. time	Ex time change rate	Results	Last run status
dprd_facturation	contrat_ligne_prestatio…	10.3 Secs	9.1 Secs	↗ 13.2%	126	Success

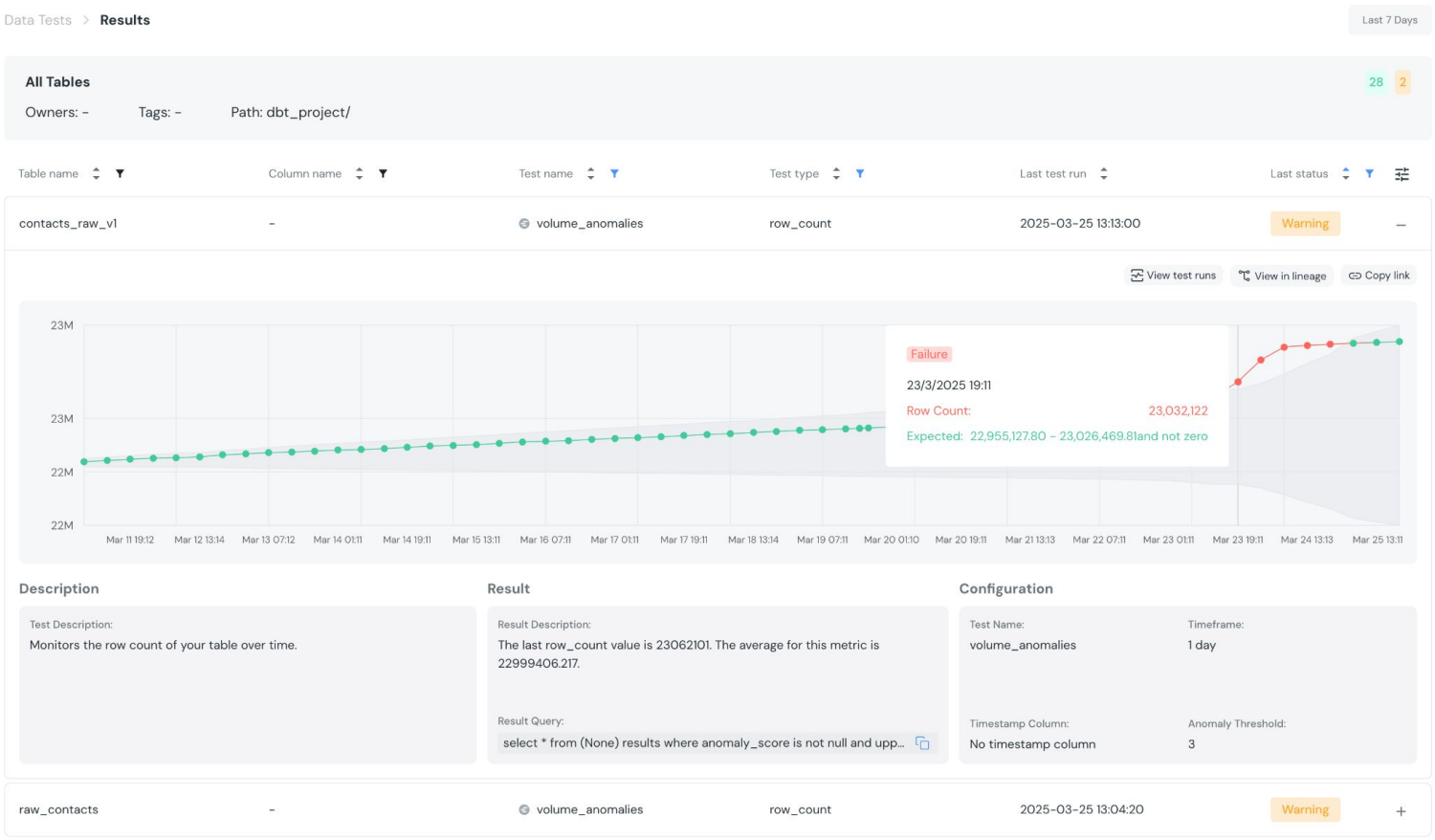
Model execution time and status over time

