NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

NTU Academy for Professional and Continuing Education

(SCTP) Advanced Professional Certificate

Data Science and AI

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

2.7 Testing and Data Orchestration

Module Overview

2.1 Introduction to Big Data and Data Engineering

2.2 Data Architecture

2.3 Data Encoding and Data Flow

2.4 Data Extraction and Web Scraping

2.5 Data Warehouse

2.6 Data Pipelines and Orchestration

2.7 Data Orchestration and Testing

2.8 Out of Core/Memory Processing

2.9 Big Data Ecosystem and Batch Processing

2.10 Event Streaming and Stream Processing

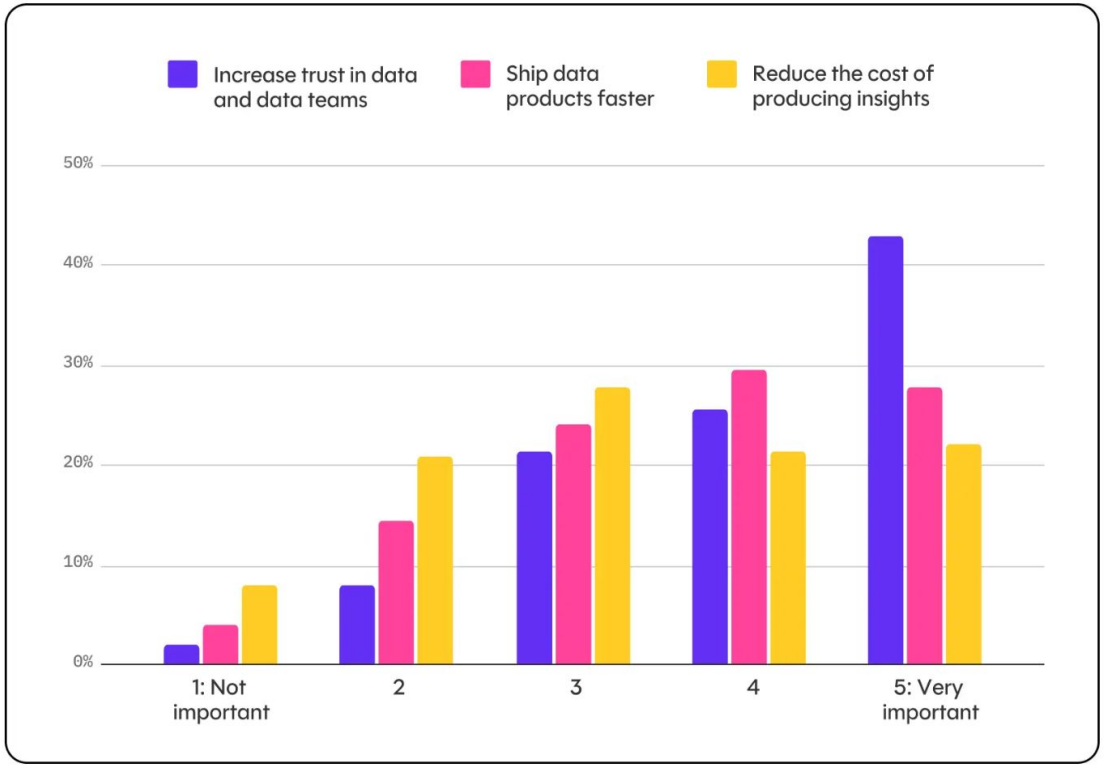
Lesson Objectives

- Understand Orchestration With Dagster
- Understand Data Quality and Testing
  - Set up and run data quality tests using Great Expectations.
  - Set up and run data quality tests using Dbt packages.

Data Tools Survey

DBT - 2025 State of Analytics Engineering Report

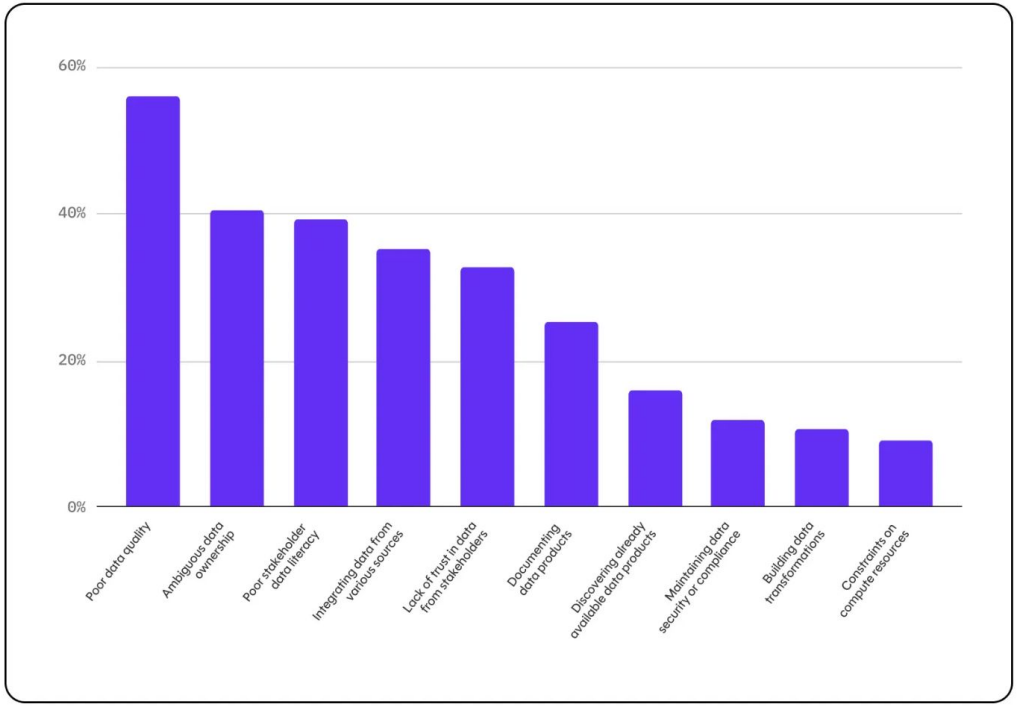
Please rate the importance of the following objectives for your organization in 2025



https://www.getdbt.com/resources/state-of-analytics-engineering-2025#data-team-budgets-and-headcount

DBT - 2025 State of Analytics Engineering Report

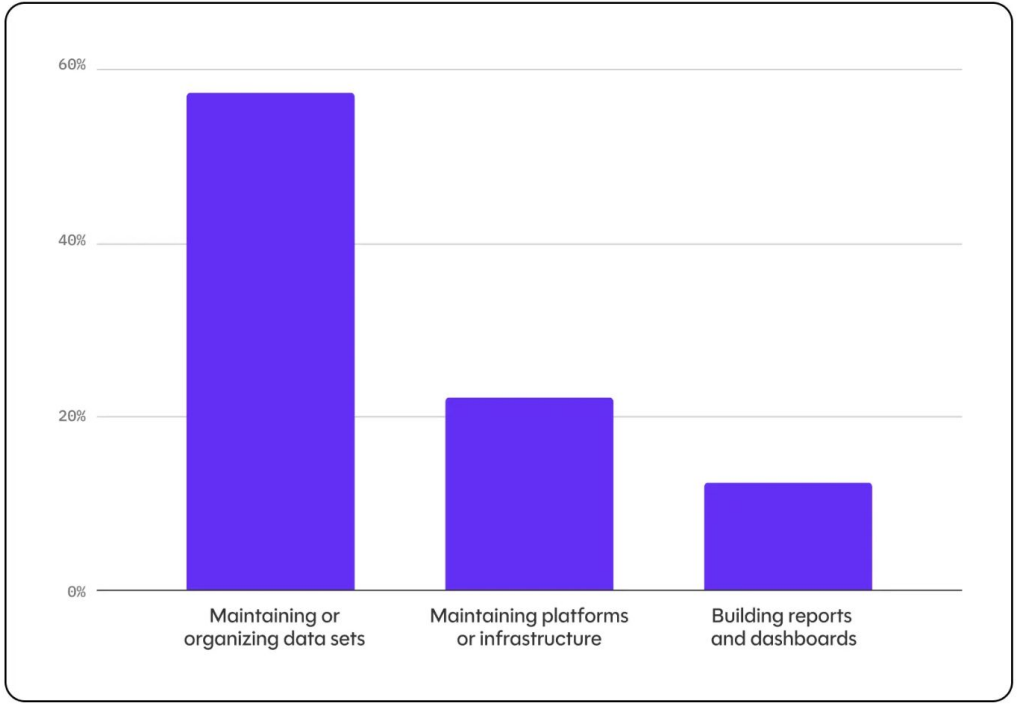
What do you find most challenging while preparing data for analysis? Select up to three.



https://www.getdbt.com/resources/state-of-analytics-engineering-2025#data-team-budgets-and-headcount

DBT - 2025 State of Analytics Engineering Report

Which of the following best describes how you spend most of your time?

