North Star View: Automation Test Strategy for GCP Data Products

# Vision

Enable trusted, high-quality, and observable data products through automation-first, shift-left testing across the GCP data lifecycle—anchored by data contracts, test pyramids, and continuous validation.

# Strategic Goals

|  |  |
| --- | --- |
| Objective | Description |
| Shift Left Testing | Integrate quality from source to consumer, starting early in development |
| Data Product Reliability | Deliver high-trust FDP/ODP views with minimal data issues |
| Automation First | Replace manual QA with pipelines that run quality checks continuously |
| Platform as Product | Build reusable testing infrastructure and templates for teams |
| Scalable Governance | Codify and track data quality using rules, metrics, and observability tools |

## 🔭 Vision

Enable trusted, high-quality, and observable data products through automation-first, shift-left testing across the GCP data lifecycle—anchored by data contracts, test pyramids, and continuous validation.

## 🚀 Strategic Goals

| **Objective** | **Description** |
| --- | --- |
| **Shift Left Testing** | Integrate quality from source to consumer, starting early in development |
| **Data Product Reliability** | Deliver high-trust FDP/ODP views with minimal data issues |
| **Automation First** | Replace manual QA with pipelines that run quality checks continuously |
| **Platform as Product** | Build reusable testing infrastructure and templates for teams |
| **Scalable Governance** | Codify and track data quality using rules, metrics, and observability tools |

## 🧱 Foundational Pillars

| **Pillar** | **Description** | **Tools** |
| --- | --- | --- |
| **Test Pyramid for Data** | Shift-left, layered tests (unit → transformation → integration → contract) | PyTest, dbt tests, Great Expectations, Soda |
| **CI/CD for Data Testing** | Integrated testing in deployment pipeline (pre-merge and post-merge) | Cloud Build, GitHub Actions, dbt Cloud |
| **Data Quality as Code** | Declarative, version-controlled DQ rules alongside pipelines | dbt-expectations, Great Expectations, Dataplex |
| **Data Contracts** | Enforced schemas between producers and consumers | Protobuf, JSON schema, Spectacles, OpenAPI |
| **Observability** | Runtime monitoring for freshness, completeness, drift | Monte Carlo, Bigeye, Dataplex, custom alerts |
| **Reusability & Standards** | Templates and modular test assets across all FDP/ODP pipelines | Airflow DAG templates, dbt project seeds, shared libraries |

## 🧭 Implementation Roadmap

### 📌 Phase 1: Foundation & Enablement

* ✅ Establish test pyramid as an org-wide principle
* ✅ Define reusable test templates (unit, transformation, DQ)
* ✅ Integrate dbt + PyTest + Great Expectations into CI
* ✅ Add PR validation pipelines
* ✅ Set up Dataplex data quality rules
* ✅ Profile raw/staging data using TFDV or Great Expectations

### 📌 Phase 2: FDP Quality Guardrails

* 🚧 Implement baseline test suites for all FDPs (schema, joins, rules)
* 🚧 Add dbt-expectations and custom SQL assertions
* 🚧 Integrate schema diff tools (e.g. DataDiff)
* 🚧 Monitor freshness and volume using Airflow/Dataplex

### 📌 Phase 3: ODP Contract & Consumer Protection

* 🚧 Add contract tests for APIs/BI consumers
* 🚧 Set up Looker CI and Spectacles for downstream validation
* 🚧 Enable anomaly and drift detection alerts
* 🚧 Track test coverage and failure rates per product

### 📌 Phase 4: Platformization & Automation Scaling

* 🛠️ Turn tests into reusable Airflow operators
* 🛠️ Automate test case generation from metadata
* 🛠️ Provide test orchestration UI or dashboards
* 🛠️ Integrate data quality SLAs and alerts
* 🛠️ Enable test authoring by product owners and analysts

## 🔗 Cross-Cutting Concerns

| **Area** | **Implementation** |
| --- | --- |
| **Security & Access Control** | Tokenize PII in test runs; enforce access policies on test environments |
| **Environment Strategy** | Isolate dev/test/prod in BigQuery and DAGs |
| **Test Data Management** | Use synthetic data or masked historical data |
| **Metadata & Lineage** | Integrate with Dataplex, dbt docs, and OpenLineage |
| **Documentation & Onboarding** | Provide central playbooks, templates, and onboarding guides |

