Data Platforms and the Cloud

Rahul Dave(@rahuldave), Univ.Ai

Univ.AI

Data Engineering Lifecycle

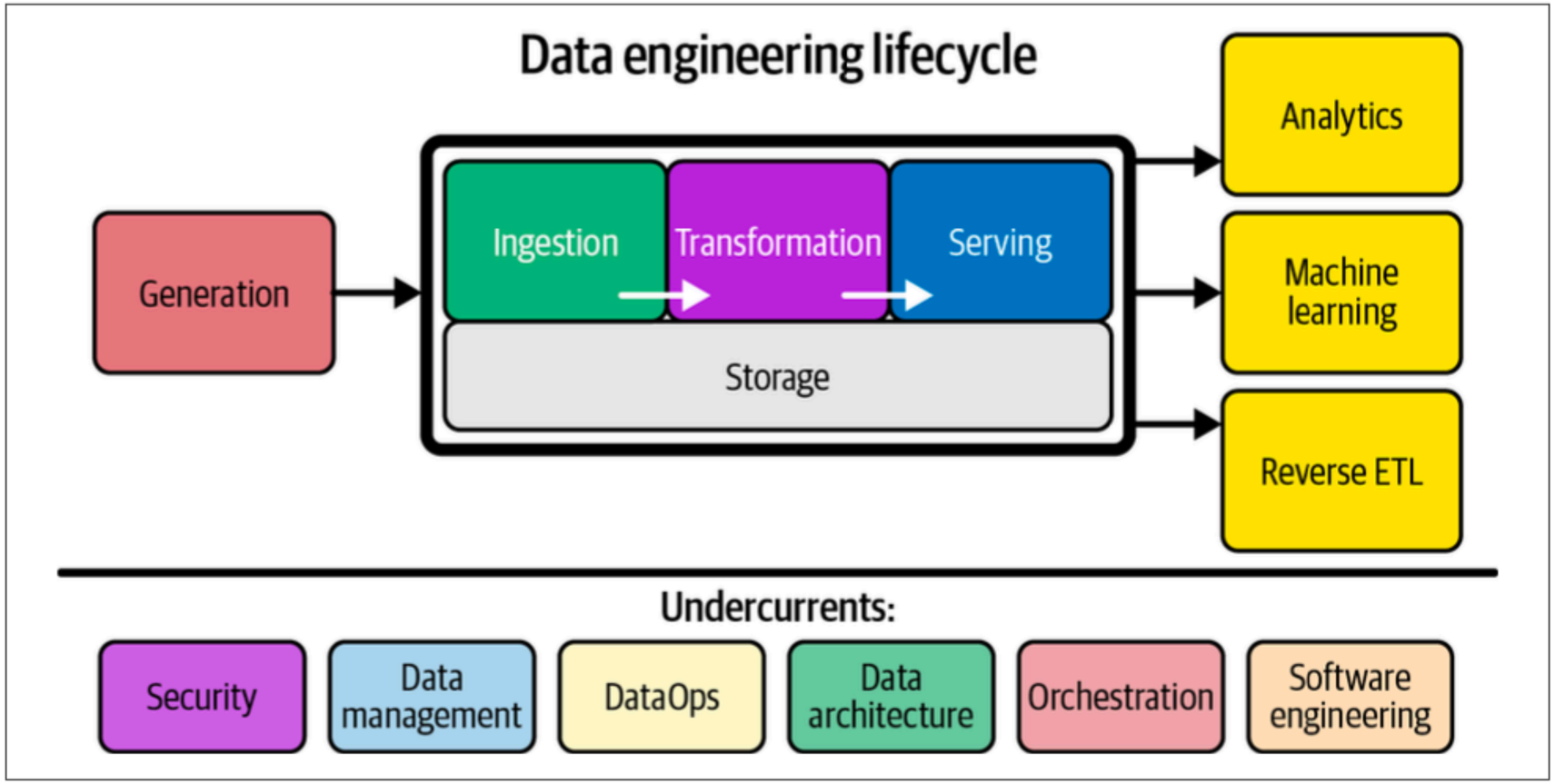
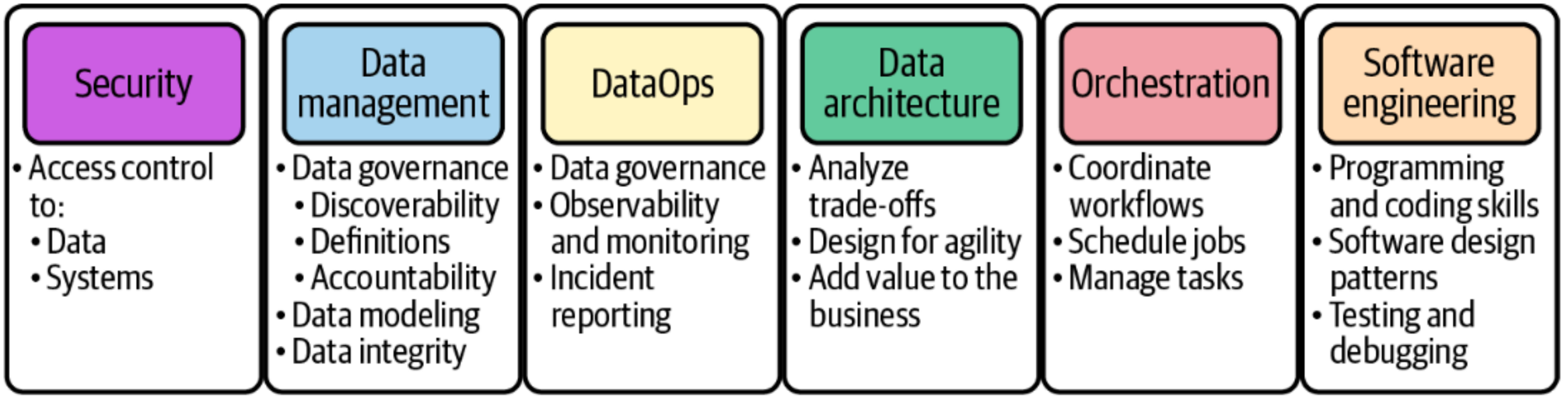


Figure 1-1. The data engineering lifecycle

Undercurrents

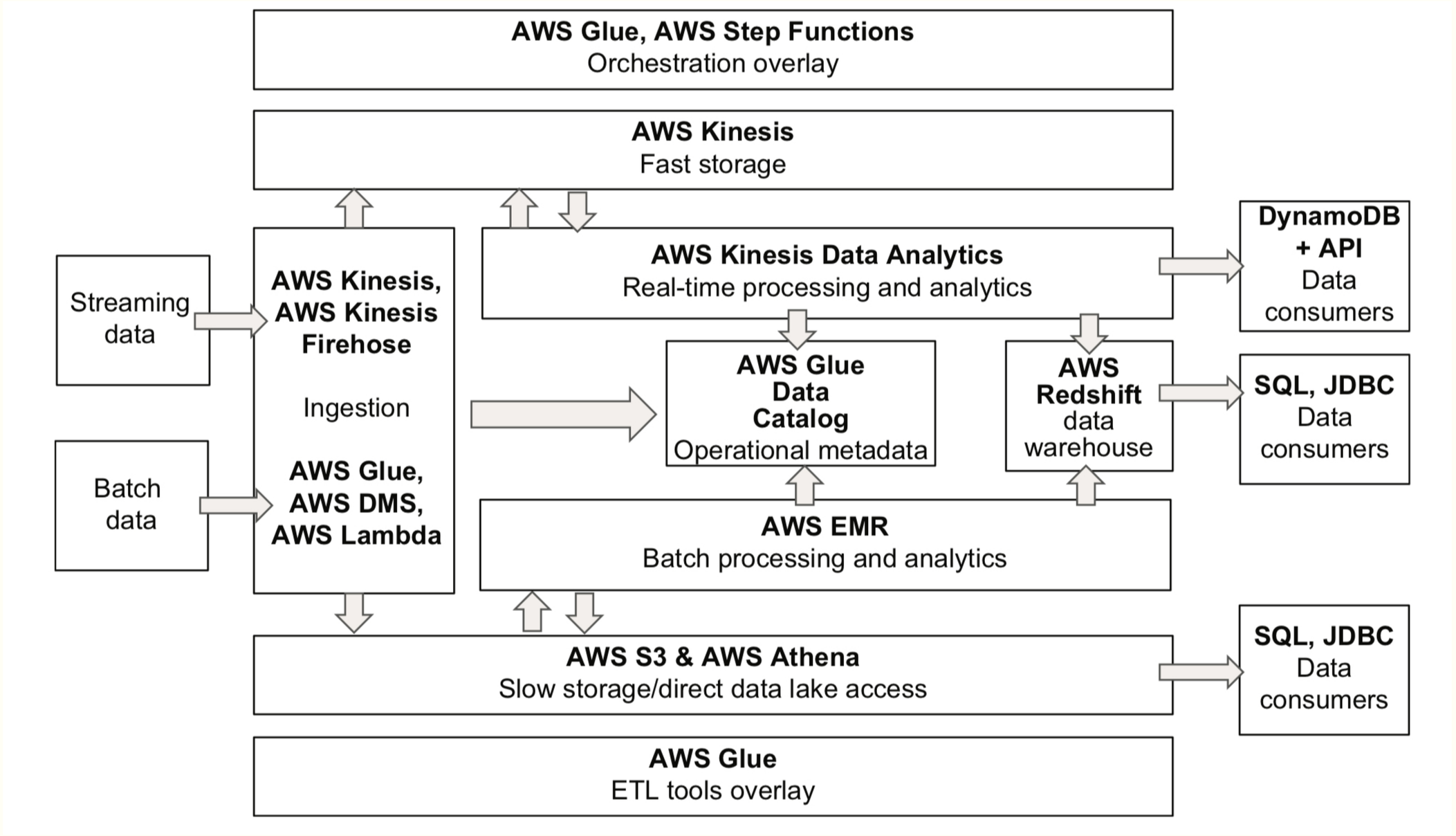


What is left

- We wont cover security and governance: these are huge topics in themselves
- we covered orchestration and data discoverabilty, and we'll cover it some more here
- we've talked about the metadata layer, and how we use it to track pipelines, discover data, and simplify data modeling (including schema on demand).
- we have covered data architecture in our discussion of the move from data lake and warehouse to data lakehouse, and really, the data platform: the combination of this all
- Now we'll focus on the stages of a data pipeline, and the best software development skills to organize and track data flow through them. We'll also focus on model integrity, testing.