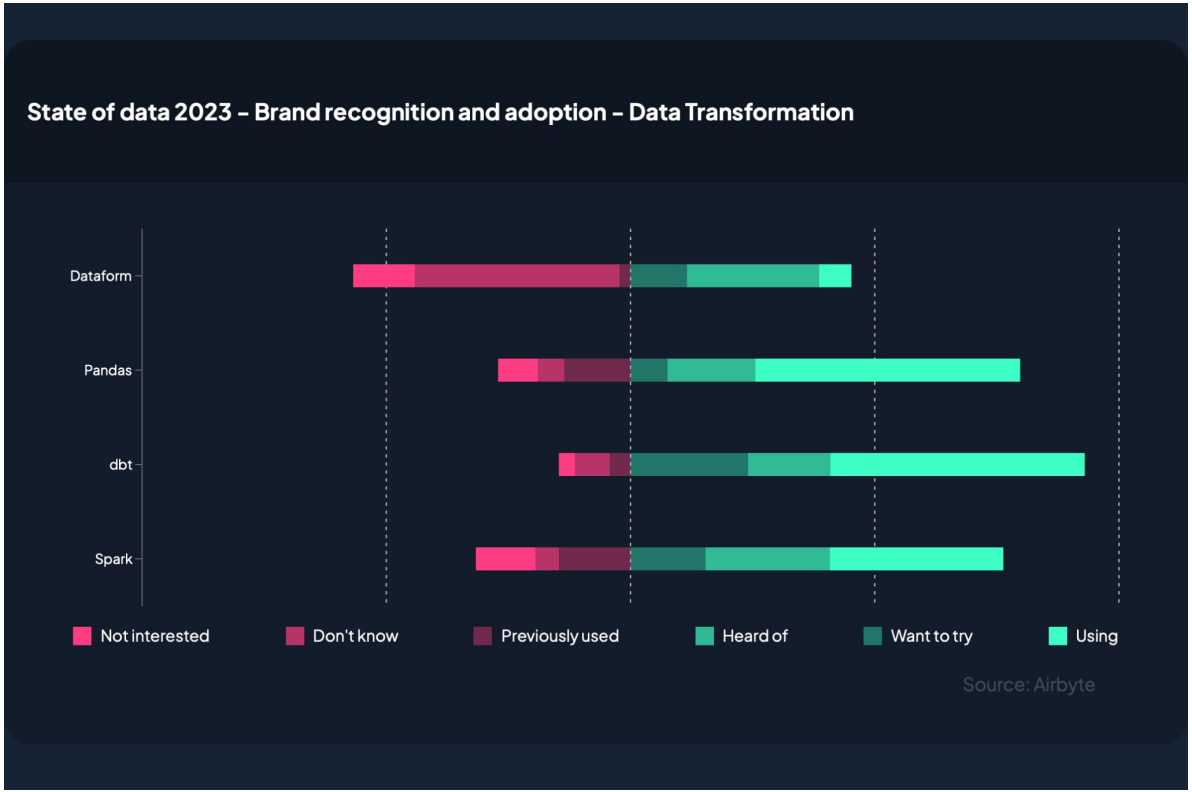


https://www.getdbt.com/resources/state-of-analytics-engineering-2025#data-team-budgets-and-headcount

2023 State of Data Survey - Data Tools (800+ respondents)

https://state-of-data.com/data-tooling-insights--data-ingestion

2023 State of Data Survey - Data Tools (800+ respondents)



https://state-of-data.com/data-tooling-insights--data-transformation

2023 State of Data Survey - Data Tools (800+ respondents)

https://state-of-data.com/data-tooling-insights--data-warehouses

Orchestration

2023 State of Data Survey - Data Tools (800+ respondents)

https://state-of-data.com/data-tooling-insights--data-orchestration

DAGSTER 101

THE CORE MODEL

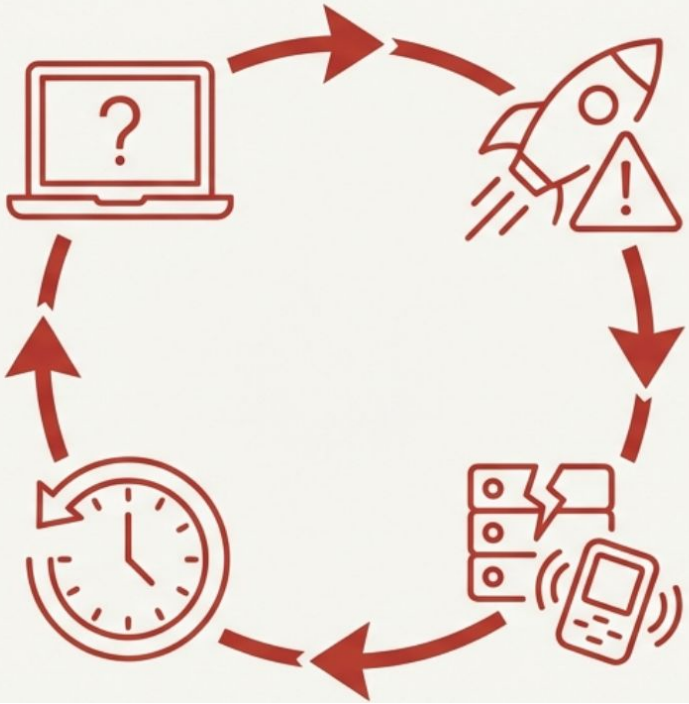
https://youtu.be/k-iQNrtKhjo

The Data Engineer's Vicious Cycle

Many engineering teams are struggling. They spend too much time babysitting production and don't have a chance to build new things.

Can't Test Locally

Code is written blind, without realistic feedback loops.



Push to Production

The only way to see if code works is to ship it.

Production Failures

Untested code leads to outages, paging on-call engineers at night.

Slows New Work

Constant firefighting and interruptions prevent paying down the technical debt that causes the problems.

The Root Cause: Imperative Tasks vs. Declarative Assets

Traditional orchestrators are built on imperative tasks. They define a sequence of commands to run, but they have no awareness of the data assets those tasks produce. You are defining *how to run*, not *what to build*.

Imperative Tasks (e.g., Airflow)

A list of instructions to execute.

Declarative Assets (Dagster)

A map of the data that should exist.

The Core Abstraction: The Software-Defined Asset

An asset is a Python function that produces a persistent object—a table, a file, a model. By decorating a function with `@asset`, you tell Dagster that this function is responsible for creating a specific piece of data. Dagster infers dependencies from the function's parameters.

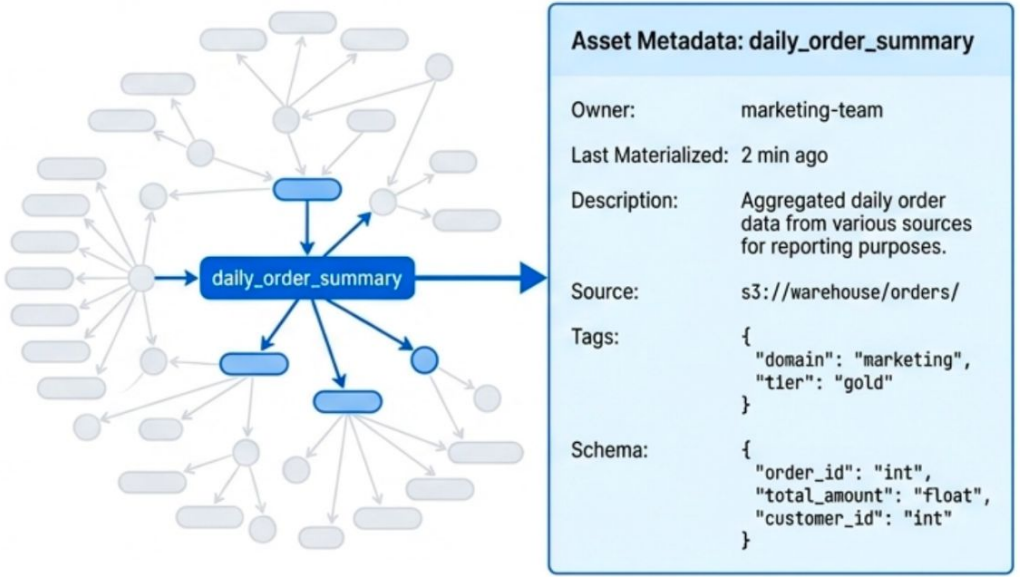
```python
@asset
def github_stargazers():
    # ... fetches raw data from API
    return raw_data

@asset
def github_stargazers_by_week(github_stargazers):
    # ... aggregates raw_data into a weekly summary
    return weekly_summary_df
```

Assets Provide a Unified View of Your Data Platform

When your orchestrator understands assets, your entire data platform becomes observable and manageable. You can immediately see the state of any asset, trace its lineage, and understand its dependencies.