View in lineage Copy link



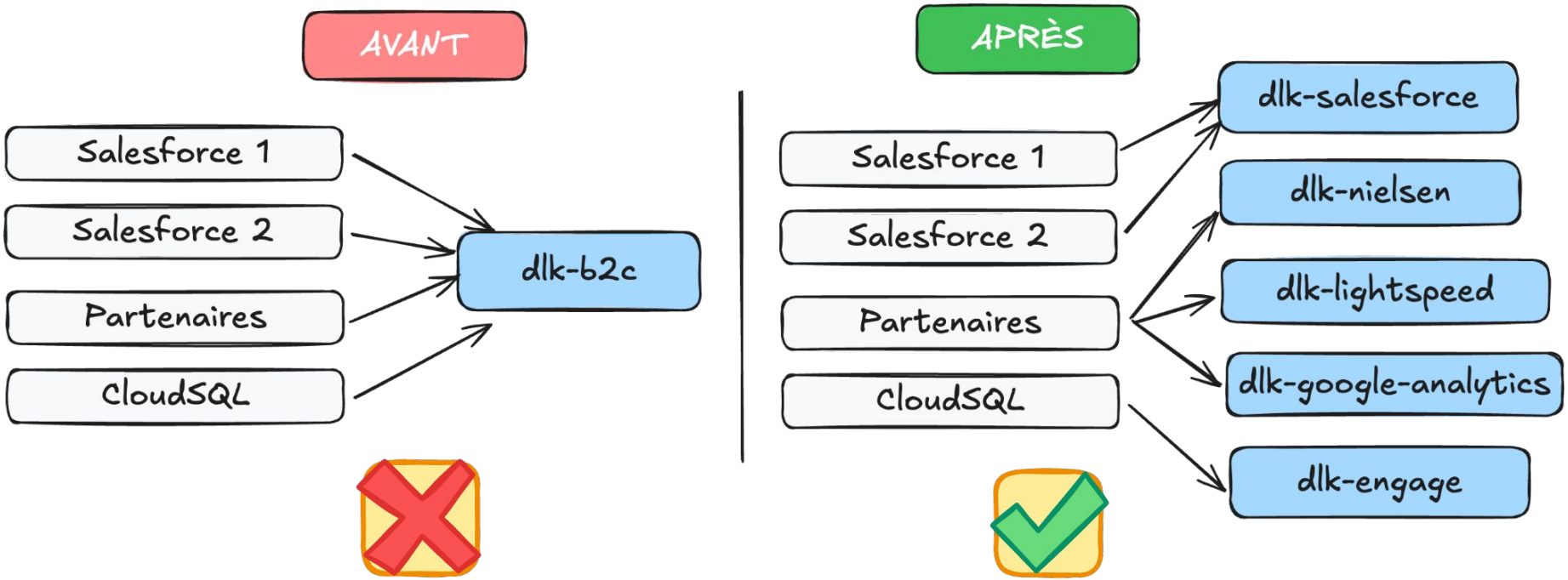
dprd_facturation	contrat_ligne	8.9 Secs	9.2 Secs	↘ 3.3%	126	Success
dprd_facturation	contrat	7.7 Secs	7.9 Secs	↘ 2.5%	126	Success
dprd_facturation	realise	7 Secs	6.7 Secs	↗ 4.5%	126	Success