## 📊 Quality & Testing Metrics

| **Metric** | **Description** | **Target** |
| --- | --- | --- |
| **Test Coverage** | % of FDP/ODP entities with active tests | > 90% |
| **Test Pass Rate** | % of tests passing in CI/CD | > 95% |
| **Data Drift Incidents** | Monthly anomalies due to unexpected schema or values | < 2 |
| **Time to Detect (TTD)** | Time to alert after issue is introduced | < 30 mins |
| **Time to Fix (TTF)** | Time to resolution after alert | < 4 hours |
| **Consumer Trust Score** | Based on data complaints or BI feedback | > 8/10 |

## 📘 Test Pyramid (Adapted from Martin Fowler)

pgsql

CopyEdit

┌────────────────────────────┐

│ Exploratory Testing │ ← BI / API

└────────────▲───────────────┘

▲ ▲

┌──────────┴─┴──────────────┐

│ Integration Tests │ ← Views, Joins

└──────────▲▲▲──────────────┘

▲ │││

┌───────┴──┴┴┴──────────────┐

│ Transformation Tests │ ← SQL Logic, Business Rules

└────────▲▲▲▲───────────────┘

▲ ▲ ▲ ▲

┌───────┴─┴─┴─┴─────────────┐

│ Unit Tests (Pipeline) │ ← Beam Functions, Schema Checks

└───────────────────────────┘

## 🧠 North Star Diagram (Data Flow Testing Stages)

pgsql

CopyEdit

┌────────────────────────────┐

│ Data Consumers (BI/API)│

└────────────┬───────────────┘

│

┌────────▼────────┐

│ ODP Layer │ ← Contract & KPI Tests

└────────┬────────┘

│

┌────────▼────────┐

│ FDP Layer │ ← Schema, Joins, Business Rules

└────────┬────────┘

│

┌──────────────▼──────────────┐

│ Staging / Cleansed Zone │ ← Schema, DQ Profiling, Duplicates

└──────────────┬──────────────┘

│

┌──────────────▼──────────────┐

│ Raw / Ingestion Zone │ ← Format, Type, Null, Contract Tests

└─────────────────────────────┘

↘ Shift-Left Starts Here ↙

## ✅ Success Checklist

* CI/CD pipelines include data quality tests
* Automated tests cover all critical views
* Contracts enforced between upstream and downstream
* Alerts for schema, freshness, and volume issues
* Data quality KPIs tracked in real-time dashboards
* Developers and analysts enabled to write tests

## 📚 References

* Martin Fowler – Test Pyramid
* ThoughtWorks – Tech Radar & Data Quality Principles
* Jez Humble – Continuous Delivery
* Barr Moses – Data Observability Framework
* dbt Labs – Best Practices for Data Testing

Let me know if you'd like this exported as:

* 📄 PDF or Markdown
* 🧩 Editable **Confluence storage format (XML or wiki markup)**
* 📊 A slide deck version for leadership

Also happy to generate reusable **test code templates**, **dbt packages**, or **CI pipeline YAMLs** for your team.

##### You said:

export it to word doc and slide format

###### ChatGPT said:

## ****Adding More Light: Key Techniques & Tools****

### 1. ****Data Drift Incidents****

**Definition**: Unexpected changes in data distribution, schema, volume, or freshness that may compromise data quality.

#### 🛠️ ****Tools for Detection****

* **Monte Carlo**, **Bigeye**, **Dataplex Observability**
* Open-source: **Evidently**, **TFDV**, **Soda Core**

#### ****Approach****

1. **Baseline Profiles** – Establish historical thresholds for freshness, volume, schema, value distributions.
2. **Real-Time Monitoring** – Use ML or thresholds to detect anomalies.
3. **Alerting + RCA** – Route incidents with lineage metadata to ownership.
4. **Remediation** – Automate rollbacks or fallback data; notify teams for manual fixes.
5. **Metrics to track**: Count of drift incidents, mean time to detection, time to resolve.

#### ****Example****

* **Column Null Drift:** If user\_age null rate spikes from 5% to 40%, a monitor triggers an incident and lineage lookup shows upstream source malfunction.