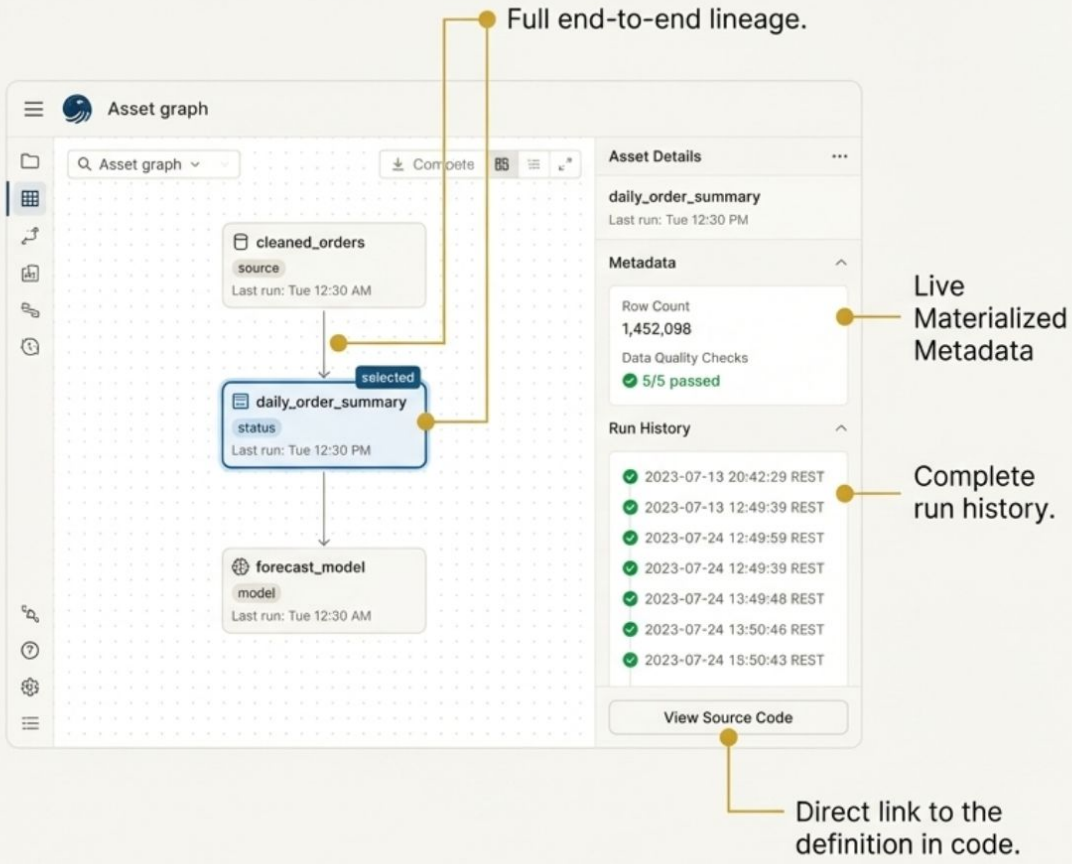
- **Clear Data Lineage**: Instantly understand how an asset was produced, making debugging trivial.
- **Stakeholder Self-Service:** Answer questions like "When was this table last updated?" directly in the UI.
- **Unambiguous Documentation:** The asset graph serves as a living, always-accurate map of your data.
- **Rich, Searchable Metadata:** Attach ownership, descriptions, and validation rules to every asset.



Answer “Why is this data wrong?” in Seconds

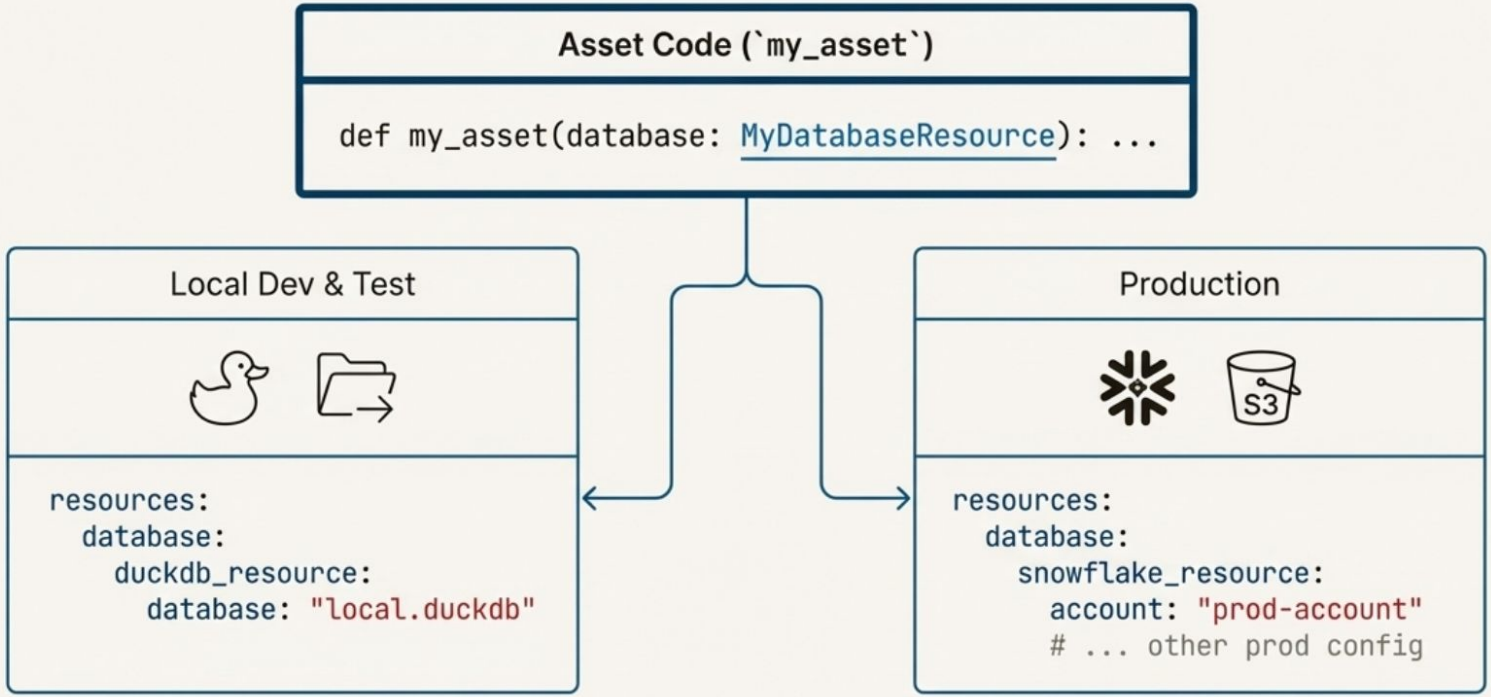
Because Dagster is asset-aware, it builds a complete, observable catalog of your data ecosystem. Answer stakeholder questions about freshness, quality, and origin without spelunking through task logs.

“An executive has a question about the `daily_order_summary`. You can see its entire history, the code that generated it, and its dependencies in one place.”



Swap Your Stack, Not Your Code

Dagster decouples your business logic from I/O and external services using **Resources** and **IO Managers**. This allows you to run the exact same asset code against entirely different infrastructure stacks simply by changing a configuration file.

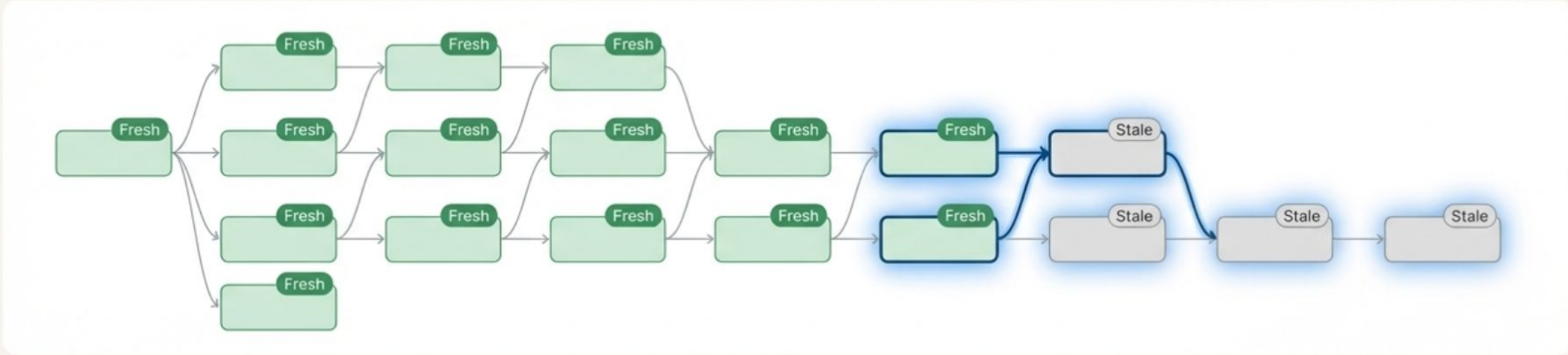


The same code runs in both environments, with no changes.