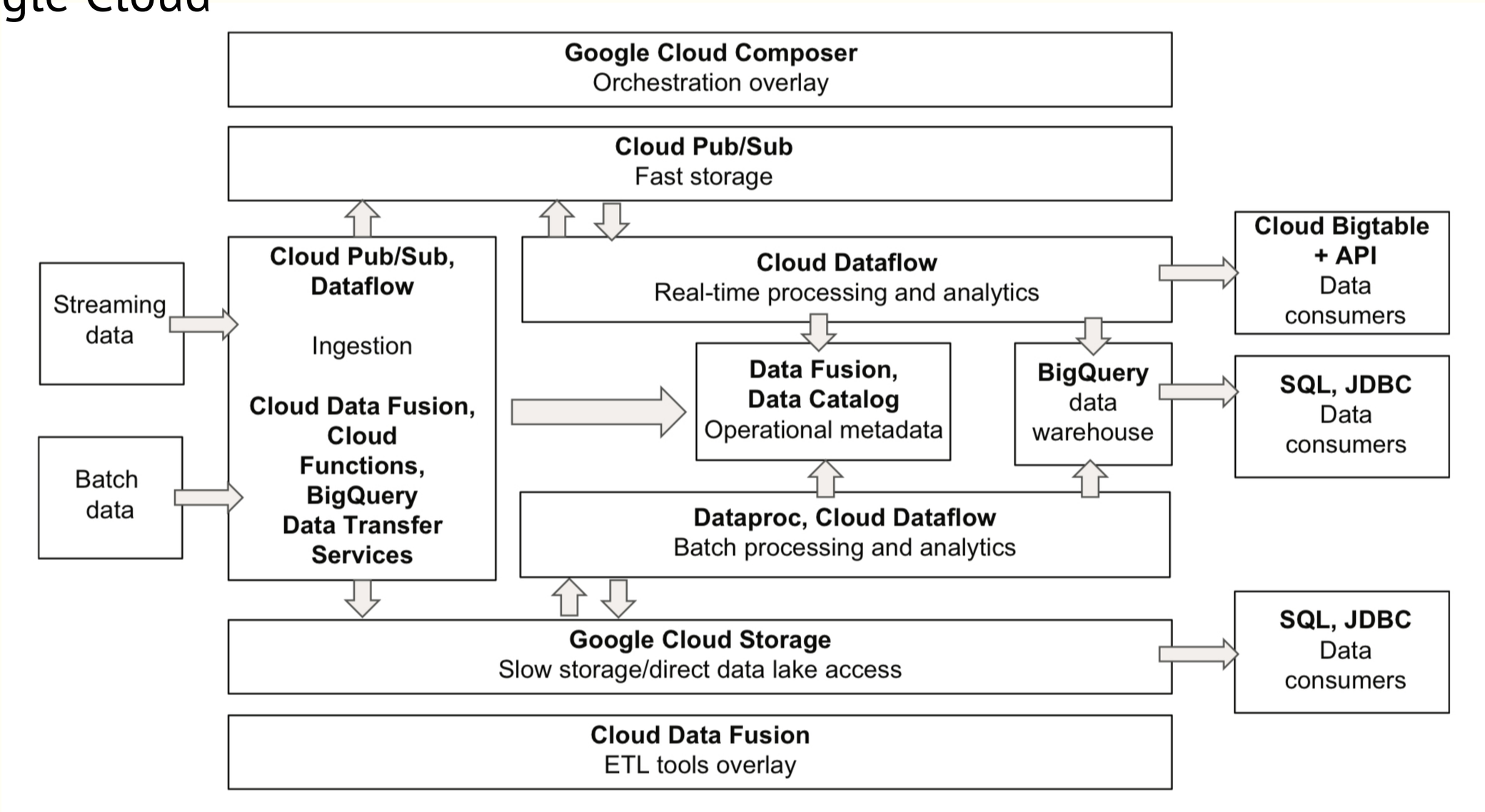
From Lakehouse to Platform

AWS

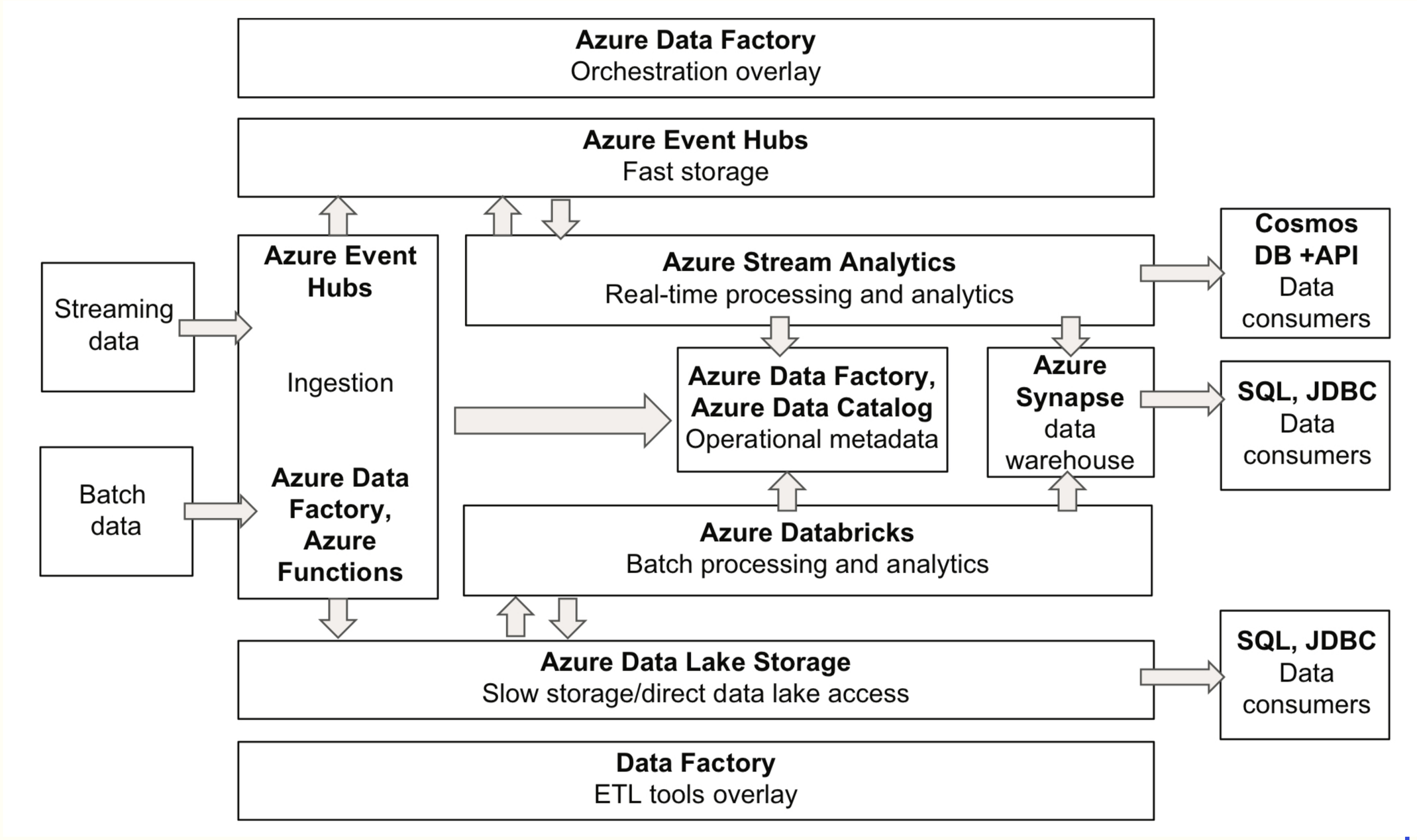


Univ.AI

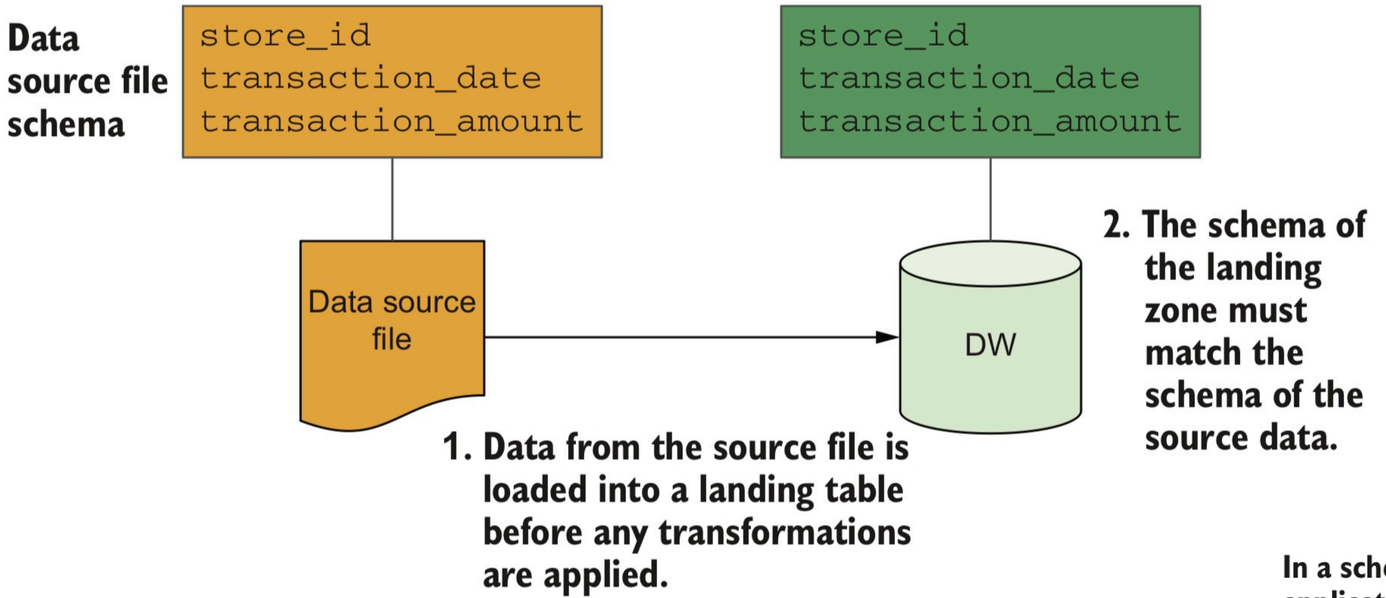
Google Cloud

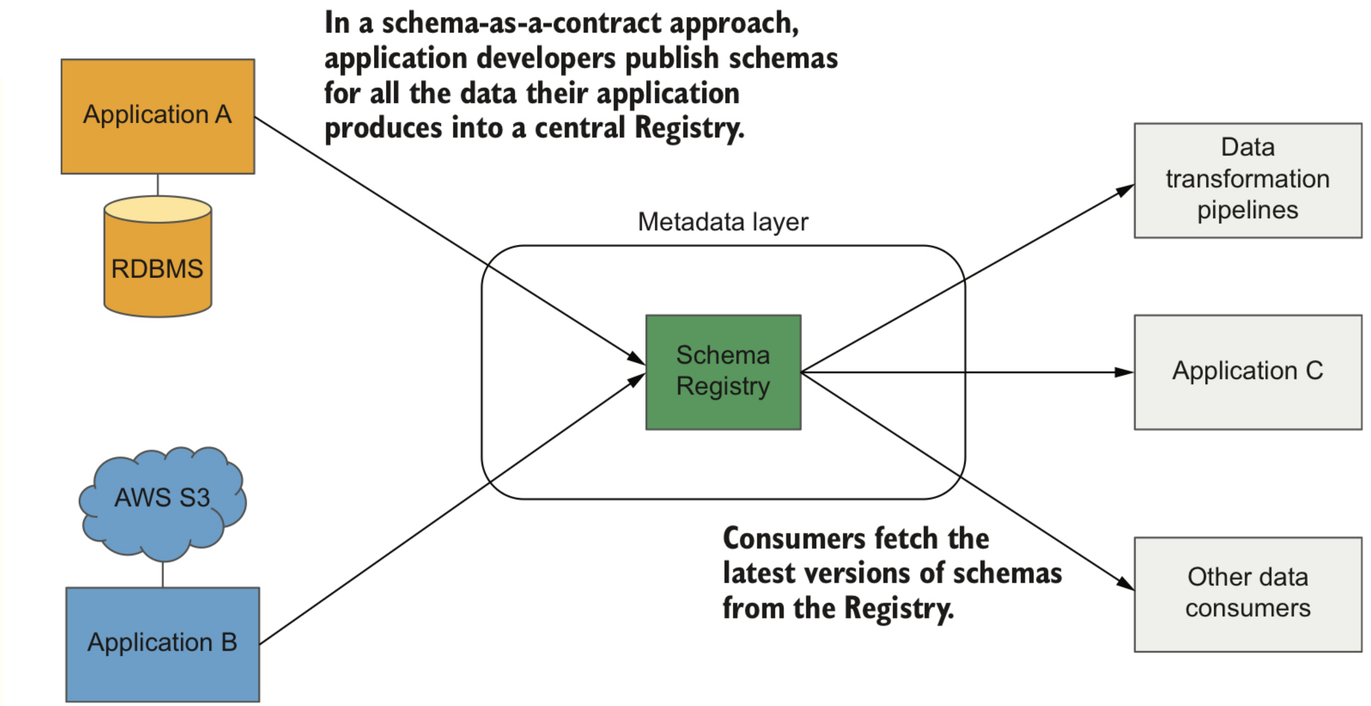


Azure

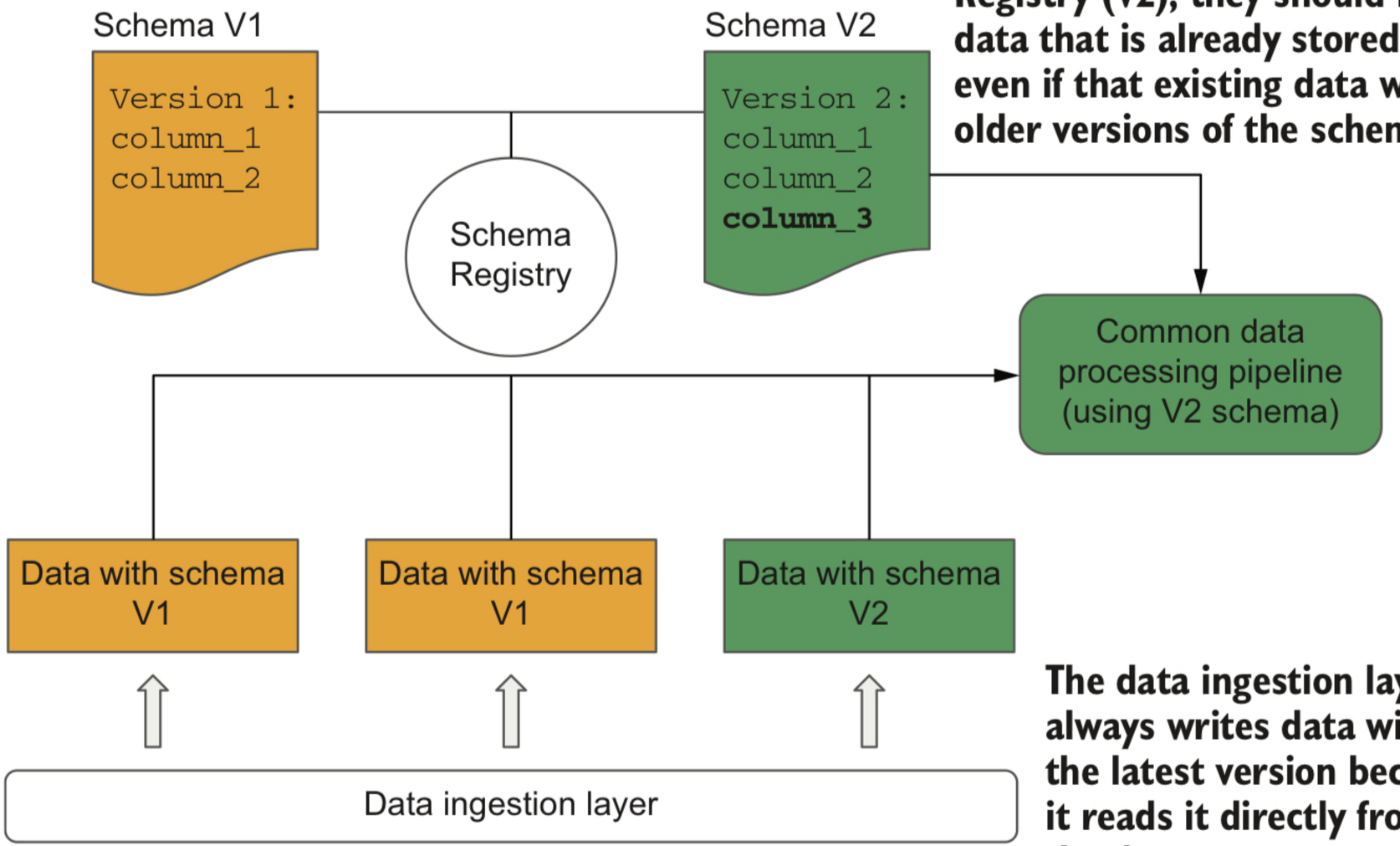


Schema Discovery



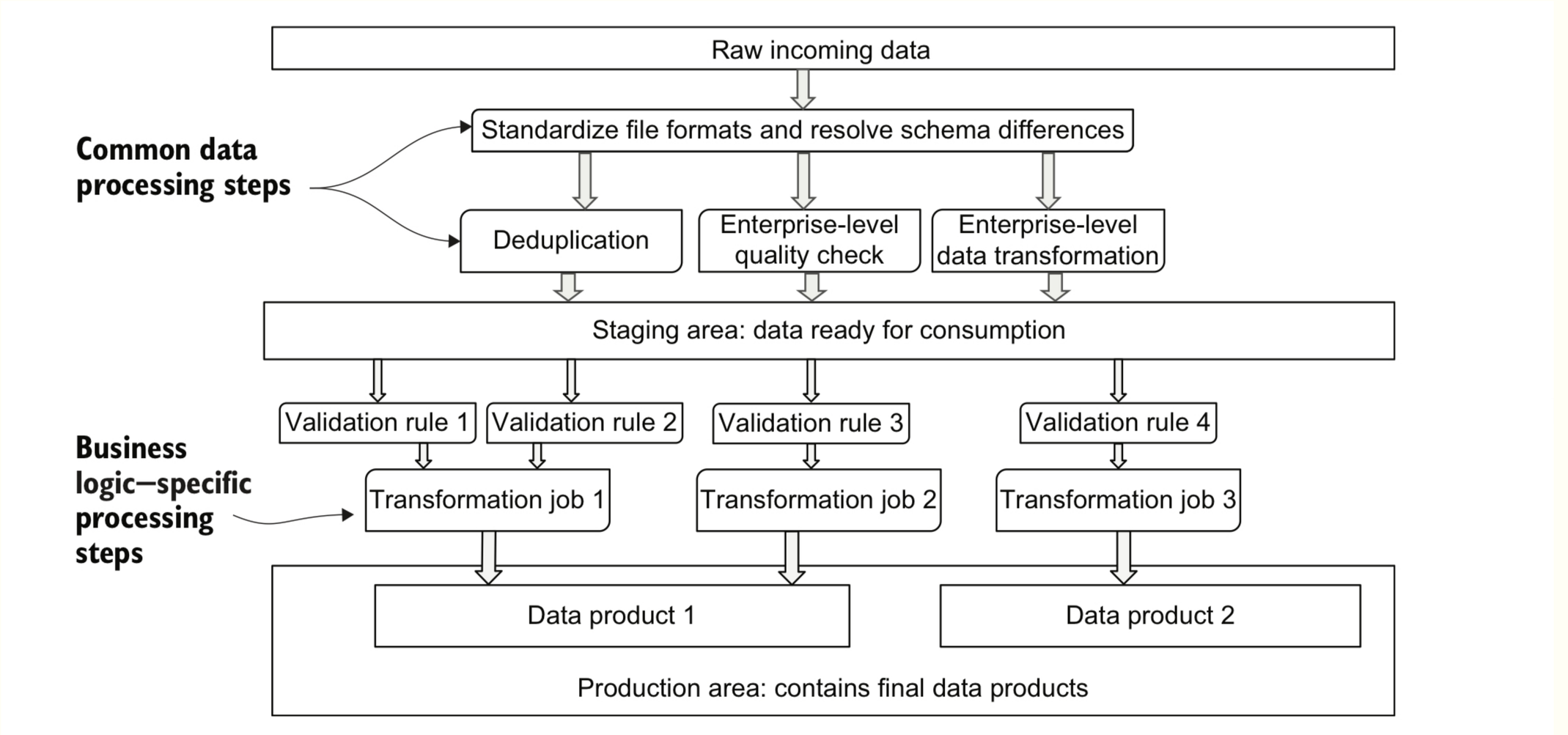


Backward-compatible schema changes mean that if our data transformation pipelines use the latest version