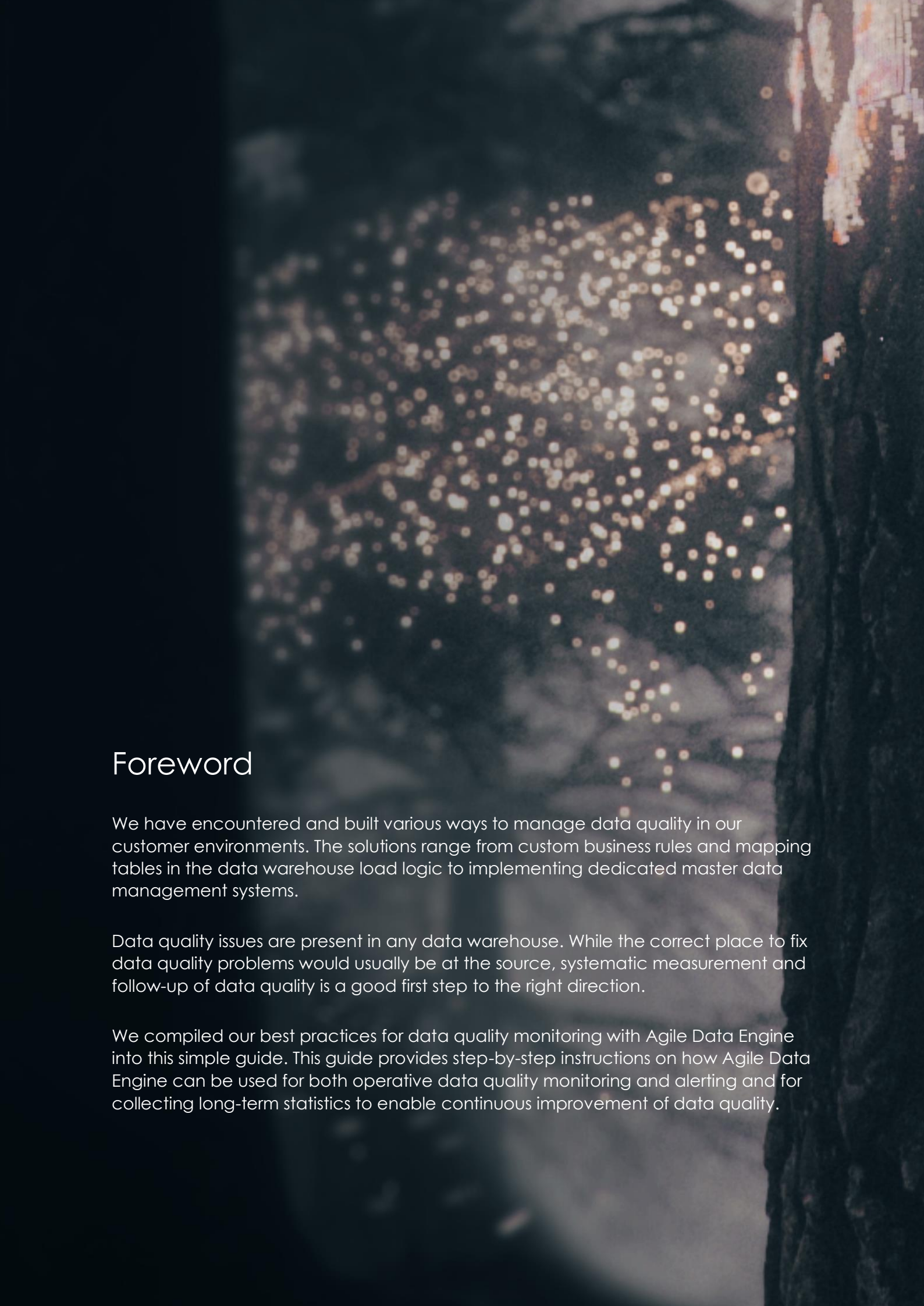

Data Quality
Monitoring with
Agile Data Engine
Agile Data
ENGINE

Table of Contents

Table of Contents 2
Foreword 3
How Do You Measure the Quality of Data? 4
Data Quality Dimensions 5
Monitoring Data Quality in a Data Warehouse 5
Typical Zones of a Data Warehouse 5
What to Test and Where? 6
Redundancy - an Extra Layer to the Monitoring of Data Quality 7
Data Quality Monitoring Outside the Data Warehouse 7
Short-term and Long-term Perspective 8
Short-term Operative Monitoring 9
Alerts Prompt Immediate Action 9
Long-term Monitoring 9
Collect Statistics for Continuous Improvement 10
Focus Your Data Quality Monitoring and Improvement Efforts on the Right Issues 10
Frequent and Critical - Eliminate Now 11
Frequent but Not Critical - Continuously Improve 11
Rare but Critical - Prepare in Advance 11
Rare and Not Critical - Follow Up Occasionally 12
In Practice 13
Smoke Tests for Operative Monitoring 14
Logging Data Quality Statistics for Long-term Monitoring 14
Examples for Setting Up Data Quality Monitoring with Agile Data Engine 15
Example 1: Logging the Quality of Incoming Data 15
Example 2: Using SMOKE_GREY to Alert for Bad Data Quality 18
Example 3: Using SMOKE_BLACK to Prevent Duplicates in the Publish Zone 19



Foreword

We have encountered and built various ways to manage data quality in our customer environments. The solutions range from custom business rules and mapping tables in the data warehouse load logic to implementing dedicated master data management systems.

Data quality issues are present in any data warehouse. While the correct place to fix data quality problems would usually be at the source, systematic measurement and follow-up of data quality is a good first step to the right direction.

We compiled our best practices for data quality monitoring with Agile Data Engine into this simple guide. This guide provides step-by-step instructions on how Agile Data Engine can be used for both operative data quality monitoring and alerting and for collecting long-term statistics to enable continuous improvement of data quality.

How Do You Measure the Quality of Data?

Before getting started, let's spend some time on data quality on a general level. We will go through data quality dimensions commonly referenced in the literature that provide you with the criteria to recognize and monitor issues with data quality. We will also discuss how testing these data quality dimensions could be applied in different data warehouse zones.

