# Ronesans Governance Strategy

The objective is to create a data governance strategy for Ronesans Holding. A proposal of how to manage the proposal is created.

## Proposal

I propose a **dbt-native governance system** built around **Elementary** and a structured, collaborative workflow. This system will enable us to automatically test our data against business rules, monitor for anomalies, and create a transparent, auditable process for managing data quality in partnership with our customer.

**The Proposed Governance Workflow**

**A diagram of a data flow

AI-generated content may be incorrect.**

The core of our strategy is a closed-loop, collaborative process that connects business needs directly to technical implementation and monitoring. This ensures that our data quality checks are always aligned with business requirements.

The workflow operates in six key stages:

1. **Rule Request & Collaboration (The "Human" Layer):**
   * **Tool:** A shared **Data Rules Registry** spreadsheet (Excel/Google Sheets).
   * **Process:** The customer logs new data quality rules, asks questions, and reports issues directly in the shared registry. This document serves as our central "source of truth" for business requirements.
2. **Review and Approval:**
   * **Process:** On a regular cadence (e.g., weekly), our team meets with the customer to review all Requested rules in the registry. We discuss, clarify, and formally move the rule's status to Approved. This is a critical step for alignment.
3. **Implementation (The "dbt" Layer):**
   * **Process:** Our data team translates each Approved rule from the registry into a specific test within our dbt project's schema.yml files. This keeps our data quality logic tightly coupled with our data transformation code, ensuring it is version-controlled and tested.
4. **Automated Testing (The "Airflow & dbt" Layer):**
   * **Process:** Our existing Airflow DAGs will be updated with a new step that executes dbt test. This command runs all the Elementary and dbt tests against our Snowflake data warehouse as part of the regular data pipeline.
5. **Monitoring & Alerting (The "Elementary" Layer):**
   * **Process:** Elementary automatically collects the results of every test run. Its capabilities extend beyond simple checks to include anomaly detection (e.g., unexpected drops in data volume) and freshness monitoring, providing a safety net for "unknown unknowns."
6. **Reporting & Feedback:**
   * **Process:** Our custom Python application uses the Elementary SDK to fetch the latest results. It then stores the results in a file storage for future dashboards(Power BI or R+) to use.

**Benefits of This Approach**

Implementing this strategy will bring significant benefits:

* **Increased Data Trust:** Provides automated, verifiable proof of data quality, building confidence for both our team and our customer.
* **Faster Issue Resolution:** Proactive monitoring means we detect and fix data bugs before they impact downstream reports and analytics.
* **Clear Collaboration & Audit Trail:** The Data Rules Registry creates a transparent, auditable log of every data quality rule—what it is, who requested it, and whether it's implemented.
* **Improved Efficiency:** Automating the testing and reporting process frees up our data team from manual validation, allowing them to focus on higher-value tasks.
* **Scalable Framework:** This system is built on scalable components and can easily be extended to cover new tables, data sources, and more complex quality checks as the project grows.

**5. Steps**

1. **Phase 1 - Setup & Foundation:**
   * Establish the shared Data Rules Registry and onboard the customer.
   * Integrate the Elementary dbt package into our project.
   * Manually implement the first batch of Approved rules.
2. **Phase 2 - Automation & Integration:**
   * Develop and deploy the Python application for automated report generation.
   * Integrate the dbt test and reporting steps into our production Airflow DAG.
3. **Phase 3 - Enhancement:**
   * Deploy and share the Elementary web UI with stakeholders.
   * Explore creating a script to semi-automate the translation of Approved rules from the registry into dbt schema.yml format.