of the schema from the Registry (V2), they should be able to read all data that is already stored in the platform even if that existing data was written using older versions of the schema.



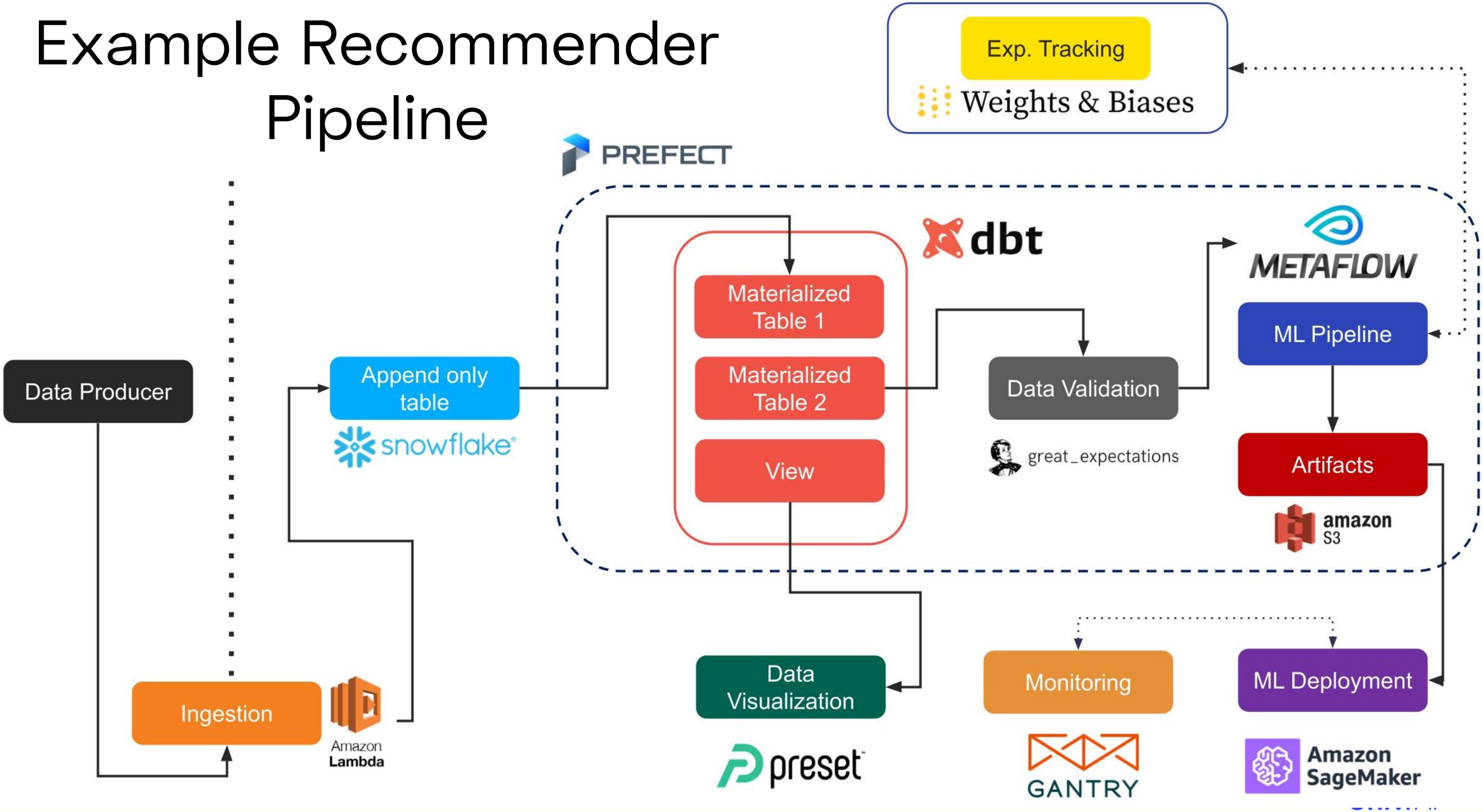
The data ingestion layer always writes data with the latest version because it reads it directly from the data source.

How can we organize stuff like this?



- we need orchestration, which we have discussed