Data Quality Dimensions

You should first approach data quality management by defining appropriate dimensions for your use case. This set of data quality dimensions is the foundation for systematic data quality management. You can implement standardized tests and templates for testing similar data quality characteristics over different entities.

Finding a fitting set of dimensions depends on your use case and business requirements. And naturally, this typically evolves over time. Several reference frameworks can help you to get started. For example, UK Government Data Quality Hub lists the following data quality dimensions:

Dimension	Description
Accuracy	Is the data correct?
Completeness	Are the expected records and attribute values available and complete?
Uniqueness	Are there duplicates?
Consistency	Are there conflicting values within a record or across different data sets?
Timeliness	Is the data available when expected?
Validity	Does the data conform to the expected format, type, and range?

Monitoring Data Quality in a Data Warehouse

Typical Zones of a Data Warehouse

The next step is to define how you test the data quality dimensions in your data warehouse. Data warehouses contain zones/layers for different purposes. Usually, there are at least the following zones in a data warehouse:

Zone	Description
Staging	• Landing zone for incoming source data • Source file format specific flat tables where source files can be loaded • Attributes and data types matching the source data
Data warehouse	• Storage of data and its change history • Modeled with e.g. Data Vault methodology • If Data Vault is used, DW could consist of Raw Data Vault for data storage and Business Data Vault for applying business rules.
Publish	• End-use of data • Modeled according to the requirements of the use case • Often e.g. star schema for BI tools or flat tables for other purposes.

What to Test and Where?

You can run data quality tests in each zone of a data warehouse for somewhat different purposes. You could, for example, be interested in analyzing and comparing the quality of incoming data from different source systems or monitoring the status of your Publish models used for reporting.