Exemple de dashboard - anomalie de volume



Les changements appliqués

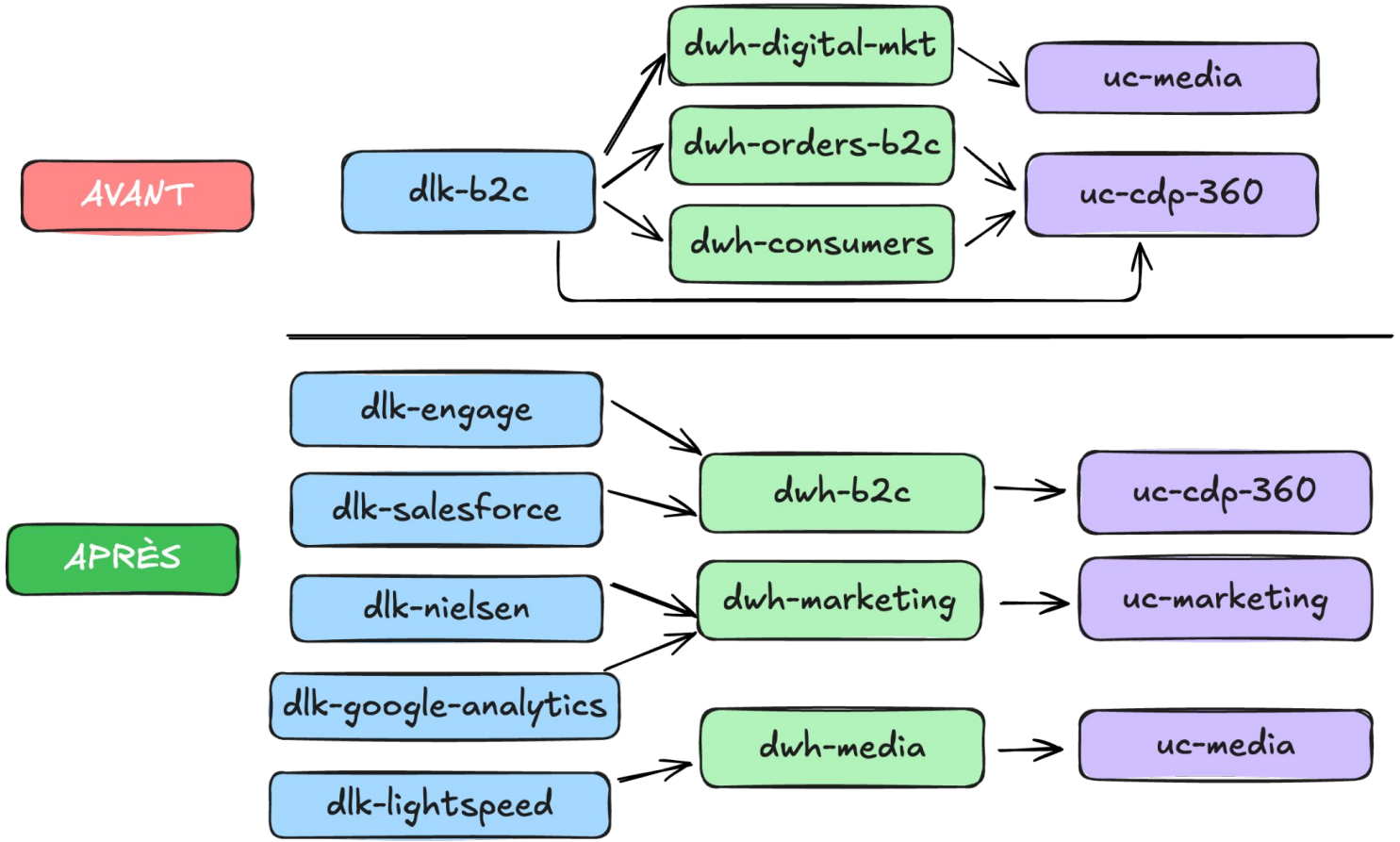
Réorganiser les sources



Et tester les sources

- Tests sur volumes / dimensions / colonnes =elementary
- Mise en place de **contrats et contraintes**
- Si possible, ajout de **timestamps** et freshness