- we also need status reporting, which we have also discussed
- part of the status reporting can be data drift, and data statistics tests, which can lead to pipeline alerts
- we want to organize the code thats used to flow through this pipeline into reusable units, be able to see it all together, and to test it.

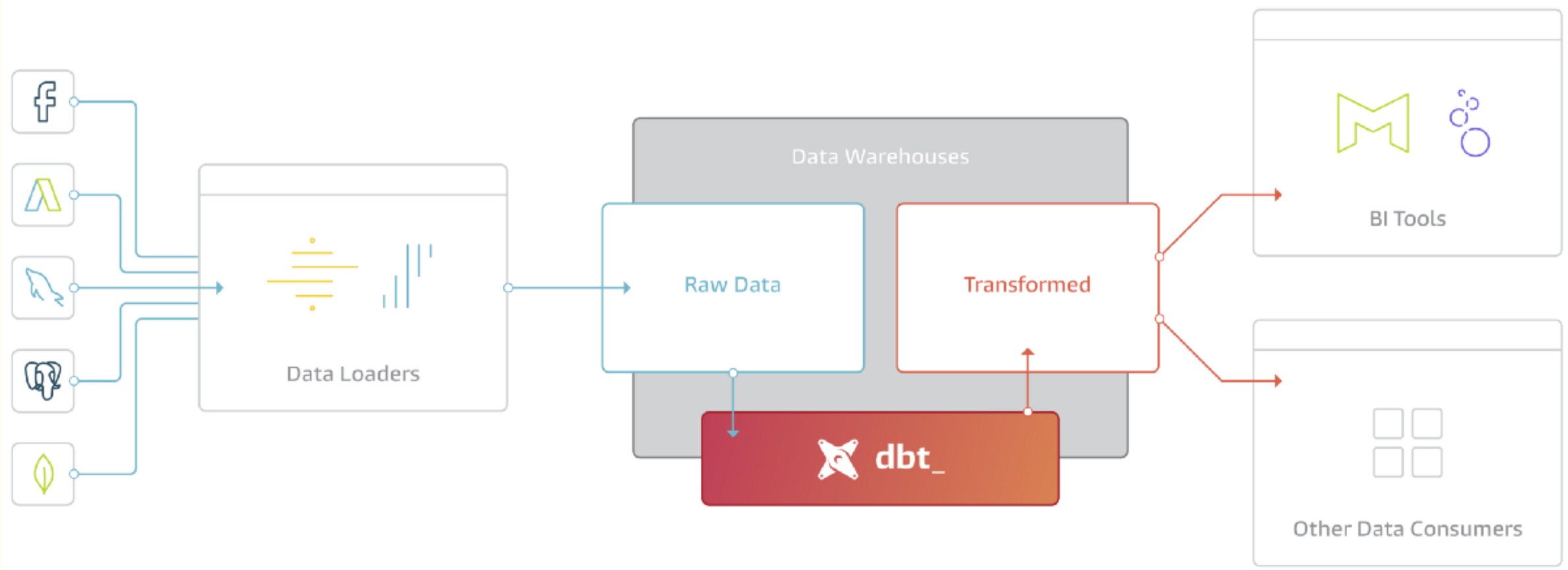
Example Recommender Pipeline



Orchestration, Code, and Transformation: What is dbt?