Orchestration That Understands Your Data

Dagster moves beyond static cron schedules. With **Jobs**, **Schedules**, and **Sensors**, you can automate your pipelines based on time, external events, or the state of the data itself.

Declarative Scheduling: You define the desired state of your assets with **Freshness Policies** (e.g., “this asset should never be more than 90 minutes stale”). Dagster automatically computes the minimal set of runs **needed** to satisfy those policies, avoiding redundant and expensive computations.

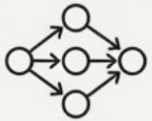


Dagster runs only what’s necessary, saving compute costs and time.

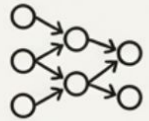
The Natural Evolution of Data Practice

The shift from imperative tasks to declarative assets is as fundamental as the web development world's move from imperative frameworks like Angular 1 to declarative ones like React. It's the next logical step for managing complexity in data platforms.

Past
Manual Scripts & Cron



Present (Legacy)
Imperative Task Orchestrators (e.g., Airflow)



Future
Declarative Asset Orchestrators (Dagster)

Key Differentiators Recap

- **Local Development & Testing**: Built-in from day one.
- **Software-Defined Assets**: The right core abstraction for data.
- **Decoupled Environments**: True portability from dev to prod.

Dagster Concepts

Dagster Concept	Role with Meltano/dbt	Dagster Syntax Example
Asset	Represents dbt models as assets (tables/views).	`@asset` `def my_dbt_model(): ...`
Pipeline (Job)	Orchestrates Meltano runs and dbt model executions.	`@job(resource_defs={...})` `def my_pipeline(): ...`
Resource	Configures access to Meltano projects, DBs, etc.	`from dagster_meltano import meltano_resource`
Schedule	Automates pipeline/job execution (e.g., daily runs).	`@schedule(job=my_pipeline, cron_schedule="0 0 * * *")`
Definitions	Registers all assets, jobs, resources, schedules.	`Definitions(assets=[...], jobs=[...], resources={...})`

Data Quality & Testing

The Foundational Challenge: Data Quality Is a Business Imperative

Poor data quality leads to flawed decision-making, regulatory non-compliance, and increased technical debt. As AI consultancy Provectus notes, data errors require engineers to "**repeatedly return to fixes**," taxing resources and eroding trust. For data to be a competitive edge, it must be **discoverable, manageable, observable**, and **reliable**.

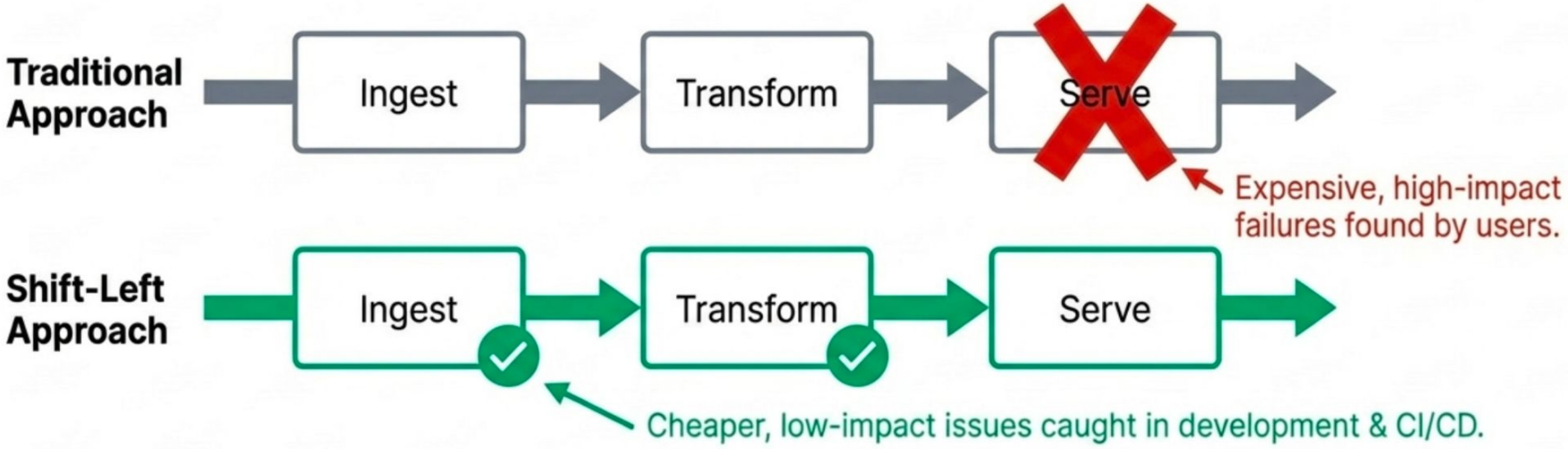
Key Problems Highlighted

Incorrect Analytics
Flawed insights lead to poor strategic and tactical decisions.

Increased Technical Debt
Constant, reactive fixes for data issues drain engineering resources.

Eroding Trust
Business users (BI, Marketing, Sales) lose confidence in the data they rely on.

Fail Fast, Fail Early. Move Quality Checks Upstream.



The 'Shift-Left' hypothesis dictates that testing and validation should be performed earlier in the data lifecycle. Integrating quality checks into development and CI/CD workflows minimizes resolution time and prevents bad data from ever reaching production systems.

A Shared Language for Measuring “Fit for Purpose”

Accuracy

Data reflects real-world truth.

A GPS system shows a restaurant's location 2 blocks away from its actual address.

Completeness

All required data is present and populated.

A customer database has no null `user_id` values.

Consistency

Data is uniform and non-contradictory across different systems.

A product's price is listed as €19.99 on the website but €24.99 in the mobile app.

Timeliness

Data is up-to-date and available when needed.

Sales data for the previous day is available for reporting by 9 AM daily.

Uniqueness

Each record is distinct; no duplicate data exists in a dataset.

A patient record system has two entries for the same person with identical details.

Validity

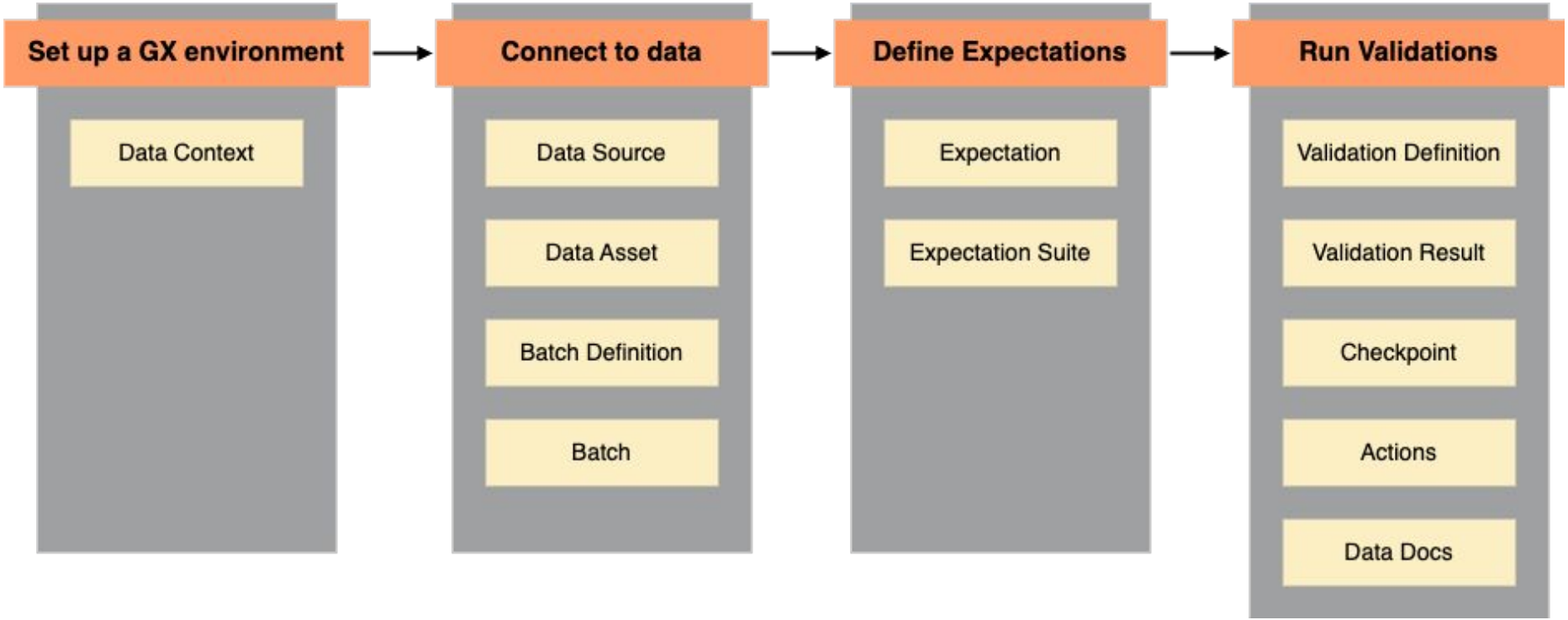
Data conforms to defined business rules, formats, and constraints.