### 2. ****Looker CI****

**Purpose**: Automated validation of Looker models (LookML) before deployment to production.

#### 🛠️ ****Tools & Plugins****

* **Spectacles** (Looker CI for data tests/validations)
* **Looker-Rundeck**, **Looker Action Hub**
* Git/GitHub Actions for CI

#### ****Approach****

1. **Git-based LookML** with enforced linting and style rules.
2. Configure **Spectacles** to:
   * Run datagroups, explores, tests using production data in staging.
   * Compare metrics across LookML versions.
3. **CI Integration**: Run Spectacles via GitHub Actions. Block PRs on failures.
4. **Scheduling**: Nightly regression runs to detect metrics drift.

#### ****Example****

* A user modifies Explore join logic; Spectacles reports dim user count decreased unexpectedly, blocking PR.

### 3. ****Contract Testing in Data****

**Definition**: Ensuring that changes to FDP/ODP views or APIs don’t break downstream consumers.

#### 🛠️ ****Approaches & Tools****

1. **Schema Contracts**: JSON Schema, Avro, or Protobuf.
2. **Contract Validation Tools**:
   * **OpenAPI + Dredd**, **Postman**, **Pact**
   * SQL layer: **Great Expectations**, **dbt tests**
3. **CI Integration**:
   * Upon view changes, regenerate schema docs.
   * Run contract tests against expected schema/row shapes.
4. **Consumer Simulations**:
   * Dummy queries by downstream apps or BI tools; validate output types, shape, freshness.

#### ****Example****

* Upstream changes remove column address\_line2, contract test fails as Looker Explore still expects it. Migration disabled until resolved.

## 🔧 Summary Table

| **Topic** | **Tools/Techniques** | **Example Scenario** |
| --- | --- | --- |
| **Data Drift** | Monte Carlo / Bigeye / Evidently / Soda Core | Null spike alert in user\_age |
| **Looker CI** | Git + Spectacles + GitHub Actions | Broken Explore metrics blocked |
| **Contract Tests** | JSON Schema, Protobuf, dbt, Great Expectations, Pact, Dredd | View drop of column → consumer failure |

If you'd like, I can build:

* Code templates for drift detection monitors (e.g. Evidently/GCP)
* Example Spectacles CI YAML for Looker
* A contract test example using JSON Schema and BigQuery

--.

Here’s a **complete example of contract testing** for data pipelines built on **BigQuery**, especially when **you don’t have APIs**—just SQL views, tables, and batch data pipelines.

## 📘 What is Data Contract Testing (in a data pipeline context)?

A **data contract** defines the expectations between:

* **Producers**: Raw data sources or data engineers building FDP/ODP
* **Consumers**: Analytics teams, dashboards, ML models, downstream pipelines

A **contract test** checks whether the **produced data still adheres to the contract** — typically **schema, shape, types, values, semantics** — even after changes.

## ✅ What to Test in a Data Contract

| **Category** | **Examples** |
| --- | --- |
| **Schema** | Column presence, names, types, nullable, primary keys |
| **Shape** | Row count expectations, unique keys, joins |
| **Semantics** | Value domains (status must be ‘active’, ‘inactive’), categorical validity |
| **Freshness** | Updated within the last X hours |
| **Volume** | Record count thresholds, e.g., at least 90% of yesterday |

## 🛠️ Tools You Can Use (Without APIs)

| **Tool** | **Use Case** | **Open Source?** |
| --- | --- | --- |
| **dbt tests** | Schema tests, relationships, custom SQL tests | ✅ |
| **Great Expectations** | Declarative tests on data batches | ✅ |
| **Soda Core / SodaCL** | Contract-like validation using YAML + SQL | ✅ |
| **Deequ (Scala)** | DQ constraints + profiles (on Spark) | ✅ |
| **Tecton Contracts / Contracts by Truelayer** | Advanced data contracts (early stage, emerging) | Partial |
| **TFDV (TensorFlow Data Validation)** | Schema + stats validation for ML pipelines | ✅ |

## 💡 BigQuery + dbt Contract Testing Example

Assume you're building a **FDP view fdp\_user\_profiles**. The downstream contract is:

* Must contain user\_id, email, signup\_date, status
* user\_id is non-null and unique
* status only has 'active' or 'inactive'
* At least 100K rows expected daily
* Table is updated daily by 3am

### Step 1: Define schema + tests in dbt

yaml

CopyEdit

# models/fdp\_user\_profiles.yml

version: 2

models:

- name: fdp\_user\_profiles

description: "Final user profile view"

columns:

- name: user\_id

description: "Unique user identifier"

tests:

- not\_null

- unique

- name: status

tests:

- accepted\_values:

values: ['active', 'inactive']

- name: email

tests:

- not\_null

### Step 2: Add a volume test

Create a custom SQL test:

sql

CopyEdit

-- tests/test\_fdp\_user\_profiles\_row\_count.sql

SELECT

CASE

WHEN COUNT(\*) < 100000 THEN 'FAIL'

ELSE 'PASS'

END as result

FROM {{ ref('fdp\_user\_profiles') }}

### Step 3: Add freshness check with dbt freshness or a BigQuery SQL monitor

yaml

CopyEdit

# sources.yml

sources:

- name: core\_data

tables:

- name: user\_profiles\_raw

freshness:

warn\_after: { count: 6, period: hour }

error\_after: { count: 12, period: hour }

### Step 4: Run in CI

In GitHub Actions or Cloud Build:

bash

CopyEdit

dbt deps

dbt run

dbt test --store-failures

dbt source freshness

Fail the CI step if any test fails.

## 🧪 Contract Testing Example with Great Expectations + BigQuery

python

CopyEdit

from great\_expectations.dataset import SqlAlchemyDataset

from sqlalchemy import create\_engine

engine = create\_engine("bigquery://project-id")

dataset = SqlAlchemyDataset("fdp\_user\_profiles", engine=engine)

dataset.expect\_column\_to\_exist("user\_id")

dataset.expect\_column\_values\_to\_not\_be\_null("user\_id")

dataset.expect\_column\_values\_to\_be\_unique("user\_id")

dataset.expect\_column\_values\_to\_be\_in\_set("status", ["active", "inactive"])

dataset.expect\_table\_row\_count\_to\_be\_between(min\_value=100000, max\_value=5000000)

Run this daily using Cloud Composer or in CI/CD.

## 📏 Summary: What Happens When Contract Breaks?

| **Scenario** | **Contract Test** | **Action** |
| --- | --- | --- |
| Column dropped | expect\_column\_to\_exist | Block merge or raise alert |
| Value changed | expect\_column\_values\_to\_be\_in\_set | Alert or revert |
| Delayed data | Freshness test | Notify SLA breach |
| Data skew | Row count threshold | Trigger incident response |

## 🧩 Bonus: Store Contracts as JSON/YAML

Using SodaCL:

yaml

CopyEdit

checks for fdp\_user\_profiles:

- schema:

fail:

when schema changes:

- column\_removed

- column\_type\_changed

- freshness(column: updated\_at) < 12h

- row\_count > 100000

- values in column status must be in [active, inactive]

Would you like:

* A ready-to-run **dbt project** with these contract tests?
* A **SodaCL YAML template** with dynamic BigQuery integration?
* A **Cloud Build YAML** to run these contract tests automatically?

--..

Here are **three robust GitHub repositories** that exemplify **automation frameworks** for **data pipeline testing** on GCP/BigQuery, with **BDD-style**, **contract-testing**, and **end-to-end pipelines**—perfect for enhancing your data quality implementation.

## 1. ****tinyquery****

**↪️ GitHub: alanpierce/tinyquery**  
A Python in-memory emulator for the BigQuery API, enabling **fast, local unit testing** of BigQuery-dependent pipeline logic without hitting the actual service [Stack Overflow+3Medium+3GitHub+3](https://medium.com/%40claudiofilho22/end-to-end-data-engineering-project-using-dbt-bigquery-cloud-run-eae10e0c7dc3?utm_source=chatgpt.com)[GitHub](https://github.com/alangpierce/tinyquery?utm_source=chatgpt.com).  
**Highlights**:

* Implements SQL operations (SELECT, JOIN, GROUP BY, etc.) in-memory.
* Seamless switch between real BigQuery and tinyquery via dependency injection.
* Ideal for BDD-style unit tests covering transformation logic in Python.