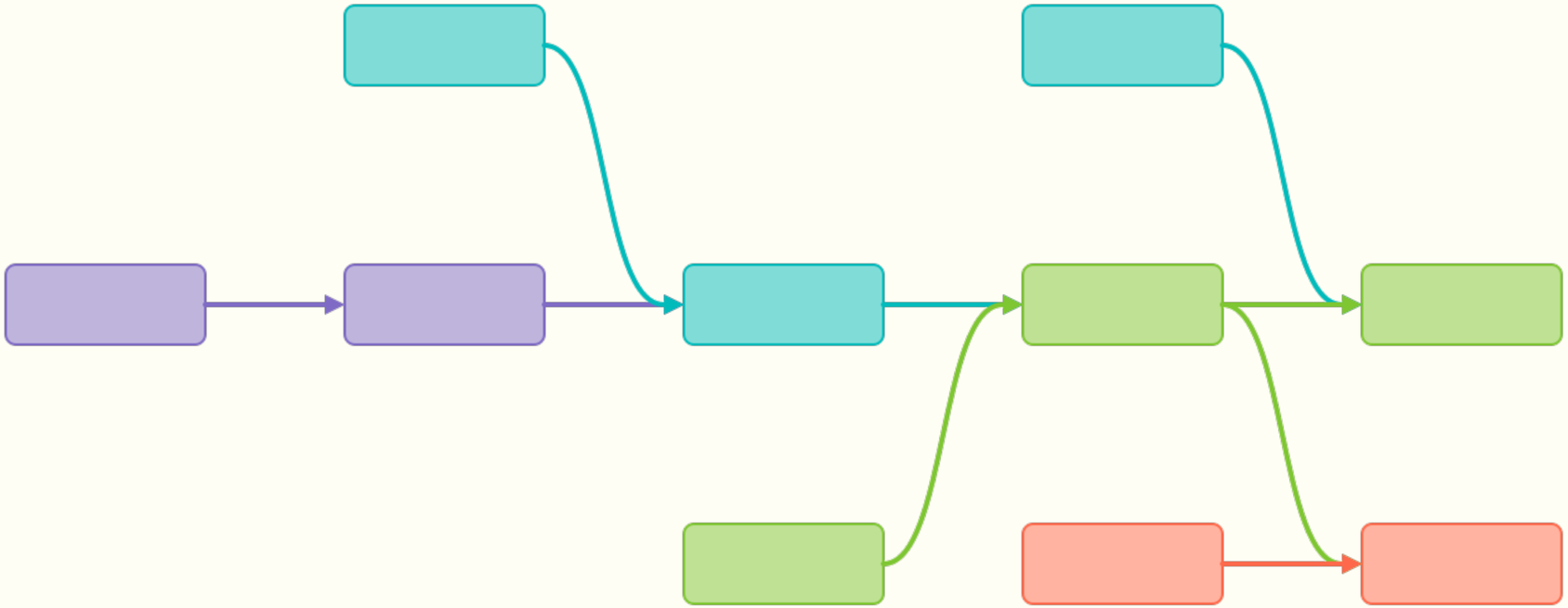
dbt is a transformation workflow that helps you get more work done while producing higher quality results. You can use dbt to modularize and centralize your analytics code, while also providing your data team with guardrails typically found in software engineering workflows. Collaborate on data models, version them, and test and document your queries before safely deploying them to production, with monitoring and visibility.

dbt compiles and runs your analytics code against your data platform, enabling you and your team to collaborate on a single source of truth for metrics, insights, and business definitions. This single source of truth, combined with the ability to define tests for your data, reduces errors when logic changes, and alerts you when issues arise.


Data Warehouses
Raw Data
Transformed
dbt_
Data Loaders
BI Tools
Other Data Consumers

Its a DAG!

- This is a DAG for transformations only
- It will be embedded in a larger dag run by airflow or prefect or dagster or similar cloud software
- The lessons we learn from it apply everywhere though
- Such systematization is very new!



Materialization

model_with_inline_materialization.sql

```
{{
    config(
        materialized='table',
        sort='timestamp',
        dist='user_id'
    )
}}

select *
from ...
```

incremental_model.sql

```
{{
    config(
        materialized='incremental'
    )
}}

select ...
```

snapshots/orders.sql

```
{% snapshot orders_snapshot %}

{{
    config(
        unique_key='id',
        strategy='timestamp',
        updated_at='updated_at',
    )
}}
```

Tests

schema.yml

```yaml
version: 2
models:
  - name: events
    description: '{{ doc("table_events") }}'
    columns:
      - name: event_id
        description: This is a unique identifier for the event
        test:
          - unique
          - not_null
```

tests/assert_payment_amount_is_positive.sql

```sql
select
    order_id,
    sum(amount) as total_amount
from {{ ref('fct_payments' )}}
group by 1
having not(total_amount >= 0)
```

packages.yml

```yaml
packages:
  - package: calogica/dbt_expectations
    version: 0.2.8
```

Docs

source_freshness.yml

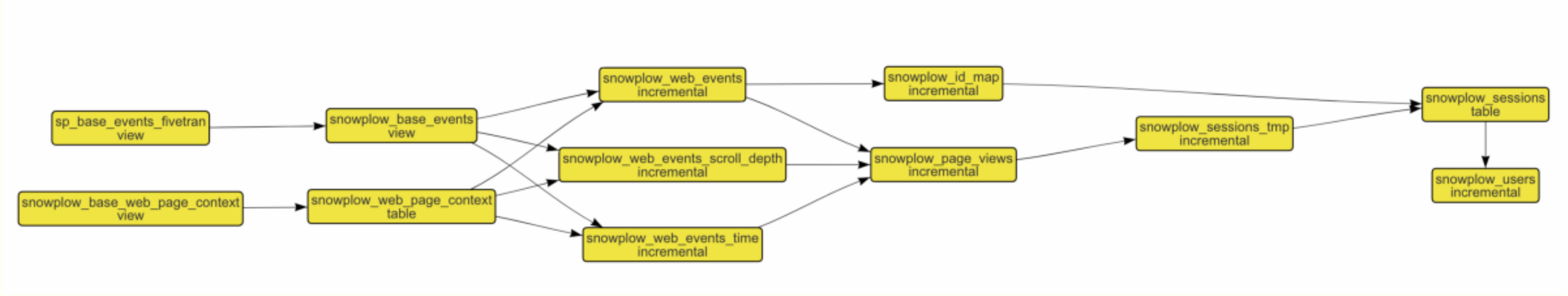
```yaml
sources:
  - name: jaffle_shop
    database: raw
    freshness:
      warn_after: {count: 12, period: hour}
      error_after: {count: 24, period: hour}
    loaded_at_field: _etl_loaded_at
```

exposures.yml

```yaml
exposures:
  - name: weekly_jaffle_metrics
    type: dashboard
    url: https://bi.tool/dashboards/1
    depends_on:
      - ref('fct_orders')
      - ref('dim_customers')
    owner:
      name: Claire from Data
      email: data@jaffleshop.com
```