A form field for a birthdate rejects an entry of 'February 30' because it is not a real date.

2023 State of Data Survey - Data Tools (800+ respondents)

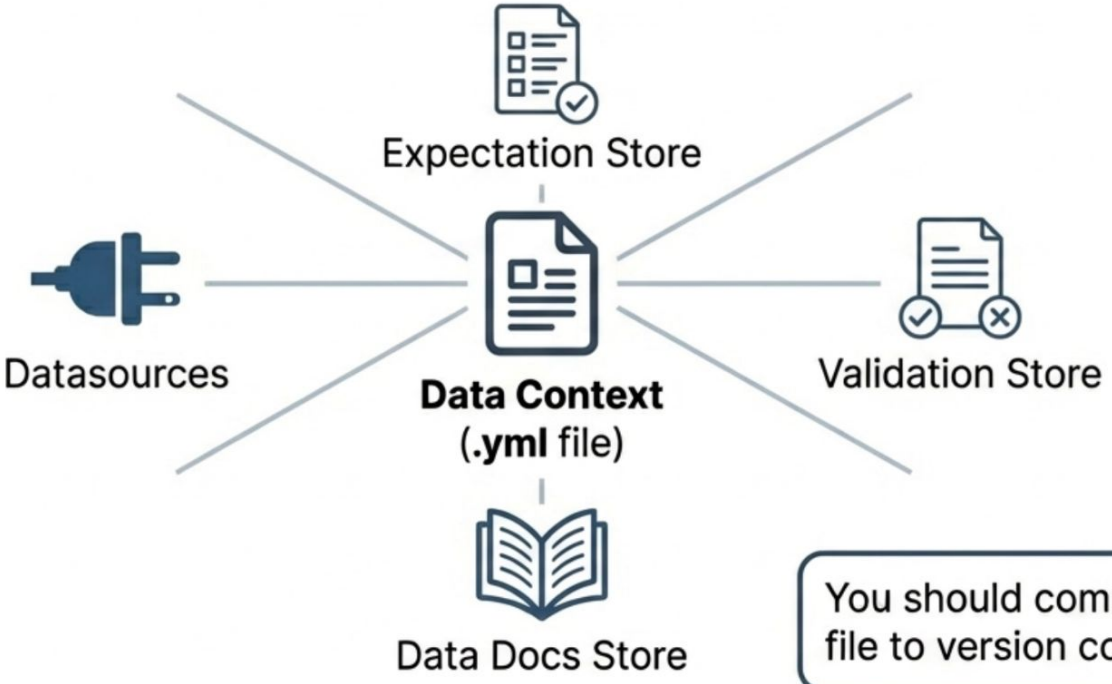
https://state-of-data.com/data-tooling-insights--data-quality

The Great Expectations Workflow



The Data Context manages your project's configuration and state.

The **Data Context** is the primary entry point for the Great Expectations API. It manages configuration for Datasources, Checkpoints, and Stores. This configuration is typically stored in a `great_expectations.yml` file.



Key Concept: Stores

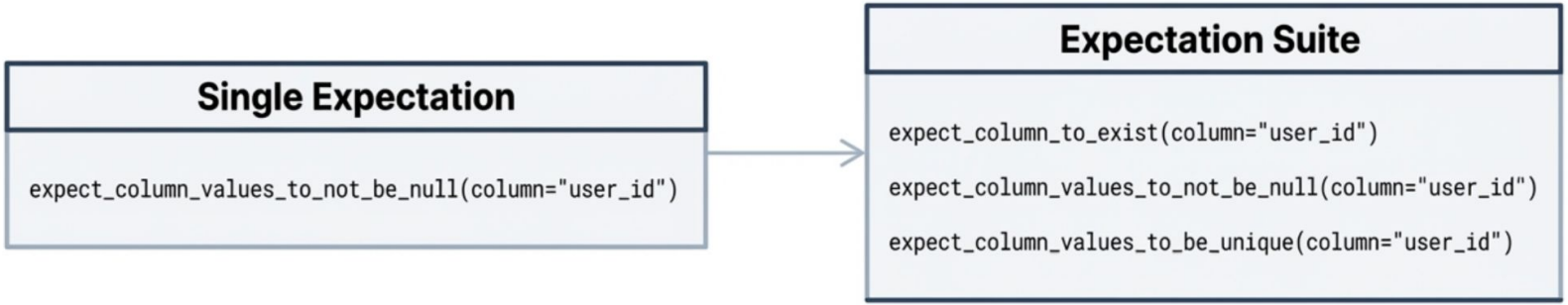
A **Store** is a generalized way of saving and retrieving GX objects. The Data Context manages stores for:

- Expectation Suites
- Validation Results
- Metrics
- Data Docs sites

You should commit your `great_expectations.yml` file to version control to share it with your team.

It all starts with the Expectation.

An **Expectation** is how we communicate the way data *should* appear. It's a declarative, human-readable assertion about your data.



When Expectations are grouped together to define a kind of data asset (e.g., 'monthly user signups'), we call it an **Expectation Suite**.

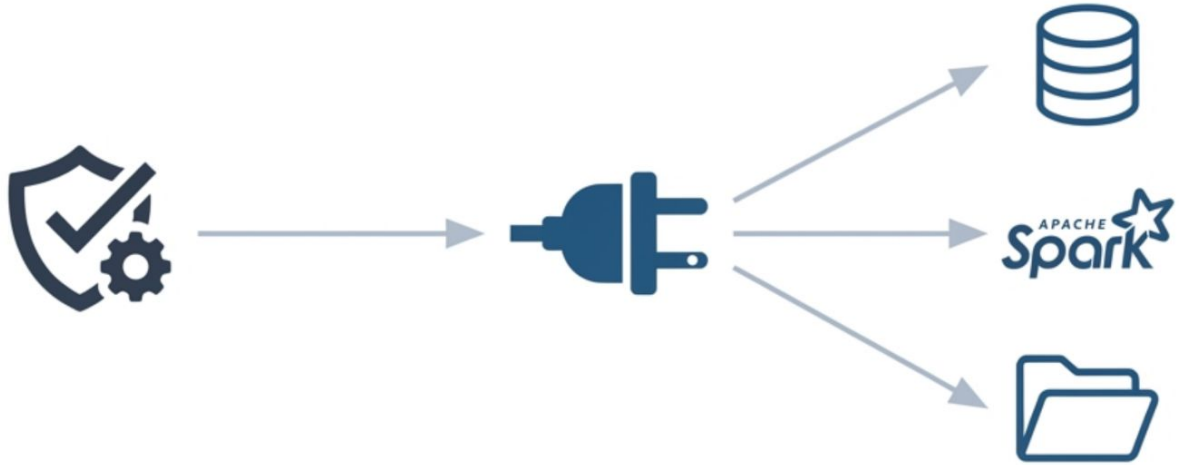
This suite becomes the shared, verifiable 'contract' for what that data asset should look like.

Expectations Gallery https://greatexpectations.io/expectations/

Datasources connect Great Expectations to your data systems.

A **Datasource** is the first thing you configure. It brings together two key elements:

1. A way of interacting with data (e.g., a database engine, a Spark cluster, the local filesystem).
2. A description of specific data you want to access (e.g., the `taxi_rides` table for last month).



Key Takeaway: With a Datasource configured, you can get a **Batch** of data to validate.