This table gives examples of how you could test the different data quality dimensions in the data warehouse zones:

	Staging	Data warehouse	Publish
Accuracy	• Monitor the source system data accuracy • Option to discard incorrect data from the data warehouse	• Business-rule based tests	• Publish data accuracy • Business-rule based tests • Option to discard incorrect data from the Publish
Completeness	• Monitor the source system data completeness • Do expected attributes contain values e.g. data warehouse key attributes? • Are all records available?	• Referential integrity • Is expected data available for applying business rules? • Are values seen in transactional data sources also found in master data?	• Referential integrity • Is expected data available for end-use? • Option to discard incomplete data from the Publish
Uniqueness	• Monitor the source system duplicates	• Duplicate tests • Logical duplicates (e.g. these two customer ids should be the same customer)	• Duplicate tests • Option to discard duplicates from the Publish
Consistency	• Monitor the source system data consistency	• Business-rule based tests • Are values consistent with each other? • Are different datasets consistent (e.g. data from separate source systems)?	• Business-rule-based tests • Are values consistent with each other?
Timeliness	• Monitor that incoming data is loaded as expected • Timestamp-based testing to check that data is up to date • Pipeline lead time monitoring	• Pipeline lead time monitoring	• Pipeline lead time monitoring
Validity	• Monitor the source system data validity • Format validity tests • Option to discard invalid data from the data warehouse	• Monitoring validity of transformed data	• Monitoring validity of transformed data

Redundancy - an Extra Layer to the Monitoring of Data Quality

Business requirements should guide your decisions on how much data quality testing you do and in which zones these tests will be applied. As testing requires many different resources, it also has cost and performance implications.

Testing the same characteristics over different data warehouse zones might seem excessive. Still, the redundancy can give insight into how effectively data quality is improved when data gets transformed through the data warehouse. This quality improvement could also happen, for example, when you introduce cleaned-up data from a master data management system in the data warehouse zone. Multi-zone testing also often makes it easier to pinpoint root causes for data quality problems.

Data Quality Monitoring Outside the Data Warehouse

Typically, you would also build some monitoring outside of the data warehouse. For example, the solution could include monitoring source data integrations for the timeliness and completeness of incoming data or end-use data in the BI/other tools. When multiple components run tests with some overlap, you can catch critical issues even if one system is temporarily down.

Short-term and Long-term Perspective

Having data quality monitoring in place will allow you to trust the data flowing through your data platform and take action to improve it.

We will discuss the differences and suggest guidelines for implementing data quality monitoring. In addition, we will dive deeper into what to do with the data quality test results for different dimensions in the short term and long term.

Short-term Operative Monitoring

Operative monitoring aims to catch and eliminate recurring issues and transient errors that cause bad data quality. To achieve this goal, you must continuously improve your data pipelines to strengthen their robustness.

Operative monitoring keeps you informed about the current status of the data warehouse and its operations. From the data quality perspective, operative monitoring identifies critical data issues and alerts your DataOps team about them. An example of a critical error could be that the flow of source data has been interrupted, and no data has been received since a given threshold period, raising a data **timeliness** alert.

Alerts Prompt Immediate Action

Alerts keep your DataOps team on top of operative data quality monitoring efficiently. However, to ensure quick responses to critical data quality problems in your data platform, you must find a balance when assessing error criticality and creating new alert rules.

While creating new alerts is easy, remember that the data platform should only raise alerts for critical issues requiring immediate action. If there are too many false positives or alerts of unimportant issues, your DataOps team will lose interest and might miss critical errors. Unfortunately, it is common in data warehouse environments that numerous recurring alerts continuously bombard the alert channels, and no one responds.

Long-term Monitoring

With long-term monitoring, you collect data for long-term analysis and continuous improvement of data quality on your data platform instead of reacting to daily issues critical to your operations. Long-term data quality monitoring involves defining a comprehensive testing plan for your chosen data quality dimensions and collecting the test results over time.

For example, you could monitor the **completeness** and **validity** of customer data loaded from an operative source system by testing if it contains all the required attribute values and if the values are in the correct format. Typically, this could include checking details such as names, phone numbers, and email addresses. These can be important details but missing them is usually not critical enough for alerting. Note that your business requirements always define the criticality of specific information!