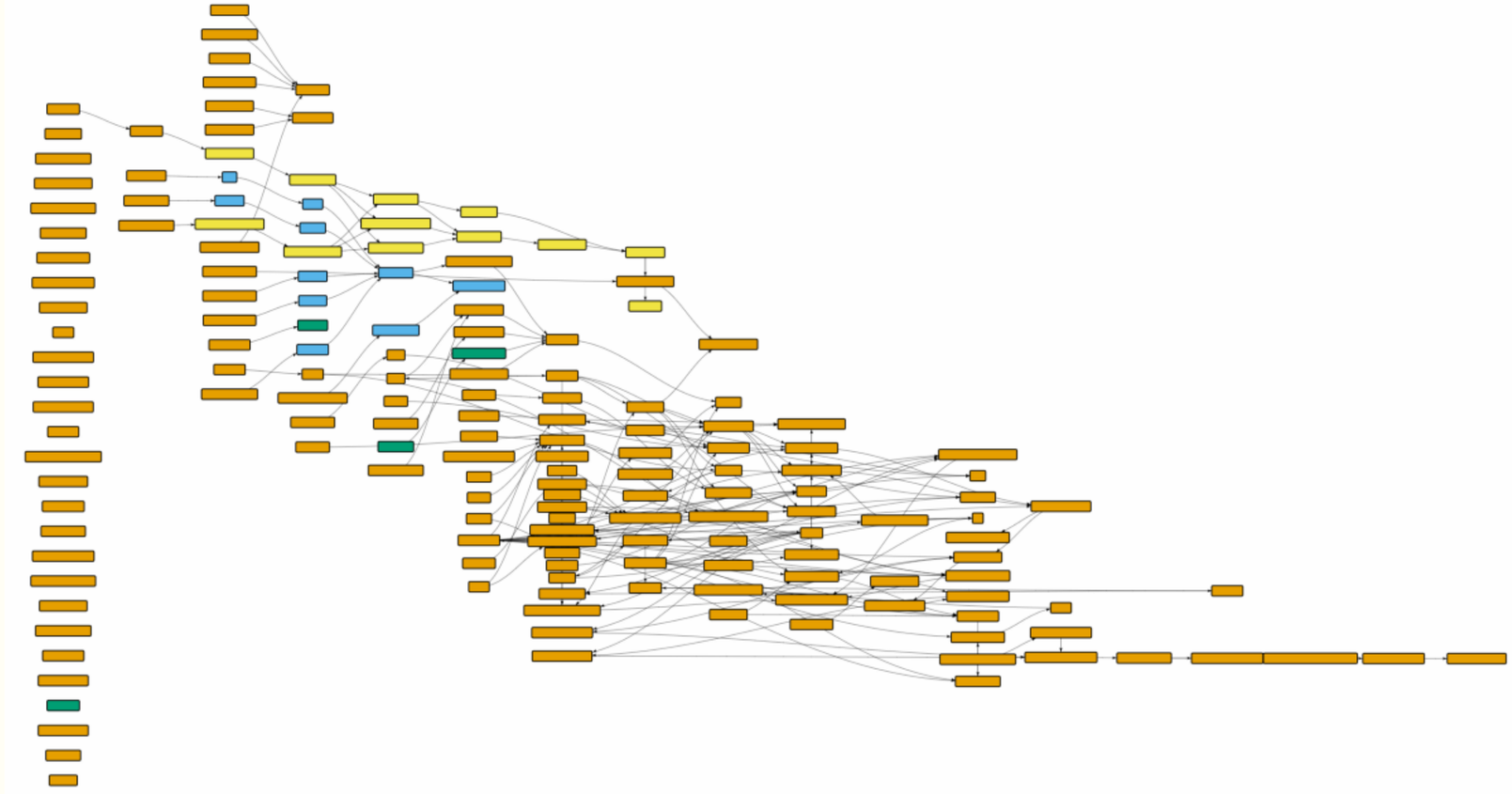
Useful Features

- dbt constructs views/materialized views/tables automatically as models from sql files
- you can add integrity statements and tests
- using the ref macro, you can refer to a model abstractly so that the implementation is resolved at run time. This enables different development and production flows, even using different databases. Ditto the source macro
- For example you might test against duckdb locally, and move to spark/trino/dremio on s3 or iceberg or just bigquery, snowflake, or redshift.
- dbt operates “inside the warehouse, for the transformation layer”, but using a multi-sql tool like dremio or spark, you can use it everywhere. It is not the E or the L layer, but in this way it can be a multi database T layer.



Snowplow incrementally sessionized DAG

Ecommerce DAG



Example Layout

- We start with the staging, where we minimally change stuff coming in from other systems (over something like meltano/airbyte/spark/etc)
- We then make intermediate transformations
- finally groups can (or you can for them) make your own transformations in data marts: you will remember this architecture from before.
- inputs or outputs could be in warehouse, lake, or lakehouse
- read: https://docs.getdbt.com/guides/best-practices/how-we-structure/1-guide-overview

```
├── models
│   ├── intermediate
│   │   └── finance
│   │       ├── _int_finance__models.yml
│   │       └── int_payments_pivoted_to_orders.sql
│   ├── marts
│   │   ├── finance
│   │   │   ├── _finance__models.yml
│   │   │   ├── orders.sql
│   │   │   └── payments.sql
│   │   └── marketing
│   │       ├── _marketing__models.yml
│   │       └── customers.sql
│   ├── staging
│   │   ├── jaffle_shop
│   │   │   ├── _jaffle_shop__docs.md
│   │   │   ├── _jaffle_shop__models.yml
│   │   │   ├── _jaffle_shop__sources.yml
│   │   │   ├── base
│   │   │   │   ├── base_jaffle_shop__customers.sql
│   │   │   │   └── base_jaffle_shop__deleted_customers.sql
│   │   │   ├── stg_jaffle_shop__customers.sql
│   │   │   └── stg_jaffle_shop__orders.sql
│   │   └── stripe
│   │       ├── _stripe__models.yml
│   │       ├── _stripe__sources.yml
│   │       └── stg_stripe__payments.sql
│   └── utilities
│       └── all_dates.sql
```

Staging

```sql
-- stg_stripe__payments.sql

with

source as (

    select * from {{ source('stripe','payment') }}

),

renamed as (

    select
        -- ids
        id as payment_id,
        orderid as order_id,

        -- strings
        paymentmethod as payment_method,
        case
            when payment_method in ('stripe', 'paypal', 'credit_card', 'gift_card') then 'credit'
            else 'cash'
        end as payment_type,
        status,

        -- numerics
        amount as amount_cents,
        amount / 100.0 as amount,

        -- booleans
        case
            when status = 'successful' then true
            else false
        end as is_completed_payment,

        -- dates
        date_trunc('day', created) as created_date,

        -- timestamps
        created::timestamp_ltz as created_at

    from source

)

select * from renamed
```

```sql
-- base_jaffle_shop__customers.sql

with

source as (

    select * from {{ source('jaffle_shop','customers') }}

),

customers as (

    select
        id as customer_id,
        first_name,
        last_name

    from source

)

select * from customers
```

```sql
-- stg_jaffle_shop__customers.sql

with

customers as (

    select * from {{ ref('base_jaffle_shop__customers') }}

),

deleted_customers as (

    select * from {{ ref('base_jaffle_shop__deleted_customers') }}

),

join_and_mark_deleted_customers as (

    select
        customers.*,
        case
            when deleted_customers.deleted_at is not null then true
            else false
        end as is_deleted

    from customers

    left join deleted_customers on customers.customer_id = deleted_customers.customer_id

)

select * from join_and_mark_deleted_customers
```

.AI

```sql
-- int_payments_pivoted_to_orders.sql

{%- set payment_methods = ['bank_transfer','credit_card','coupon','gift_card'] -%}

with

payments as (

   select * from {{ ref('stg_stripe__payments') }}

),

pivot_and_aggregate_payments_to_order_grain as (

   select
      order_id,
      {% for payment_method in payment_methods -%}

         sum(
            case
               when payment_method = '{{ payment_method }}' and
                    status = 'success'
               then amount
               else 0
            end
         ) as {{ payment_method }}_amount,

      {%- endfor %}
      sum(case when status = 'success' then amount end) as total_amount

   from payments

   group by 1

)

select * from pivot_and_aggregate_payments_to_order_grain
```