## Process

**Step 1: Customers Create Rules**

* **What happens:** Your customer has a business rule in mind (e.g., "Employee IDs must always be unique"). They log this request in a shared **Excel/Google Sheet or we enter it ourselves after meeting with the customer**. Data Engineer reviews it, and once approved, you translate it into code in a schema.yml file.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rule\_ID | Table\_Name | Column\_Name | Rule\_Description | Rule\_Type | Status |
| HR-001 | employees | employee\_id | Every employee must have a unique ID. | uniqueness | Approved |
| HR-002 | employees | employment\_status | Status must be Active, Terminated, or On Leave. | accepted\_values | Approved |

**Step 2: Schema Creation**

* **What happens:** The new test code (schema.yml) is saved in **Git**. This kicks off an automated **CI/CD pipeline** (like GitHub Actions) which prepares the code to be used in our live system.
* **Why we do this:**
  + **Git:** It gives us a complete history of every rule change. We can see who added what rule and when, providing a perfect audit trail.
  + **CI/CD:** This automates the deployment process, reducing human error and ensuring that approved rules are implemented consistently.

*# models/hr\_schema.yml*

models:

- name: employees

columns:

*# Translating rule HR-001*

- name: employee\_id

description: "The unique identifier for an employee."

tests:

- unique

- not\_null

*# Translating rule HR-002*

- name: employment\_status

description: "The current employment status of the employee."

tests:

- accepted\_values:

values: ['Active', 'Terminated', 'On Leave']

- name: annual\_salary

description: "The employee's annual salary."

tests:

*# A more advanced test to catch data entry errors*

- dbt\_expectations.expect\_column\_values\_to\_be\_between:

min\_value: 30000

max\_value: 1000000

**Step 3: Automation**

* **What happens:** Every day, **Airflow**, wakes up and runs our data pipeline. It tells **Snowflake** (our data warehouse) to first transform the raw HR data using **dbt (dbt run)**. Then, it tells dbt to test that data using the new rules we added **(dbt test)**.
* **Why we do this:**
  + **Airflow:** It's the "conductor" of our orchestra, ensuring every job runs in the right order and on schedule without manual intervention.
  + **Snowflake:** This is where our data lives and all the heavy lifting happens. It's fast and powerful.
  + **dbt:** It's the "chef" that cleans and prepares our data, making it ready for analysis and testing.

**Step 4: The Report Card (The Results Part)**

* **What happens:** During the dbt test step, **Elementary** is working in the background. It takes the pass/fail results from dbt, enriches them with more detail (like anomaly detection), and saves everything neatly in a results table inside a file storage system. A **Python script** then reads these results and creates a simple Markdown report and sends alerts for any failures.

**Step 5: The Feedback Loop**

* **What happens:** The customer and your team review the reports and alerts. This review often leads to new questions or ideas for more rules, starting the cycle all over again at Step 1.
* **Why we do this:** Data governance is not a one-time setup; it's a continuous process of improvement. This loop ensures our data quality rules evolve as the business evolves.

## Tool Selection

Elementary is better for us as its free, dbt naïve and useful for our needs.

* Both Elementary and Great Expectations are excellent tools, but they have different philosophies and are optimized for slightly different use cases.
* Given your specific stack (**dbt, Snowflake, Airflow**) and your goal of creating a tightly integrated governance system, here is a direct comparison to help you decide.

**The Core Difference in Philosophy**

* **Great Expectations (GX):** A comprehensive, standalone data quality *framework*. It can connect to anything (Spark, pandas, SQL) and manages its own configuration, state, and UI ("Data Docs"). It's powerful but requires you to learn and manage its ecosystem.
* **Elementary:** A dbt-native data *observability platform*. It is built specifically to live *inside* your dbt project. It leverages dbt's workflows and enhances them with powerful testing, lineage, and anomaly detection. It feels like a natural extension of dbt, not a separate tool.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Elementary** | **Great Expectations (GX)** |
| **dbt Integration** | **Native & Seamless.** It's a dbt package. Tests are defined in your existing schema.yml files. It feels like "dbt on steroids." | **Possible, but not native.** Requires a separate library (dbt-gx) and feels like you're orchestrating two different tools. GX has its own way of doing things. |
| **Setup & Learning Curve** | **Much lower for dbt users.** If you know dbt, you're 80% of the way there. Setup is adding a package and running a dbt command. | **Steeper.** You must learn the GX concepts: Data Context, Expectation Suites, Checkpoints, Data Docs, etc. The init command creates many new config files. |
| **Defining Tests** | In **YAML**, directly on your dbt models. Clean, simple, and version-controlled alongside your transformation code. | Primarily in **Python or JSON** in separate "Expectation Suite" files. More flexible for complex custom logic, but separates tests from your dbt models. |
| **Key Strengths** | - **Data Lineage:** Automatic, column-level lineage graphs. - **Anomaly Detection:** Out-of-the-box monitors for volume, freshness, schema changes. - **Central UI:** A single, shareable web app to view all test results and trends over time. | - **Massive Test Library:** A huge, mature library of "Expectations." - **Framework Agnostic:** Works with almost any data source, not just dbt. - **Detailed Reports:** "Data Docs" provide very detailed static HTML reports for each validation run. |
| **UI & Reporting** | An interactive, modern web application that shows historical trends and lineage. | Generates static HTML files ("Data Docs") for each validation run. Very detailed but can be hard to navigate historically. |