## 2. ****ChuQuEmeka/Airflow-dbt-bigquery-gcs-healthcare-data-pipeline****

**↪️ GitHub: ChuQuEmeka/Airflow-dbt-bigquery-gcs-healthcare-data-pipeline**  
An **end-to-end healthcare data pipeline** combining Airflow for orchestration, dbt for transformation & tests, and BigQuery & GCS as the data store [GitHub](https://github.com/alangpierce/tinyquery?utm_source=chatgpt.com)[LinkedIn+6GitHub+6GitHub+6](https://github.com/ChuQuEmeka/Airflow-dbt-bigquery-gcs-healthcare-data-pipeline?utm_source=chatgpt.com).  
**Highlights**:

* Contains **pytest-driven DAG tests** (tests/dags/) for BDD-style task validation.
* Built-in **dbt tests** (schema & data validity)
* Great template for contract testing by asserting expected schema and row-level contracts via dbt.

## 3. ****chayansraj/Data-Pipeline-with-dbt-using-Airflow-on-GCP****

**↪️ GitHub: chayansraj/Data-Pipeline-with-dbt-using-Airflow-on-GCP**  
A comprehensive project that integrates Airflow, dbt, **Soda Core**, and BigQuery to build and test data pipelines .  
**Highlights**:

* Utilizes Soda’s **YAML-based data quality checks** for freshness, schema, and value integrity—ideal for contract-style testing.
* Airflow DAGs contain test tasks invoking Soda before allowing pipeline continuation.
* Offers a full **BDD-ish testing environment**, with each pipeline run verifying contract compliance before proceeding.

### 🧩 Framework Comparison & BDD/Contract Testing Fit

| **Repository** | **BDD Coverage** | **Contract Testing** | **Tools Used** | **Best For** |
| --- | --- | --- | --- | --- |
| **tinyquery** | ✅ Unit-level SQL logic test | ✅ Schema, transformation logic verification | Python, pytest | Logic-heavy workloads with local testing |
| **ChuQuEmeka/...** | ✅ DAG-level BDD tests via pytest | ✅ dbt contract/schema tests | Airflow, dbt, pytest | End-to-end template with built-in validity tests |
| **chayansraj/...** | ⚠️ Airflow + Soda for contract checks | ✅ Soda-based schema/value monitoring | Airflow, dbt, Soda | Emulating contract tests via YAML in pipelines |

### 🚀 How to Leverage These in Your Stack

1. **Isolation + BDD (tinyquery)**
   * Use tinyquery for local pre-checking of SQL and Python code.
   * Write BDD tests with pytest (Given raw input, When transformation runs, Then expected output).
2. **Orchestration + Integration (Airflow + pytest)**
   * Define 'bdd\_' styled DAG tests (e.g., “source file arrives → table loaded → soda/dbt test passes → downstream runs”).
   * Example: tests/dags/test\_dag\_healthcare.py.
3. **Contract Testing (dbt + Soda)**
   * Use **dbt tests** to assert schema, uniqueness, and value domains.
   * Use **Soda Core** to codify contracts in YAML (freshness, thresholds).
   * Integrate these into DAG or CI pipelines to enforce production-grade contracts.

### 🛠️ Example BDD Contract-Test Scenario (combining all three)

gherkin

CopyEdit

Feature: User Profile Pipeline

Scenario: Freshly ingested user records

Given a sample raw CSV with 5 user records

When the Airflow DAG 'user\_profiles' is triggered

Then BigQuery table 'stg\_user\_profiles' is created

And dbt test 'not\_null\_user\_id' passes

And Soda check 'row\_count > 1000' and 'status in [active,inactive]' passes

* **tinyquery**: Validate SQL transformation logic (unit test)
* **Airflow pytest**: Trigger DAG and assert table creation
* **dbt/Soda**: Contract tests in CI and pipelines

Let me know if you'd like me to:

* Spin up boilerplate code using any of these templates
* Generate sample pytest DAG tests
* Configure CI (e.g. GitHub Actions) for dbt + Soda testing
* Explore other frameworks for BDD-style data testing on GCP

--..

Awesome! I'll help you build a **boilerplate automation framework** using **dbt** that covers:

* **BDD-style testing** (using pytest-bdd)
* **Contract testing** (dbt schema & custom tests)
* **Integration with BigQuery** on GCP
* Includes examples for unit tests, data quality tests, and DAG-like orchestration tests (simulated via pytest)

### What you'll get:

1. **DBT project scaffold** with models and tests (schema + custom SQL)
2. **Pytest + pytest-bdd** tests to run BDD scenarios verifying:
   * Model transformations
   * Contract tests (unique, not null, accepted values)
   * Row counts & data freshness
3. **Sample BDD feature file** describing user flows
4. **Test runner** that connects to BigQuery for real executions or can be adapted for local/mocked runs
5. A **README** explaining how to run and extend