Intermediate Transforms

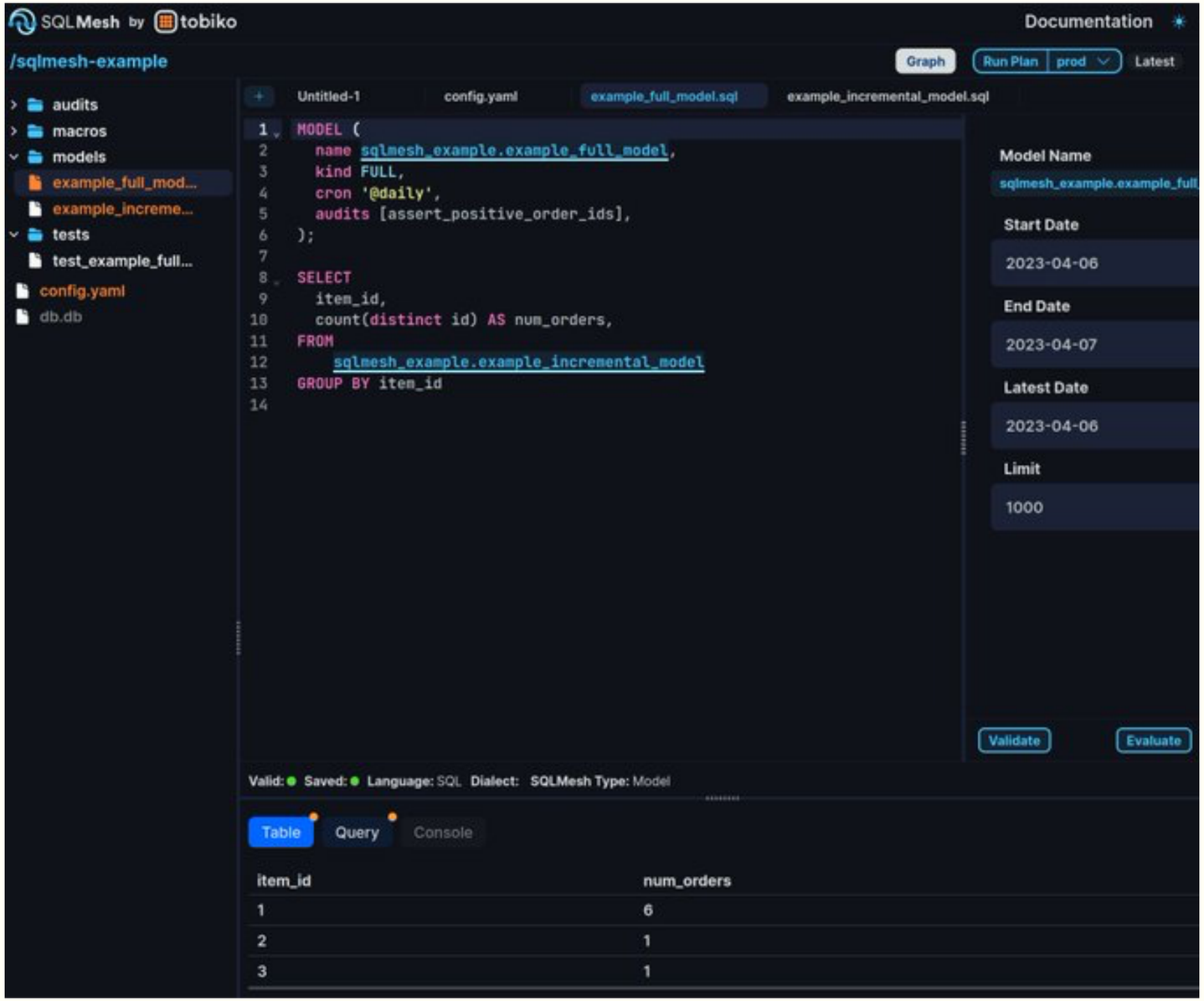
- create tables or views that are useful to everyone in different groups, start with views, then move to tables, and then move to incremental tables
- great place to do course grained kimball tables
- Using templating can make your SQL easier to read and much more DRY. You could use this idea in your E and T stages as well.

```sql
-- customers.sql

with

customers as (

    select * from {{ ref('stg_jaffle_shop__customers')}}

),

orders as (

    select * from {{ ref('orders')}}

),

customer_orders as (

    select
        customer_id,
        min(order_date) as first_order_date,
        max(order_date) as most_recent_order_date,
        count(order_id) as number_of_orders,
        sum(amount) as lifetime_value

    from orders

    group by 1

),

customers_and_customer_orders_joined as (

    select
        customers.customer_id,
        customers.first_name,
        customers.last_name,
        customer_orders.first_order_date,
        customer_orders.most_recent_order_date,
        coalesce(customer_orders.number_of_orders, 0) as number_of_orders,
        customer_orders.lifetime_value

    from customers

    left join customer_orders on customers.customer_id = customer_orders.customer_id

)

select * from customers_and_customer_orders_joined
```

Vertical Marts

- materialize as tables or incremental models
- unlike kimball, this is a great plave to keep your stuff as wide denormalized tables as it will help your BI software
- After marts, what? depends on what you are doing with the data. You might use BI to make a report
- Consider adding a metrics or semantic layer so that more stuff becomes as self-serve as possible

Also check out sqlmesh



Entire Process Considerations

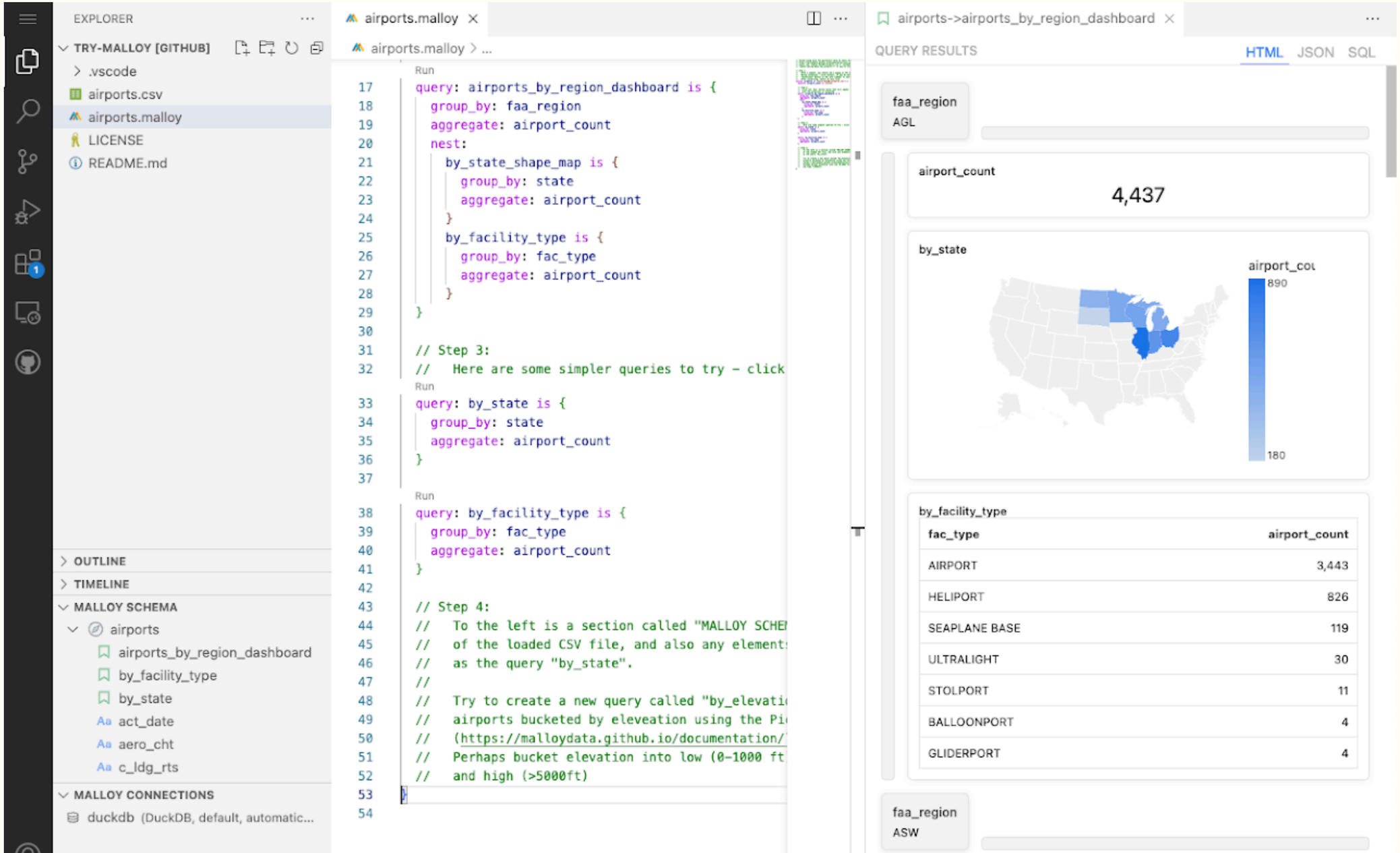
- You need to decide what is your platform: perhaps warehouse for reports/BI, lakehouse for lower latency BI, data for ML, perhaps lake for raw data and even some ML products, a catalog for a transactional lakehouse, schema discovery, and a model registry/dataset registry

- How do you want to move data through your system: full refreshes, log structured append only tables, incremental ingestion, etc. See https://tobikodata.com/correctly-loading-incremental-data-at-scale.html and https://discourse.getdbt.com/t/on-the-limits-of-incrementality/303 for some considerations

- Your project should have an organized structure and be well tested. This is not just in the dbt parts, but the E and L parts as well.

- We'll revisit all of these in the lab, using a complete open-source stack.