Collect Statistics for Continuous Improvement

All data quality issues cannot be critical enough to trigger alerts and immediate action. Also, not all issues can be fixed immediately as a response to an alert, regardless of how critical they are. Instead, issues of this type would require longer-term analysis and continuous improvement.

The root causes could, for example, be in a rigid workflow in an operative system where users have found various creative workarounds to get their work done. However, fixing this issue would require a heavily manual process since they originate from business processes that produce incomplete or invalid data.

The key to success here is to acknowledge that there are issues in the data, define tests and collect data quality statistics systematically over time. Once you have statistics, it is easier to identify the significant problems and analyze their root causes. Then, you can better focus your data quality improvement efforts and measure their effectiveness.

Focus Your Data Quality Monitoring and Improvement Efforts on the Right Issues

Not all data quality issues are equal. You should focus your efforts on the ones that impact your business the most.

We have defined a loose framework for categorizing data quality issues and focusing the monitoring and improvement efforts based on the issue's frequency and criticality.

	NOT CRITICAL	CRITICAL
FREQUENT	• Monitor and analyze • Improve with well-designed solutions • Potential future blockers if criticality increases	• Focus on decreasing the frequency and mitigating the criticality • Blockers for production environments • Development team in “survival mode”
RARE	• Not in focus • May become more critical with new use cases • May become more frequent with process changes	• Monitor with redundancy (from multiple perspectives) • Define alerts, respond immediately • Prepare for recovery scenarios

Frequent and Critical - Eliminate Now

Data quality issues in the *Eliminate* category are blockers for your production environment. Since these issues are critical, you likely have alerts configured that are triggered constantly due to the high frequency. Also, the business will have a hard time trusting the data.

Continuous production incidents are a significant burden for your DataOps team, which must be in constant survival mode to fix the problems. Also, focusing on development tasks will be difficult due to constant interruptions. As a result, they might favor quick fixes over well-designed improvements creating further technical debt and more problems in the future.

Your priority should be to eliminate such issues or to mitigate them by reducing the frequency, criticality, or both: a solution cannot be production-ready if constant critical data quality issues exist! That is why it is crucial to identify such blockers in the PoC or development phase and to mitigate them before moving to production.

Too often, PoC-grade solutions are used for production purposes because the demand for the solution is high. Also, it can still take quite a bit of effort to achieve production readiness after the PoC phase. DataOps products such as Agile Data Engine can reduce this burden by offering the needed technical capability from day one.

Frequent but Not Critical - Continuously Improve

Data quality issues in the *Continuously improve* category are common but not that critical for the business. These issues are good candidates for long-term monitoring and continuous improvement as they are not critical enough for operative alerts.

As always, seek well-designed solutions at the source of the issues. Having statistics available will help you analyze the data quality status and trends of your data platform, focus improvement efforts and monitor the effectiveness of improvements.

Note that new business cases could change the criticality of specific data sets! This type of issue might become a blocker in the future.

Rare but Critical - Prepare in Advance

Your DataOps team can *Prepare* for critical data quality issues that rarely occur. Alerts work best for this category. Alerts notify the team about an issue, even before

end users have detected it, and enable immediate response. In the best case, the team can fix the problem before end users notice it. Also, your DataOps team will take alerts seriously when there is no constant stream of alerts.

Preparing for different recovery scenarios and having the tools, documentation, and knowledge in place will help solve problems more quickly. In addition, creating some redundancy when configuring the monitoring and alerts will make it easier to pinpoint root causes and understand the scope of the effects of the problem. It will also ensure that the platform raises alerts even when one component of the platform is down.

Whenever new issues in this category are detected, evaluate if the issue could reoccur and if alerts could catch it. Then, define improvements to your pipelines, monitoring, and alerts accordingly. Note that there can always be issues so random in nature that predefined alert rules cannot catch them. That is where general preparedness to recover from such incidents comes in handy.