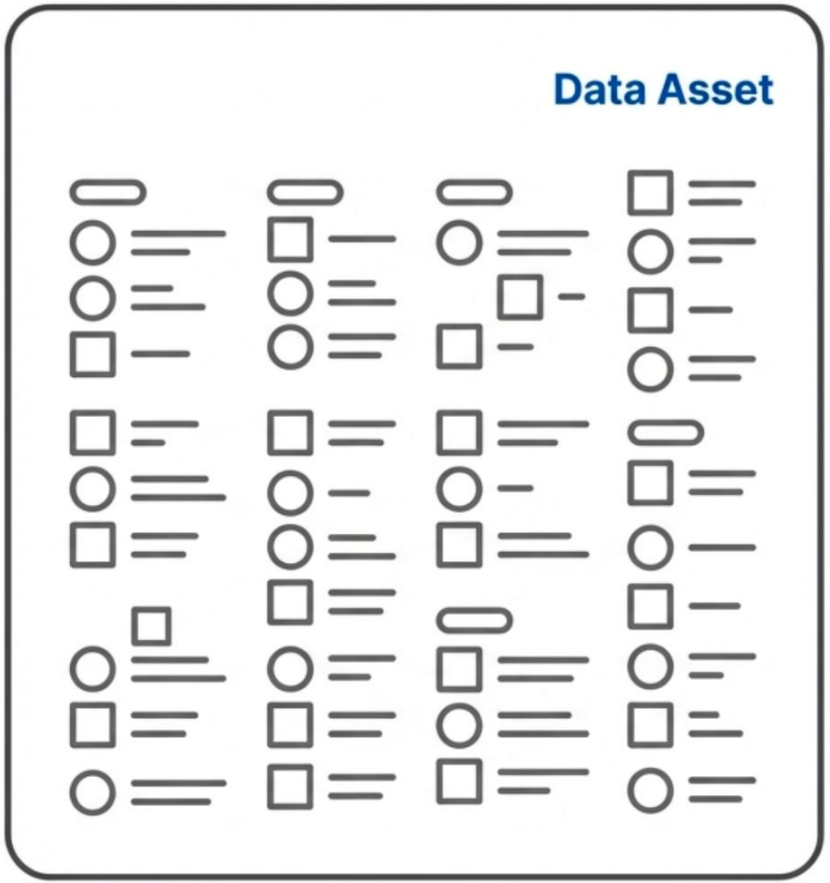
The Tour Begins with a Simple Question: What Data Are You Protecting?

We start with the highest-level concept, the fundamental subject of all data quality work. This is the **Data Asset**.

A formal definition: “A logical collection of records.”

> “A collection of records is a Data Asset when it’s **worth giving it a name**.”

- a user table in a database
- monthly financial data
- a collection of event log data



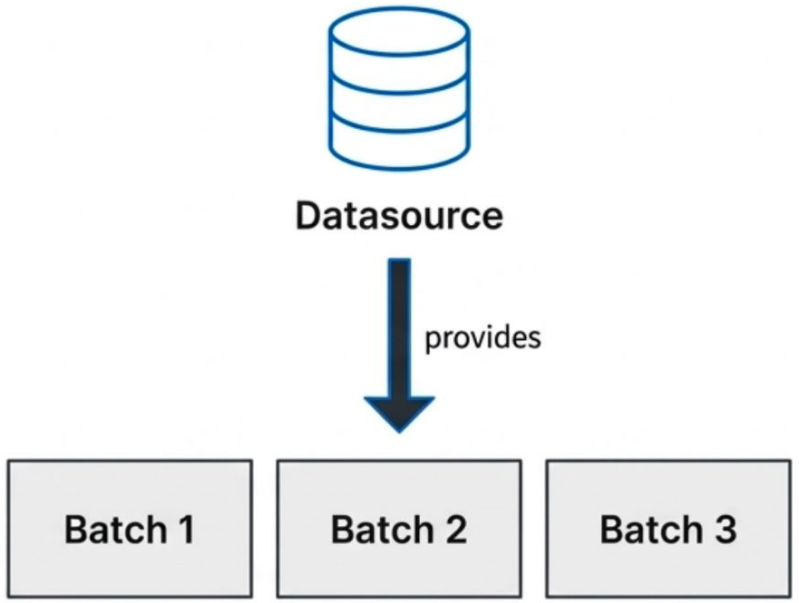
Connecting the Ideal to the Real: Datasources and Batches

Datasources: This is your connection to a data system—a database, Spark cluster, or filesystem. It defines the "where."

Batch: This is the critical concept. No matter the source, Great Expectations validates **batches** of data.

A **Batch** is a discrete subset of a Data Asset.

- For a data engineer, a Batch might be a "month's delivery" of data.
- For an analyst, a Batch could be the result of a "specific query" like rides with >2 passengers.



Checkpoints are the quality gates for your data pipeline.

When deploying Great Expectations, you use a **Checkpoint** to run a validation. A Checkpoint bundles an Expectation Suite with a Batch of data and performs other actions upon completion.

Analogy: Think of a Checkpoint as an automated testing step in your CI/CD pipeline, but for data.

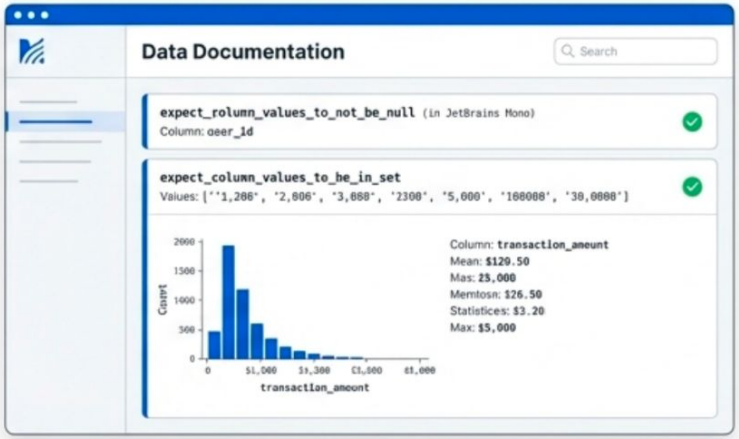
The Payoff: Validation Results and Living Data Docs

Validation Results

The direct, machine-readable output of a Checkpoint. A detailed JSON report on the success or failure of each Expectation for a given Batch.

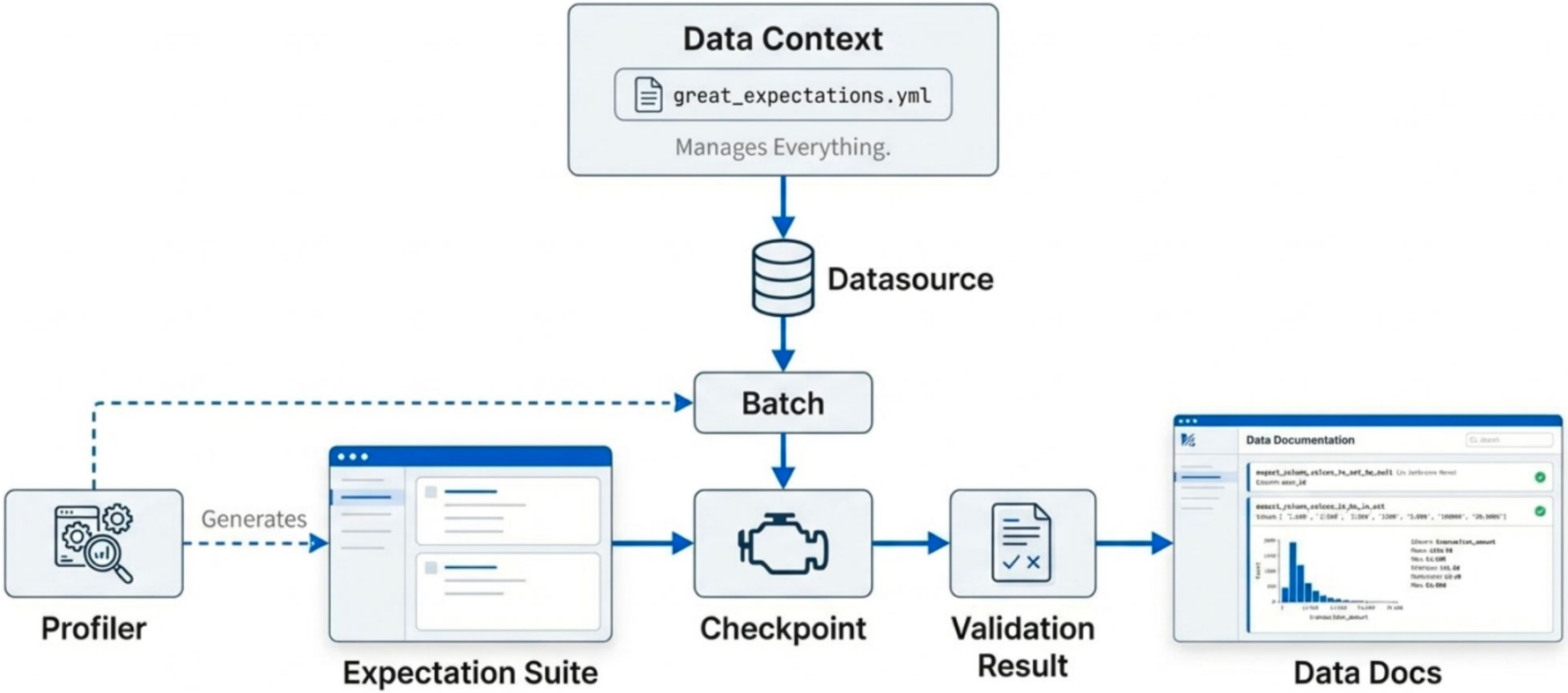
Data Docs

The human-readable output. A living documentation site generated from your Expectation Suites and Validation Results.