For your specific situation, **Elementary is the clear winner.**

Here's why:

1. **dbt:** Your entire transformation layer is in dbt. Elementary is designed from the ground up to integrate with this workflow. You won't be fighting to connect two different systems; you'll be enhancing the one you already have.
2. **Simplicity and Speed:** You will get up and running much faster. Defining tests in the YAML files you already use is far more efficient than building and managing separate Great Expectations suites.
3. **Observability over Static Reports:** Your goal is a continuous governance system. Elementary's UI, with its focus on trends and anomaly detection, is better suited for monitoring data health over time. Your customer can more easily explore the data lineage and understand the impact of issues.
4. **Collaboration:** While the rule intake (from the Excel file) is external to both tools, the output from Elementary (the shareable UI) is a better artifact to use in your collaborative review sessions with the customer than a folder full of static HTML files from Great Expectations.

## Test examples

1. **Layer 1: Built-in & Package Tests (The 80% Solution)**
2. **Layer 2: Custom SQL Tests (For Business Logic)**
3. **Layer 3: Anomaly Detection (For "Unknown Unknowns")**
4. **Layer 4: Python Models (For Ultimate Power)**

**Demonstration 1: Multi-Column Business Logic**

This is the most common type of complex check, where the validity of one column depends on another.

* **The Complex Rule:** "An employee's termination date, if it exists, must be after their start date. It's impossible to be terminated before you are hired."
* **The Elementary/dbt Solution:** You use a **custom dbt test**. This is a simple SQL file that returns the rows that *fail* the test. If it returns zero rows, the test passes. Elementary automatically picks this up.

**Step A: Write the SQL (tests/assert\_termination\_date\_after\_start\_date.sql)**

**select**

employee\_id,

start\_date,

termination\_date

**from**

{{ **ref**('employees') }}

**where**

termination\_date **is** **not** **null**

**and** termination\_date <= start\_date

**Step B: Attach the test in your schema.yml**

models:

- name: employees

columns:

- name: employee\_id

tests:

*# This custom test runs the SQL file we just created.*

- assert\_termination\_date\_after\_start\_date

**Result:** Elementary will run this SQL as a test, and if any "invalid" employees are found, it will fail and appear in your reports and alerts just like any other test.

**Demonstration 2: Cross-Table Logic**

This involves validating data in one table based on data in another.

* **The Complex Rule:** "An employee's salary must be within the min\_salary and max\_salary range defined for their department in the separate departments table."
* **The Elementary/dbt Solution:** This is another perfect use case for a **custom dbt test** that involves a JOIN.

**Step A: Write the SQL (tests/assert\_salary\_within\_department\_band.sql)**

**select**

e.employee\_id,

e.annual\_salary,

d.department\_name,

d.min\_salary,

d.max\_salary

**from**

{{ **ref**('employees') }} e

**join**

{{ **ref**('departments') }} d on e.department\_id = d.department\_id

**where**

e.annual\_salary < d.min\_salary

**or** e.annual\_salary > d.max\_salary

**Step B: Attach the test in your schema.yml**

models:

- name: employees

tests:

*# This test runs on the whole table, not just one column*

- assert\_salary\_within\_department\_band

**Result:** You are now enforcing complex business rules that span multiple tables, and Elementary will track the failures.