From the long-term monitoring perspective, you could follow the frequency of this type of issue and react if it increases. Such statistics might also help monitor the availability of your data platform.

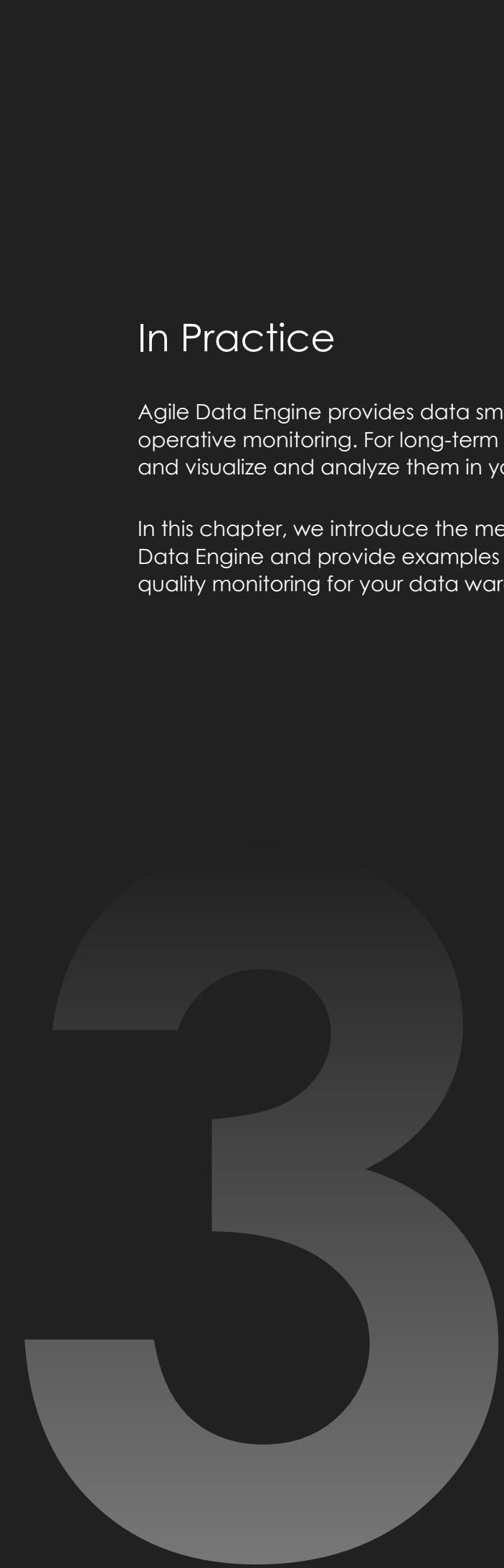
Rare and Not Critical - Follow Up Occasionally

Data quality issues in this category are not your primary concern and, therefore, not the focus of your DataOps team. However, as noted in the previous category, criticality can increase. Also, issue frequency can increase when you change the underlying processes. Thus, issues in the *Follow* category may become more significant later.

In Practice

Agile Data Engine provides data smoke tests to alert and control load workloads for operative monitoring. For long-term monitoring, you can log data quality statistics and visualize and analyze them in your favorite BI tool.

In this chapter, we introduce the mechanisms for data quality monitoring in Agile Data Engine and provide examples you can use as templates when setting up data quality monitoring for your data warehouse.



Smoke Tests for Operative Monitoring

Agile Data Engine enables you to run data smoke tests within load workflows to produce an alert when a test fails. Optionally, you can set smoke test results to control load workflows so that the dependent loads will be skipped if a test fails. This could be useful, e.g. in the staging zone when data quality is considered so bad that you do not want to load it into the data warehouse. Another typical use case is to avoid publishing incorrect data, such as duplicates in dimension tables.

In practice, you define smoke tests into entity loads as additional SQL load steps next to other load steps doing the actual loading of data. Alternatively, you can define smoke tests into separate loads, giving scheduling more freedom. The operating principle of smoke tests is simple: if the defined SQL query returns something, the test has failed.