****Tests are docs, and docs are tests.****

Putting It All Together: The Great Expectations Mental Model



Codifying Your Assumptions with `dbt test`

dbt provides four out-of-the-box generic tests that form the bedrock of in-warehouse validation. These simple YAML declarations are powerful tools for enforcing basic data integrity.

```yaml
# models/orders.yml
models:
  - name: orders
    columns:
      - name: order_id
        data_tests:
          - unique
          - not_null
      - name: status
        data_tests:
          - accepted_values:
              values: ['placed', 'shipped', 'completed', 'returned']

      - name: customer_id
        data_tests:
          - relationships:
              to: ref('customers')
              field: id
```

Enforces Uniqueness

Enforces Completeness

Enforces Validity

Enforces Consistency (Referential Integrity)

dbt

Data quality as code: Version-controlled, documented, and automated.

Moving Beyond Basic Integrity to Enforce Business Rules

Custom Generic Tests

Sometimes, built-in tests aren't enough. You can write custom macros for conditional logic. For example, ensuring `cancelled_at` is not null only if `status` is 'cancelled'.

```
-- tests/generic/not_null_when_status_is.sql

{% test not_null_when_status_is(model, column_name,
status_column, status_value) %}
  select * from {{ model }}
  where {{ status_column }} = '{{ status_value }}'
  and {{ column_name }} is null
{% endtest %}
```

Defines a custom generic test macro.

Introducing `dbt-expectations`

This package brings the power of Great Expectations' rich, declarative assertions directly into your `dbt` project. It's ideal for statistical, time-series, and other sophisticated checks.

`dbt-expectations` in Practice

```
# models/products.yml
models:
  - name: products
    columns:
      - name: list_price
        data_tests:
          - dbt_expectations.expect_column_values_to_be_between:
              min_value: 0
              max_value: 10000
              # Annotation: Ensures price is within a valid business range.

      - name: product_sku
        data_tests:
          - dbt_expectations.expect_column_values_to_match_regex:
              regex: '^PRD-[A-Z]{2}-[0-9]{6}$'
              # Annotation: Validates SKU format against a specific pattern.
```

Enforces business range values.

Validates patterns with regex.

Validates patterns with regex.

dbt Expectations

dbt Expectations is a package that extends dbt's testing functionality, inspired by the Great Expectations Python library.

- Provides 50+ test macros for common data quality checks
- Allows for complex data validation scenarios
- Integrates seamlessly with dbt's testing framework
- Supports both column-level and table-level tests

Key Features

- **Column Tests:** Validate individual column values, formats, and relationships
- **Table Tests:** Verify table-level properties like row counts, uniqueness across columns
- **Multi-column Tests:** Test relationships between columns within a table
- **Schema Tests:** Ensure schema stability and expected structure

Installation

Add to your dbt packages.yml file:

```
packages:
 - package: metaplane/dbt_expectations
   version: 0.10.9
```

Then run:

```
dbt deps
```

Basic Usage

Add tests to your schema.yml files:

```
models:
 - name: my_model
   columns:
     - name: user_id
       tests:
         - dbt_expectations.expect_column_values_to_not_be_null
         - dbt_expectations.expect_column_values_to_be_unique
```

Common Test Examples

Column Value Tests

```
- dbt_expectations.expect_column_values_to_be_between:
    min_value: 0
    max_value: 100

- dbt_expectations.expect_column_values_to_match_regex:
    regex: '^[A-Z]{2}$'

- dbt_expectations.expect_column_values_to_be_in_set:
      value_set: ['active', 'inactive', 'pending']
```

Table-Level Tests

```
models:
  - name: my_model
    tests:
      - dbt_expectations.expect_table_row_count_to_be_between:
          min_value: 1
          max_value: 1000

      - dbt_expectations.expect_table_columns_to_contain_set:
          column_list: ["id", "name", "created_at"]

      -
dbt_expectations.expect_table_row_count_to_equal_other_table:
          compare_model: ref('other_model')
```

Implementation Tips

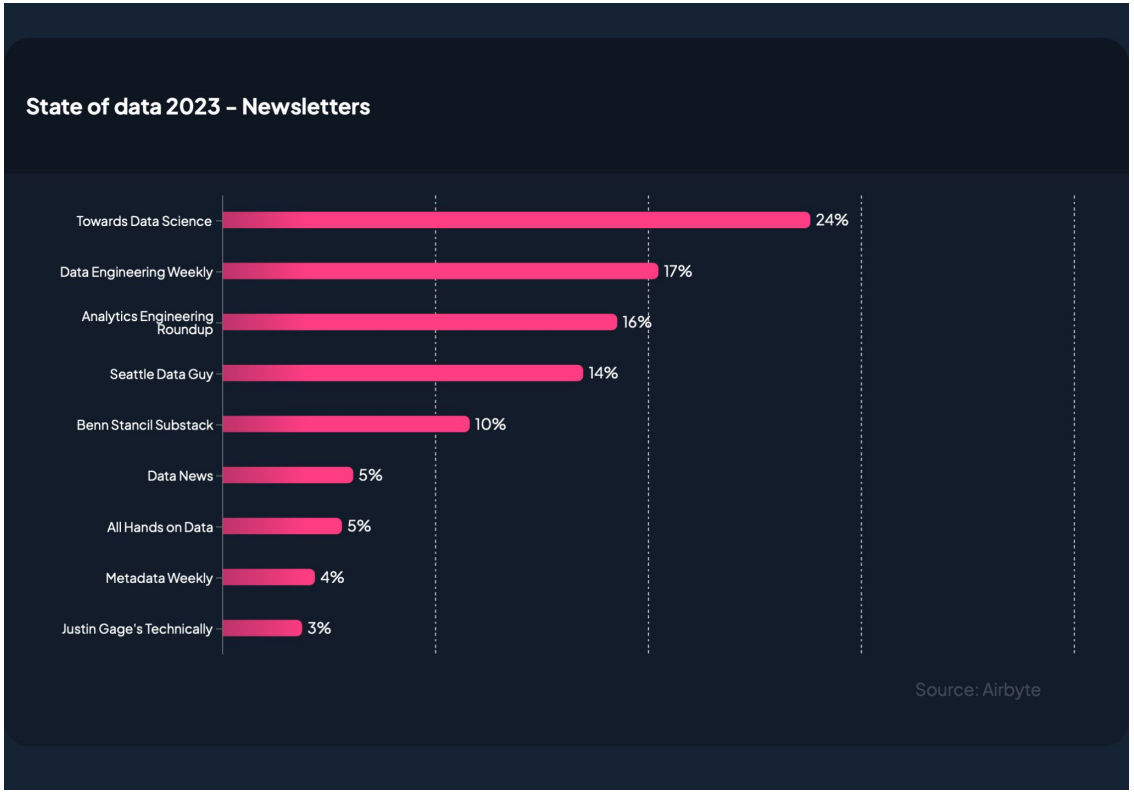
https://www.metaplane.dev/blog/dbt-expectations