**Demonstration 3: Historical & Anomaly-Based Checks**

This is where Elementary's unique power shines. These are problems you can't solve with fixed rules.

* **The Complex Rule:** "The number of new employees hired each day should be normal. Alert me if we suddenly hire 10 times the usual number, or zero for three days straight, as this could signal either a data error or a major business event."
* **The Elementary Solution:** You don't write a test for this. You configure an **anomaly monitor**. Elementary automatically collects metrics over time, learns the normal patterns (including weekly seasonality), and alerts you on deviations.

**The configuration in schema.yml:**

models:

- name: employees

meta:

elementary:

timestamp\_column: 'created\_at' *# Tell Elementary when new records arrive*

tests:

*# This is not a pass/fail test, but a monitor.*

- elementary.volume\_anomalies:

tags: ['anomaly']

**Result:** Elementary's machine learning models will now monitor your hiring volume. You will get an alert if there's a statistically significant spike or drop, something a simple SQL test could never do effectively. This moves you from data validation to true data observability.

**Demonstration 4: Advanced Statistical Checks with Python**

For the most complex cases, you can use the dbt-Python model pattern.

* **The Complex Rule:** "Identify any employee whose bonus percentage is a statistical outlier (more than 3 standard deviations from the mean) for their job level and country, as this could indicate a data entry error or fraud."
* **The Elementary/dbt Solution:** Use a Python dbt model to find the outliers, and a simple dbt test to flag if any were found.

**Step A: The Python model (models/employees\_bonus\_outliers.py)** finds and outputs *only the failing rows*.

**import** snowflake.snowpark.functions **as** F

**def** model(dbt, session):

    """

    Identifies employees with outlier bonus percentages.

    Args:

        dbt: The dbt runtime object.

        session: The Snowpark session object.

    Returns:

        A Snowpark DataFrame containing only the rows of employees with outlier bonuses.

    """

*# 1. Configure the dbt model*

*# This tells dbt to materialize the output of this script as a table.*

    dbt.config(materialized="table")

*# 2. Get a reference to the upstream 'employees' table.*

*# This creates a Snowpark DataFrame from your existing dbt model.*

*# For this to work, you need an 'employees' model with 'job\_level', 'country',*

*# 'annual\_salary', and 'bonus\_percentage' columns.*

    employees\_df = dbt.**ref**("employees")

*# 3. Calculate the mean and standard deviation of 'bonus\_percentage' for each group.*

*# The peer group is defined by 'job\_level' and 'country'.*

    stats\_df = employees\_df.group\_by("job\_level", "country").agg(

        F.mean("bonus\_percentage").**alias**("mean\_bonus"),

        F.stddev("bonus\_percentage").**alias**("stddev\_bonus")

    )

*# 4. Join the calculated statistics back to the original employees DataFrame.*

*# Now, each employee row will have the mean and stddev for its specific peer group.*

    employees\_with\_stats\_df = employees\_df.**join**(stats\_df, on=["job\_level", "country"])

*# 5. Filter for and return ONLY the rows that are outliers.*

*# An outlier is defined as any value that is more than 3 standard deviations*

*# away from the mean for its group.*

    outliers\_df = employees\_with\_stats\_df.filter(

        F.abs(F.col("bonus\_percentage") - F.col("mean\_bonus")) > (3 \* F.col("stddev\_bonus"))

    ).**select**(

        "employee\_id",

        "job\_level",

        "country",

        "annual\_salary",

        "bonus\_percentage",

        "mean\_bonus",

        "stddev\_bonus"

    )

*# The returned DataFrame will be materialized as a table named 'employees\_bonus\_outliers'.*

*# If no outliers are found, this DataFrame will be empty.*

**return** outliers\_df

**Step B: The dbt test in schema.yml** checks if this output table is empty.

models:

- name: employees\_bonus\_outliers

description: "Python model to find statistical bonus outliers. This table should always be empty."

tests:

*# This test fails if the Python model found any outliers.*

- dbt\_utils.equal\_rowcount:

compare\_model: **ref**('empty\_table') *# A reference to an empty table*

## Data Observability

On every HR table (like employees or payroll\_changes), you want Elementary to:

1. Look at the \_extract\_timestamp column.
2. Find the most recent (max) timestamp in that column.
3. Calculate the time that has passed since that most recent timestamp and now.
4. Learn what a "normal" delay is (e.g., usually less than 24 hours).
5. Alert you if the delay becomes abnormally long (e.g., it's been 36 hours and there's no new data).

This is crucial for HR data. If the payroll\_changes table is stale, you could be running payroll on outdated information.

**How to Implement it with Elementary**

You enable this by adding two simple pieces of configuration to your schema.yml file.

**Step 1: Tell Elementary Which Column to Watch**

You use the meta block to tell Elementary that \_extract\_timestamp is your special timestamp column for monitoring.

**Step 2: Add the freshness\_anomalies Test**

You add a test from the Elementary package that specifically activates the freshness monitoring.

Here is the code snippet you would add to your hr\_schema.yml:

*# models/hr\_schema.yml*

models:

- name: employees

description: "The core model for employee data."

*# STEP 1: Tell Elementary to use your specific timestamp column.*

meta:

elementary:

timestamp\_column: '\_extract\_timestamp'

tests:

*# STEP 2: Activate the freshness anomaly detection test.*

*# This test will now use the column defined above.*

- elementary.freshness\_anomalies:

tags: ['anomaly']

*# Optional: you can define a specific time bucket if needed*

*# time\_bucket: { period: 'day', count: 1 }*

*# BONUS: This column also powers volume anomaly detection.*

*# This test alerts you if the number of new records per day is abnormal.*

- elementary.volume\_anomalies:

tags: ['anomaly']

columns:

- name: employee\_id

tests:

- unique

- not\_null

*# ... other column tests ...*

## Data Lineage

**Option 1: Use the Elementary Open Source Report**

This is the most direct and integrated solution. The free, open-source version of Elementary **can generate a full data observability report, including the interactive lineage graph**. It's not a cloud service, but rather a self-contained HTML file that you can host yourself or share with your team.

**How to generate it:**

After you've run dbt run and dbt test, you simply run one command from the Elementary CLI:

edr report

**What it creates:**

This command generates a target/elementary\_report.html file. When you open this file in a browser, you get a powerful, local web application that includes:

* **An interactive, column-level lineage graph** (exactly like the one in the paid version).
* All your data quality test results over time.
* Anomaly detection charts for freshness, volume, etc.

This is the easiest and most efficient way to get lineage because it's built directly into the tool you're already using.

**Option 2: Integrate with Marquez/OpenLineage (More Powerful, More Complex)**

This is a more advanced option for when you need a centralized, enterprise-wide lineage server that tracks more than just your dbt jobs.

* **OpenLineage:** This is an open standard (an API) for collecting lineage metadata from various tools (like Airflow, Spark, dbt, etc.).
* **Marquez:** This is an open-source tool that acts as the backend. It collects the data sent via the OpenLineage API and provides a UI to explore the lineage graph.

**How to integrate it with your current stack:**

1. **Set up Marquez:** You would need to deploy and manage your own Marquez instance (often using Docker or Kubernetes).
2. **Integrate with Airflow:** You would add the openlineage-airflow provider to your Airflow environment. This provider automatically "listens" to your DAG runs.
3. **Configure for dbt:** The OpenLineage integration can automatically parse the manifest.json file from your dbt runs to extract the lineage information.
4. **Visualize:** When your Airflow DAG runs dbt run, the OpenLineage integration sends metadata about the job and its lineage to your Marquez server in real-time. You can then log into the Marquez UI to see the lineage graph.