## Step 1: DBT Project Setup

### models/user\_profiles.sql

sql

CopyEdit

WITH raw\_data AS (

SELECT \* FROM {{ source('raw', 'user\_profiles') }}

)

SELECT

user\_id,

email,

signup\_date,

status

FROM raw\_data

WHERE status IN ('active', 'inactive')

### tests/schema.yml

yaml

CopyEdit

version: 2

models:

- name: user\_profiles

columns:

- name: user\_id

tests:

- not\_null

- unique

- name: email

tests:

- not\_null

- name: status

tests:

- accepted\_values:

values: ['active', 'inactive']

## Step 2: Custom DBT Test (SQL)

tests/custom/test\_row\_count.sql:

sql

CopyEdit

SELECT

CASE WHEN COUNT(\*) >= 1000 THEN 0 ELSE 1 END AS failures

FROM {{ ref('user\_profiles') }}

## Step 3: Pytest BDD Feature File

features/user\_profiles.feature

gherkin

CopyEdit

Feature: User Profiles Data Product

Scenario: Validate user\_profiles table and data quality

Given the user\_profiles model is run

When the contract tests execute successfully

Then the row count should be greater than or equal to 1000

## Step 4: Pytest BDD Step Definitions

tests/test\_user\_profiles.py

python

CopyEdit

import subprocess

import pytest

from pytest\_bdd import scenarios, given, when, then

scenarios('../features/user\_profiles.feature')

@given('the user\_profiles model is run')

def run\_dbt\_model():

result = subprocess.run(['dbt', 'run', '--select', 'user\_profiles'], capture\_output=True, text=True)

assert result.returncode == 0, f"DBT run failed: {result.stderr}"

@when('the contract tests execute successfully')

def run\_contract\_tests():

result = subprocess.run(['dbt', 'test', '--select', 'user\_profiles'], capture\_output=True, text=True)

assert result.returncode == 0, f"DBT tests failed: {result.stderr}"

@then('the row count should be greater than or equal to 1000')

def test\_row\_count():

# Custom SQL test result

result = subprocess.run(

['dbt', 'test', '--select', 'test\_row\_count'],

capture\_output=True, text=True

)

assert result.returncode == 0, f"Row count test failed: {result.stderr}"

## Step 5: How to Run

bash

CopyEdit

# Install dependencies

pip install dbt-bigquery pytest pytest-bdd

# Run all BDD tests

pytest tests/test\_user\_profiles.py

## Bonus: CI Integration with GitHub Actions (simplified)

.github/workflows/dbt-tests.yml

yaml

CopyEdit

name: DBT Contract & BDD Tests

on: [push, pull\_request]

jobs:

test:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Setup Python

uses: actions/setup-python@v4

with:

python-version: 3.9

- name: Install dependencies

run: pip install dbt-bigquery pytest pytest-bdd

- name: Run BDD tests

run: pytest tests/test\_user\_profiles.py

Framework

Got it! I’m preparing a full runnable **dbt + pytest-bdd** automation test framework project zip for you.

It will include:

* dbt models and schema tests
* Custom SQL test for row count
* pytest-bdd feature and test files
* README with instructions
* GitHub Actions CI workflow YAML

I’ll package everything so you can just unzip and run.

Give me a moment. I’ll upload it shortly.

Here is the **full runnable dbt + pytest-bdd automation framework project** zipped for you:

dbt\_bdd\_automation\_framework.zip

### What’s inside:

* models/user\_profiles.sql - dbt model SQL
* models/schema.yml - dbt schema & contract tests
* tests/custom/test\_row\_count.sql - custom SQL test for row count
* features/user\_profiles.feature - BDD feature file
* tests/test\_user\_profiles.py - pytest-bdd test steps
* README.md - setup & run instructions
* .github/workflows/dbt-tests.yml - GitHub Actions CI workflow example
* dbt\_project.yml - dbt project config

### How to get started

1. Unzip the file
2. Update your profiles.yml or environment to connect dbt to your BigQuery dataset
3. Define your source tables in dbt or via BigQuery datasets
4. Run tests via:

bash

CopyEdit

pip install dbt-bigquery pytest pytest-bdd

pytest tests/test\_user\_profiles.py