Documentation - Available Tests

https://hub.getdbt.com/metaplane/dbt_expectations/latest/

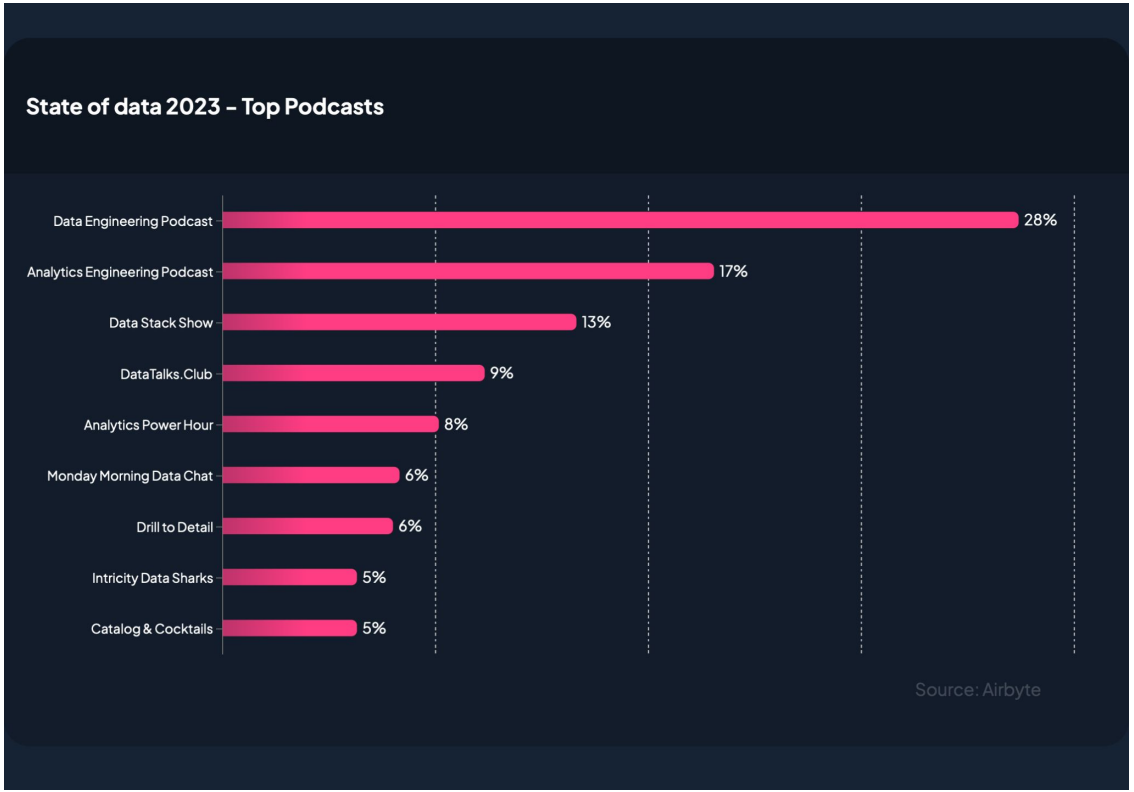
Bonus - Knowledge Sources

2023 State of Data Survey - Knowledge Sources



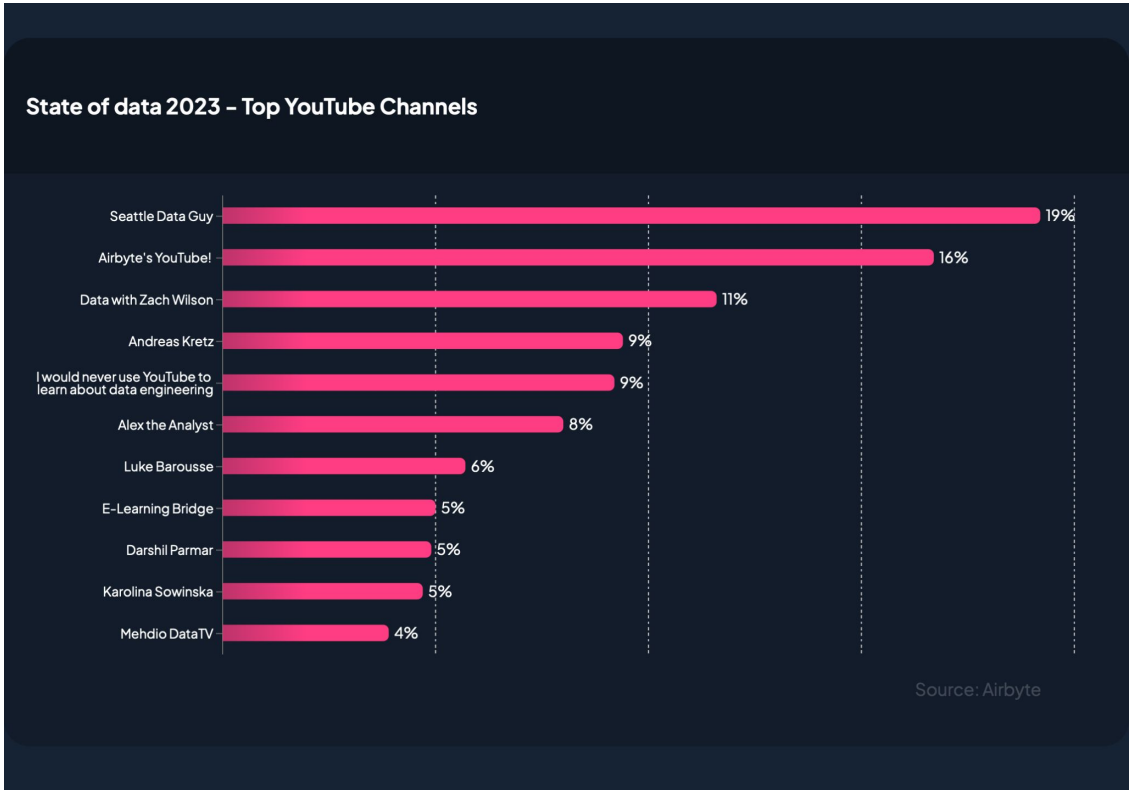
https://state-of-data.com/data-community-survey--newsletters

2023 State of Data Survey - Knowledge Sources



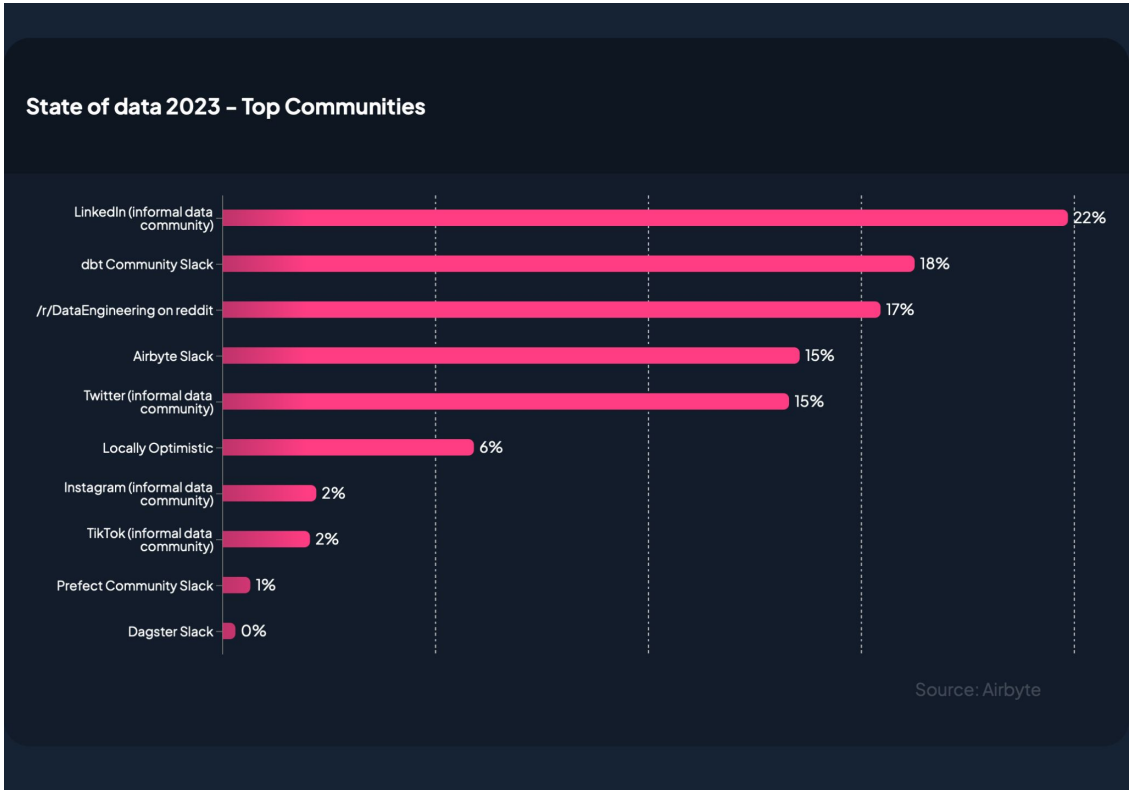
https://state-of-data.com/data-community-survey--podcasts

2023 State of Data Survey - Knowledge Sources



https://state-of-data.com/data-community-survey--youtube

2023 State of Data Survey - Knowledge Sources



https://state-of-data.com/data-community-survey--communities

Coding

1. Data Testing with Great Expectations
2. Testing Dbt

End of Lesson - Exit Ticket

Survey Link

https://www.slido.com