Related load step types in Agile Data Engine:

Load step type	**Description**
SMOKE_GREY	SMOKE_GREY errors are logged and can be set to trigger alerts, but the execution of subsequent loads in the workflow will continue regardless of the result.
SMOKE_BLACK	SMOKE_BLACK errors are logged and can similarly be alerted. In addition, a failure will also cause the workflow to fail and stop any subsequent dependent loads from executing to prevent the errors from spreading.
(GATEKEEPER)	GATEKEEPER steps are not smoke tests but are worthy of mentioning here as they can be useful in some cases. A GATEKEEPER load step is used as a checkpoint to test if the execution of a load should proceed or stop. If the GATEKEEPER is fulfilled, execution of subsequent load steps will proceed. Opposite to smoke tests, a GATEKEEPER step is fulfilled when the SQL query returns a row/rows.

As smoke test load steps are SQL-based, the possibilities for customization are endless. You can make tests more generic and reusable with variables, save tests as templates, and set templates as default for selected entity types.

Logging Data Quality Statistics for Long-term Monitoring

You can design data quality logging and monitoring with Agile Data Engine like any other use case: You can create tables for storing logs of data quality tests and loads that run the tests and insert the results into the tables. This way, data quality statistics will be collected into your data warehouse and visualized and analyzed in any BI tool.

Like smoke tests, you can add data quality logging into entity loads as additional SQL load steps or define them as separate loads, giving more scheduling freedom.

Related load step types in Agile Data Engine:ß

Load step type	Description
PRE	PRE load steps are executed before generated load steps. Use PRE steps to store data quality statistics before the entity load is executed.
POST	POST load steps are executed after generated load steps. Use POST steps to store data quality statistics after the entity load is executed.
OVERRIDE	Use OVERRIDE load steps only if you have created a separate load for storing data quality statistics. Otherwise, an OVERRIDE load step will override the generated data load. You can also use OVERRIDE load steps if the entity load already uses OVERRIDE load steps. Then the order of the load steps will define the execution order.

Typically, you should run a smoke test and log data quality statistics for the same test case. Therefore, an approach used in some of our customer cases has been to define the test logic into views so that it is maintained in one place and is usable for both smoke tests and for logging long-term statistics. The examples in the next section will help you understand how this would work in practice.

Examples for Setting Up Data Quality Monitoring with Agile Data Engine

Example 1: Logging the Quality of Incoming Data

In the first example, we will create a few data quality tests for a staging load to monitor incoming data. These tests will be for logging purposes only, meaning we will not define smoke tests at this point.

Our sample table *stage.stg_taxi_zone_lookup* contains New York taxi zone data:

LOCATIONID	BOROUGH	ZONE	SERVICE_ZONE
1	EWR	Newark Airport	EWR
2	Queens	Jamaica Bay	Boro Zone
3	Bronx	Allerton/Pelham Gardens	Boro Zone
...	...	...	...

We will define the following data quality tests for the sample data:

- Completeness: Borough should not be blank.
- Validity: Zone should be minimum of 3 characters long.

Continuing the idea of having one place to maintain data quality tests, we will start by defining a view for the tests in Agile Data Engine:

Entity name	TAXIDATA_DQ_TESTS
Entity type	GENERIC
Physical type	VIEW
Schema	dq
Attributes	• test_timestamp - Test timestamp for logging purposes • target_schema - Schema of tested entity • target_entity_name - Name of tested entity • target_attribute_name - Tested attribute (or list of attributes) • test_type - E.g. data quality dimension or some other classification • test_description – Description • total_count - Total number of rows in the tested entity • error_count - Number of errors detected • threshold - For setting an error threshold if smoke testing (alerting) is needed later. E.g. maximum percentage of error cases. Note that these attributes are an example. Define your own format based on your requirements.
View SQL	-- stg_taxi_zone_lookup borough completeness SELECT current_timestamp AS test_timestamp, 'stage' AS target_schema, 'stg_taxi_zone_lookup' AS target_entity_name, 'borough' AS target_attribute_name, 'Completeness' AS test_type, 'Borough should not be blank' AS test_description, (SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup) AS total_count, (SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup WHERE trim(borough) = '' OR borough IS null) AS error_count, null AS threshold UNION ALL -- stg_taxi_zone_lookup zone validity SELECT current_timestamp AS test_timestamp, 'stage' AS target_schema, 'stg_taxi_zone_lookup' AS target_entity_name, 'zone' AS target_attribute_name, 'Validity' AS test_type, 'Zone should be minimum 3 characters' AS test_description, (SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup) AS total_count, (SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup WHERE LEN(zone) < 3 OR zone IS null) AS error_count, null AS threshold;