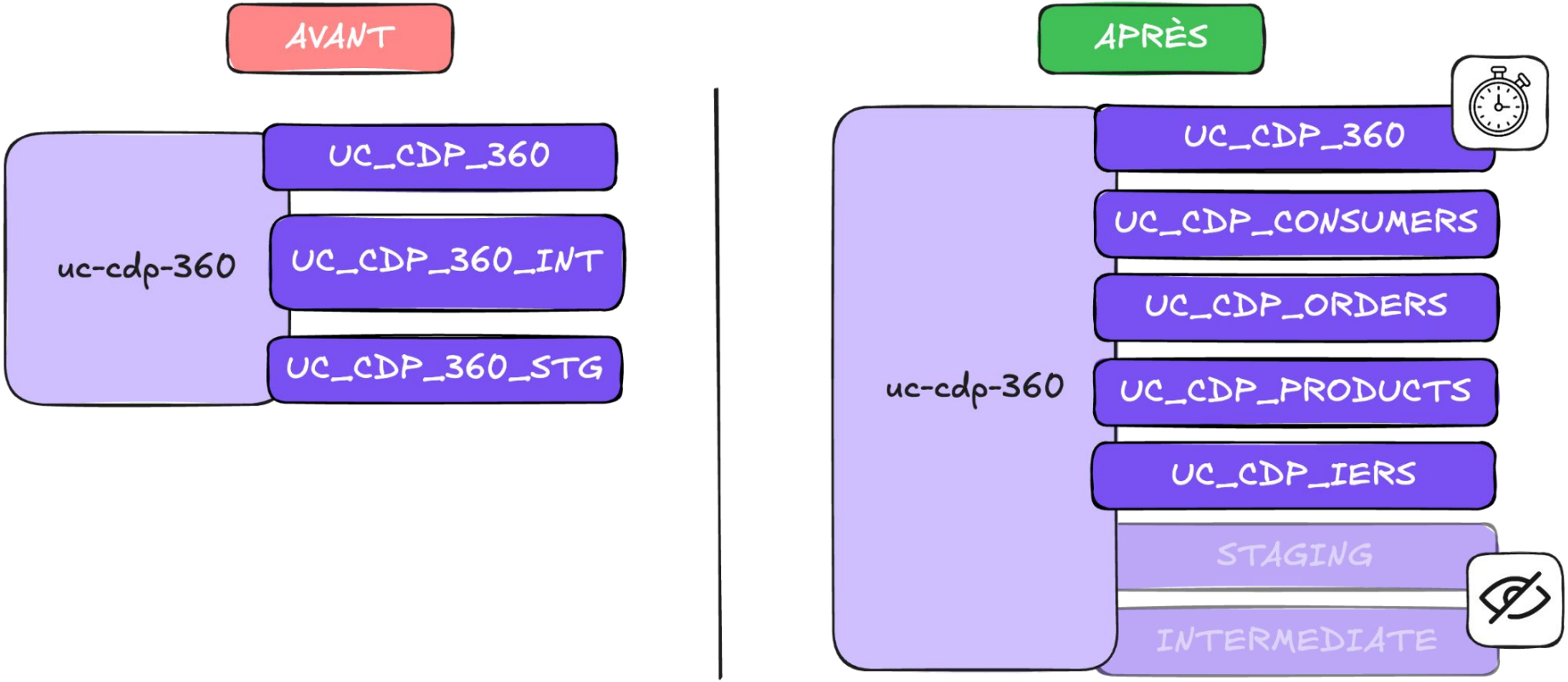
Créer de nouveaux data products



Et tester ces nouveaux data products

- Tests de **réconciliation** avant / après =elementary
- **Data & Units Tests** sur le code des règles métiers =elementary
- Contrats sur **tous** les datasets de sorties
- Mise en place de **versions**
- Suivi des **temps d'exécution** des pipelines =elementary

Exposer de nouveaux assets



Et tester ces nouveaux assets

elementary

- Mesure des **volumes** et **distribution** de donnée
- Contrats sur **tous** les datasets produits
- Mise en place de **versions**

Exemples de test

Les premiers tests basiques

Data tests *génériques*

- Sur les **colonnes**
- Dans le **YAML**
- Via des **packages**
  - ***dbt_utils***
  - ***dbt_expectations***
  - ***etc.***

```yaml
models:
  - name: INT__CONSUMERS
    description: |
      Merged sources for whole contact sources
    columns:
      - name: __source
        description: |
          Source system
        data_tests:
          - accepted_values:
              values: ["SFCC", "SAPECC", "SAPB1", "SHOPIFY"]
      - name: customer_code
        description: |
          Normalized customer code
        data_tests:
          - dbt_expectations.expect_column_values_to_match_regex:
              regex: "^\d{10}$"
```

Les premiers tests basiques

Data tests *génériques*

- Sur les **sources & modèles**
- Dans le **YAML**
- Via des **packages**
  - ***dbt_utils***
  - ***dbt_expectations***
  - ***etc.***

```yaml
sources:
  - name: SAP
    project: "dlk-sap-{{ env_var('ENV', 'dev') }}"
    dataset: SAP"
    tables:
      - name: AUFK
      - name: MAKT
        data_tests:
          - dbt_utils.unique_combination_of_columns:
              combination_of_columns:
                - MANDT
                - MATNR
                - SPRAS
      - name: T001L
      - name: T001W
```

Gestion de la sévérité

Error ou Warn ?