Post warehouse/mart: metrics

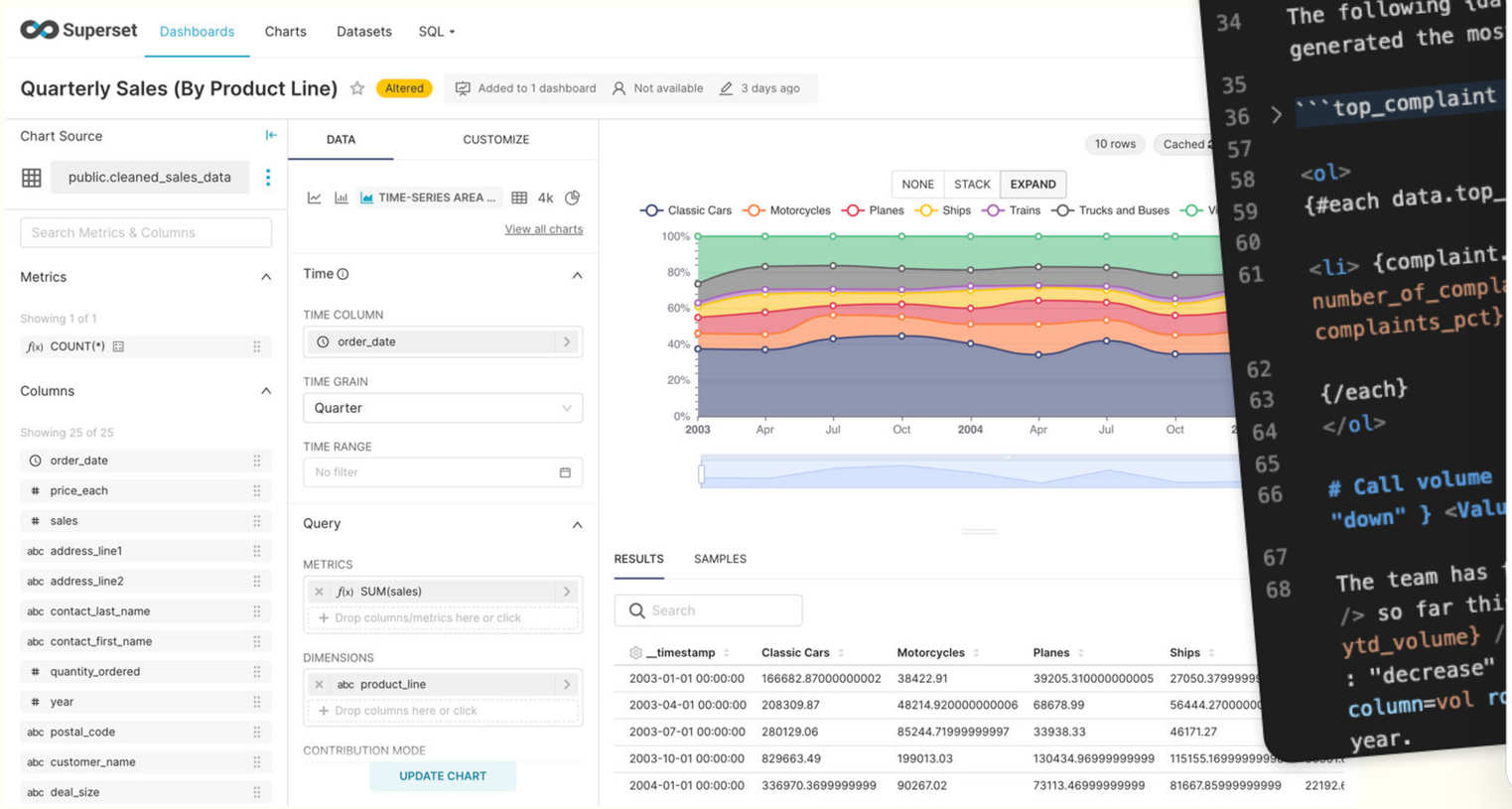


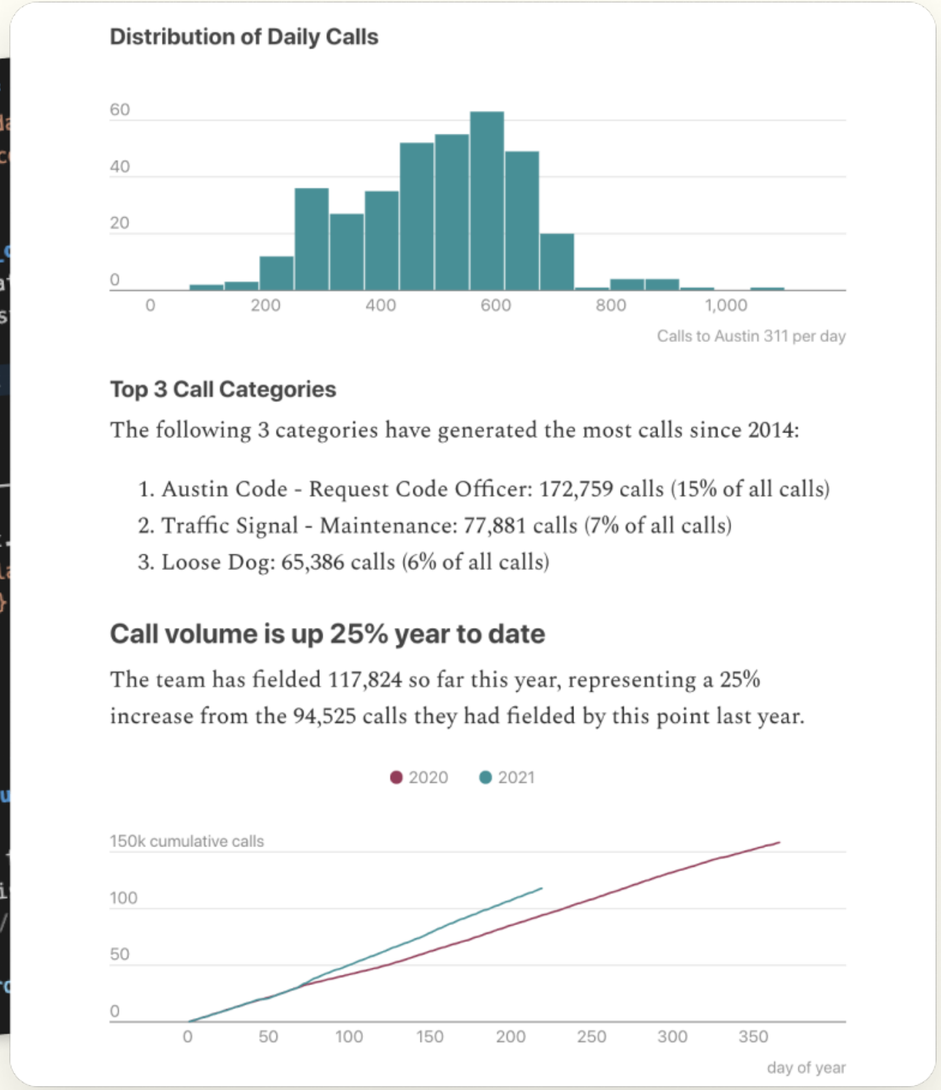
Metrics layer: malloy, ibis, dbt semantic layer

Univ.AI

Post warehouse/mart: reports/dashboard

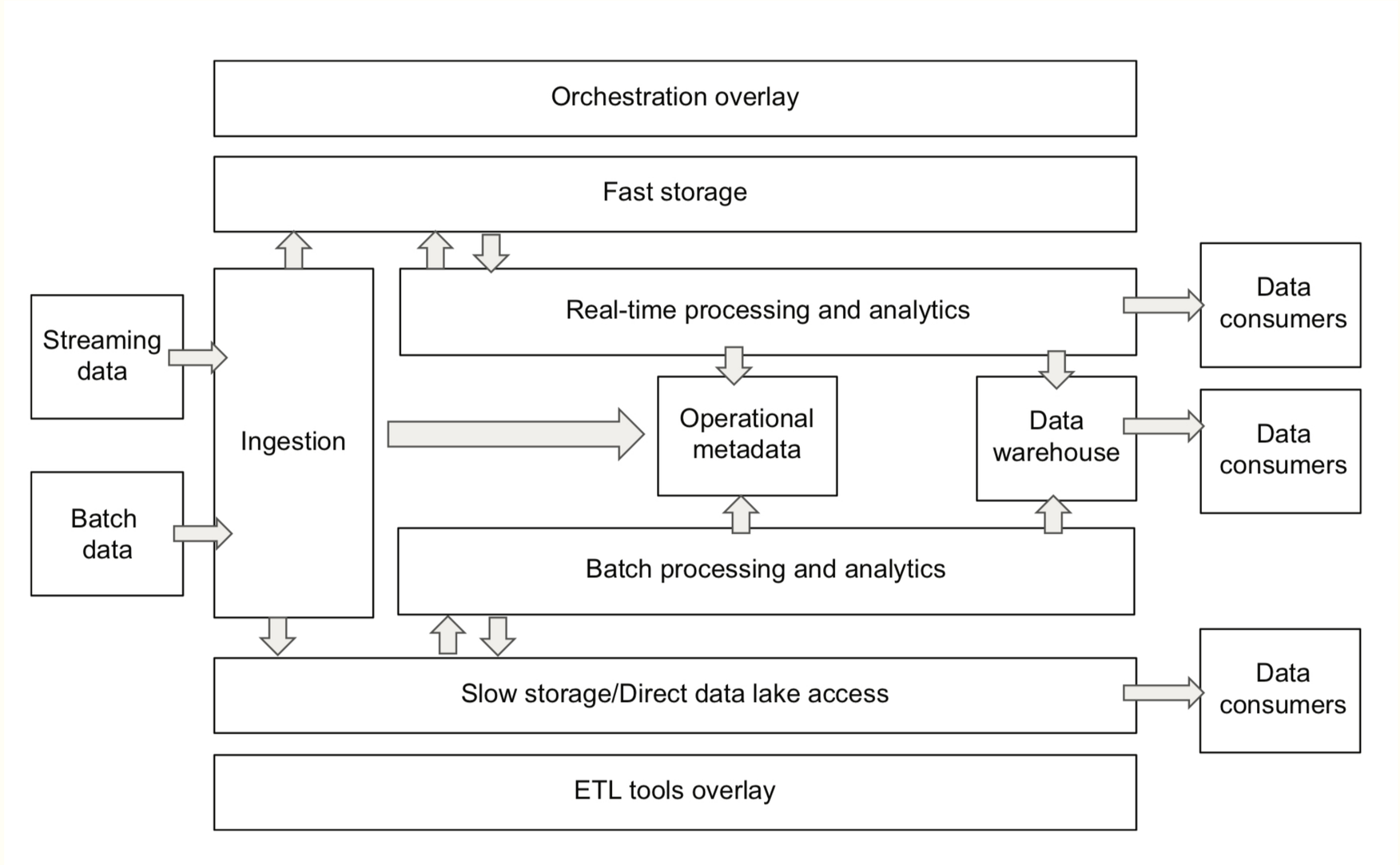
Presentation layer: preset/superset, evidence.dev





Univ.AI

Infra for the holy grail



O'REILLY®

Data Algorithms with Spark

Recipes and Design Patterns for Scaling Up Using PySpark

Hadoop

Designing Cloud Data Platforms

Danil Zburivsky
Lynda Partner

Ergest Xheblati

Minimum Viable

SQL Patterns

Hands on Design Patterns and Best Practices with SQL

Univ.AI

Fin (not really)