As seen in the view SQL, tests have been consolidated with UNION ALL. In a real-life production scenario, you could write tests into the source system, data warehouse zone, and/or domain-specific views to keep them better organized and avoid excess complexity.

Next, we will define a table for logging the test results:

Entity name	TAXIDATA_DQ_LOG
Entity type	GENERIC
Physical type	TABLE
Schema	dq
Attributes	Same as in the above view

Finally, we will define a POST load step into the file load of *stage.stg_taxi_zone_lookup*:

Load step name	insert_dq_log
Type	POST
Language	SQL
Logic	INSERT INTO dq.taxidata_dq_log SELECT * FROM dq.taxidata_dq_tests WHERE LOWER(target_schema) = LOWER('<target_schema>') AND LOWER(target_entity_name) = LOWER('<target_entity_name>')

The load step can be made generic and reusable using Agile Data Engine variables target_schema and target_entity_name. Since we have placed it into the same load where the actual data is loaded, data quality statistics will be logged whenever new data is loaded. Note that we assume that the stage is truncated before each file load. Otherwise, the tests would not be targeted to new records only. In other data warehouse zones, it might be more interesting to run tests on all records if feasible from the performance and cost standpoints. However, it is possible to target new batches only with Run ID Logic, Agile Data Engine's delta loading feature.

After a couple of test runs, we have the first log entries:

TEST_TIMESTAMP	TARGET_SCHEMA	TARGET_ENTITY_NAME	TARGET_ATTRIBUTE_NAME	TEST_TYPE	TEST_DESCRIPTION	TOTAL_COUNT	ERROR_COUNT	THRESHOLD
2022-02-28 15:03:17	stage	stg_taxi_zone_lookup	borough	Completeness	Borough should not be blank	265	0	null
2022-02-28 15:03:17	stage	stg_taxi_zone_lookup	zone	Validity	Zone should be minimum 3 characters	265	2	null
2022-02-28 15:15:23	stage	stg_taxi_zone_lookup	borough	Completeness	Borough should not be blank	265	0	null
2022-02-28 15:15:23	stage	stg_taxi_zone_lookup	zone	Validity	Zone should be minimum 3 characters	265	2	null

This data can now be visualized in a BI tool for further analysis and monitoring purposes.

Example 2: Using SMOKE_GREY to Alert for Bad Data Quality

We will continue working with the sample data and set up a SMOKE_GREY test to alert for bad data quality.

Let's go back to the data quality view created in the first example and set an error threshold for the validity test:

View SQL

```
-- stg_taxi_zone_lookup zone validity
SELECT
CURRENT_TIMESTAMP AS test_timestamp,
'stage' AS target_schema,
'stg_taxi_zone_lookup' AS target_entity_name,
'zone' AS target_attribute_name,
'Validity' AS test_type,
'Zone should be minimum 3 characters' AS test_description,
(SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup) AS total_count,
(SELECT COUNT(*) FROM stage.stg_taxi_zone_lookup WHERE LEN(zone) < 3 OR zone IS null)
AS error_count,
0.10 AS threshold;
```