- **Error** → arrête le pipeline
- **Warn** → logs uniquement
- **Avec conditions**

```yaml
models:
  - name: INT__CONSUMERS
    columns:
      - name: __source
        data_tests:
          - accepted_values:
              values: ["SFCC", "SAPECC", "SAPB1", "SHOPIFY"]
              config:
                severity: error
                error_if: ">10000"
                warn_if: ">100"
      - name: phone_number
        data_tests:
          - not_empty:
              config:
                severity: warn
```

Les génériques personnalisés

Tests personnalisés

- Sur les sources & modèles
- En SQL + YAML
- À écrire soit même
- Plusieurs tests en un seul
- Créer des packages

```sql
{% test acme_phone_number_format(model, column_name, min_length=10) %}
SELECT {{column_name }}
FROM {{ model }}
WHERE
    {{ column_name }} IS NULL
    OR LENGTH({{ column_name }}) < {{ min_length }}
    OR NOT REGEXP_CONTAINS({{ column_name }}, '^\\+?[0-9]+$')
{% endtest %}
```

```yaml
models:
  - name: INT__CONSUMERS
    columns:
      - name: mobile_phone_number
        data_tests:
          - acme_phone_number_format:
              minimum_length: 8
```

Les tests singular

Tests singular + complexes

- Sur les **sources & modèles**
- En **SQL uniquement**
- Tests **fonctionnels**
- Utiliser les **relations**

```sql
{{ config(severity = 'warn') }}

SELECT
    o.order_id
FROM
    {{ ref('orders') }} o
LEFT JOIN
    {{ ref('order_items') }} oi ON o.order_id = oi.order_id
LEFT JOIN
    {{ ref('products') }} p ON oi.product_id = p.product_id
GROUP BY
    o.order_id
HAVING
    COUNT(DISTINCT oi.product_id) > 10
    OR SUM(oi.quantity * p.price) < 0
```

Les tests de fraîcheur

Avec ou sans timestamp

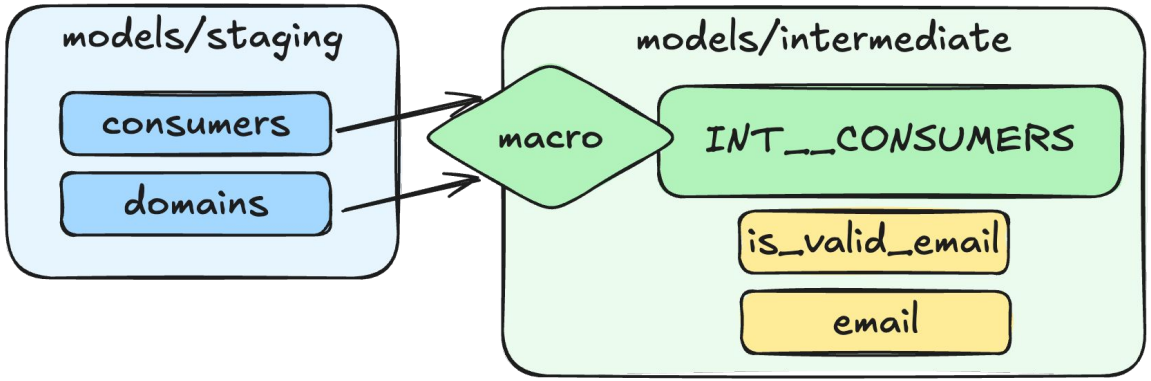
- Seulement sur les **sources**
- **Error** ou **Warn**
- En **début** de pipeline

```yaml
sources:
  - name: SAP
    project: "dlk-sap-{{ env_var('ENV', 'dev') }}"
    dataset: SAP
    loaded_at_field: __ingestionTimestamp
    tables:
      - name: MAKT
        freshness:
          warn_after: { count: 24, period: hour }
          error_after: { count: 96, period: hour }
      - name: T001W
        loaded_at_field: null
        freshness:
          error_after: { count: 12, period: hour }
```