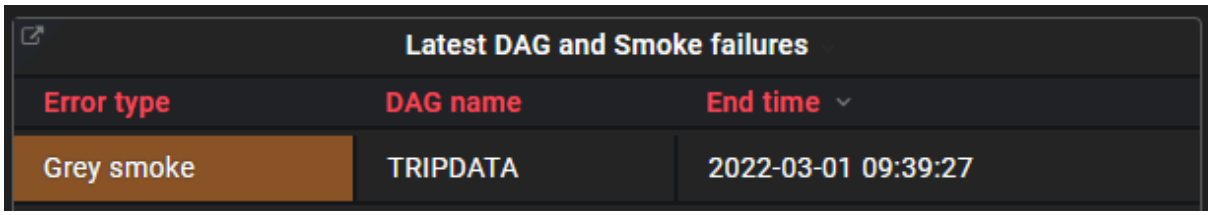
We set the threshold to 0.10, meaning that we want to be alerted if 10% of the records or more are invalid.

To enable alerting, we will configure a SMOKE_GREY step into the file load of *stage.stg_taxi_zone_lookup* querying the data quality view.

Load step name	validity_smoke
Type	SMOKE_GREY
Language	SQL
Logic	SELECT 1 FROM dq.taxidata_dq_tests WHERE LOWER(target_schema) = LOWER('<target_schema>') AND LOWER(target_entity_name) = LOWER('<target_entity_name>') AND test_type = 'Validity' AND (CASE WHEN total_count > 0 THEN error_count/total_count >= threshold ELSE false END)

We are again using variables to make the step more generic and reusable by selecting the validity test and avoiding division by zero if there are no records to test.

Finally, if the smoke test fails, the error will be visible in the Runtime Dashboard, triggering a notification through the selected channels, e.g. Slack, Teams, or email.



Example 3: Using SMOKE_BLACK to Prevent Duplicates in the Publish Zone

The third example is how to prevent duplicates in Publish. This time, we aim to stop a load from updating the Publish entity and trigger an alert if duplicates exist.

Below is a complete load template often used for Publish entities in our Snowflake customer cases. You can implement similar logic for other target databases just as easily. The main idea is to prepare a refreshed version of the entity in a temporary table and then swap it with the original. In this way, the entity is always available for end users, and it is also possible to test the entity for critical errors before running the swap operation.

Note that all load steps are not entity-specific except steps 3 and 5.

#	Load step name	Type	Logic
1	drop_tmp_if_exists	OVERRIDE	DROP TABLE IF EXISTS <target_schema>.z_tmp_<target_entity_name>
2	create_tmp_like_original	OVERRIDE	CREATE TABLE <target_schema>.z_tmp_<target_entity_name> LIKE <target_schema>.<target_entity_name>
3	load_into_tmp	OVERRIDE	INSERT INTO <target_schema>.z_tmp_<target_entity_name> -- input entity specific load logic here --
4	prevent_empty_table	GATEKEEPER	SELECT COUNT(*) FROM <target_schema>.z_tmp_<target_entity_name> HAVING COUNT(*) > 0
5	duplicate_test	SMOKE_BLACK	-- Define the key attribute over which duplicates are checked SELECT 1 FROM <target_schema>.z_tmp_<target_entity_name> GROUP BY key HAVING COUNT(*) > 1 LIMIT 1
6	swap_tmp_with_original	OVERRIDE	ALTER TABLE <target_schema>.z_tmp_<target_entity_name> SWAP WITH <target_schema>.<target_entity_name>
7	drop_tmp	OVERRIDE	DROP TABLE IF EXISTS <target_schema>.z_tmp_<target_entity_name>

Step 3 contains the actual entity load logic, which naturally is entity-specific.

The GATEKEEPER in step 4 stops the process if there is no data. GATEKEEPER steps will not produce alerts. If necessary, you could replace this step with another SMOKE_BLACK test.

Step 5 is the duplicate test as a SMOKE_BLACK load step. This step can also be generalized if the key naming in Publish is standardized. You can save load logic in Agile Data Engine and reuse it for other entities.

If a SMOKE_BLACK test fails, it will also fail the workflow (DAG), and subsequent dependent loads will not be run. This gives the DataOps team some time to fix the issues. Again, the test will trigger notifications through the selected channels, e.g., Slack, Teams, or email.