Les tests unitaires

Éviter les régressions

- Exécution **locale** et/ou dans la **CI/CD**
- **Valider** le code métier
- Long à écrire



Les tests unitaires

SI (given)

consumers =

```
[
    {"id": "101", "email": "xxx@yyy.com"},
    {"id": "102", "email": null},
    {"id": "103", "email": "zzz@xxx.com"}
]
```

domains =

```
[
    {"domain": "yyy.com"},
    {"domain": "yahoo.com"}
]
```

ALORS (expect)

INT__CONSUMERS =

```
[
    {"id": "101", email: "xxx@yyy.com", "is_valid_email": true}
    {"id": "102", email: null,
    {"id": "103", email: "zzz@yyy.com"
]
```

SINON

Contrats et versions

Contrats de données

Forcer types et contraintes

- Au niveau *staging*
- Au niveau *marts (output)*
- YAML ← → SQL
- avant l'écriture des données

```yaml
models:
  - name: STG__EDA_REFERENTIALS
    config:
      contract:
        enforced: true
    constraints:
      - type: primary_key
        columns: [ reference_code, __filename ]
    columns:
      - name: __source
        data_type: STRING
      - name: reference_code
        data_type: INT64
      - name: referential_type_label
        data_type: STRING
      - name: reference_label
        data_type: STRING
      - name: __filename
        data_type: STRING
      - name: __ingestionTimestamp
        data_type: FLOAT64
```

Versions de modèles

Avancer sans casser

- Au niveau *marts* (output)
- 1 fichier .sql par version
- Bien gérer la durée de vie
- Attention au coût

```yaml
models:
  - name: CONSUMERS
    config:
      contract:
        enforced: true
    latest_version: 2
    versions:
      - v: 1  # CONSUMERS_v1.sql
        deprecation_date: "2024-10-17 00:00:00.00+00:00"
        config:
          enabled: false
      - v: 2  # CONSUMERS_v2.sql
        columns:
          - name: preferences
            data_type: ARRAY
          - name: preferences.key
            data_type: STRING
          - name: preferences.value
            data_type: ARRAY<STRING>
```

Les tests *elementary*

Elementary : paramétrage

Quelques paramètres **communs:**

- *timestamp_column* : analyser les changements dans le temps
- *severity* : réglable par test, WARN ou ERROR
- *anomaly_direction* : détection des pics, des creux, ou des deux
- *detection_period* et *training_period*

Elementary : tests de volume

```yaml
models:
  - name: CONTACTS_PROCESSED
    config:
      contract:
        enforced: true
      elementary:
        timestamp_columns: updated_at
    data_tests:
      - elementary.volume_anomalies:
          severity: warn
          fail_on_zero: true
      - at_least_one_not_null:
          combination_of_columns:
            - email
            - wechat_id
            - line_id
```

Elementary : tests de répartition des valeurs

```yaml
models:
  - name: CONTACTS_PROCESSED
    config:
      contract:
        enforced: true
      elementary:
        timestamp_columns: updated_at
    data_tests:
      - elementary.dimension_anomalies:
          dimensions:
            - __source
            - brand
            - market
```

Elementary : anomalies de colonnes

```yaml
models:
  - name: CONTACTS_PROCESSED
    config:
      contract:
        enforced: true
      elementary:
        timestamp_columns: updated_at
    columns:
      - name: first_order_delay
        data_tests:
          - elementary.column_anomalies:
              severity: warn
              column_anomalies:
                  - average
                  - standard_deviation
                  - min
                  - max
```

Elementary : (toutes) anomalies de colonnes

```yaml
models:
  - name: CONTACTS_PROCESSED
    config:
      contract:
        enforced: true
      elementary:
        timestamp_columns: updated_at
    data_tests:
      - elementary.all_columns_anomalies:
          severity: warn
          column_anomalies:
              - average
              - standard_deviation
              - min
              - max
```

TL;DR

Leçons apprises

Respecter les guidelines, les policies

Utiliser les tests, les contrats, les versions

Travailler avec le métier pour avancer par étape

Restructurer au fur et à mesure

Décomposer les modèles en étapes

On oublie vite la syntaxe des insert / update

MERCI !