DATA ENGINEERING · GOVERNANCE BRIEF

WHEN DATA CHANGES MEANING

Preparing changes to a data flow across applications

Published on 10 November 2026 · Version 1.3.0 · Updated 10 November 2026
Online article:
https://datanalysis.re/ressources/data-engineering/data-changes-meaning-across-applications

An update can be deployed correctly, jobs can finish on time, and the amounts finance relies on can still become wrong. Before changing a shared piece of data, you therefore need to check what the applications receiving it will make of it. Here is a method for organising that review, with a worksheet to fill in for one specific data flow.

Fictional example: same field, same type, different unit

Before · invoice_total	After · invoice_total
48.50 euros	**4850** cents

Read as euros: €4,850, a hundred times too high

Normalised: 4850 ÷ 100 = €48.50

A valid number can be misread

Take a fictional example. An invoicing application sends the total amount of an invoice, including VAT, in the `invoice_total` field. For an invoice of 48.50 euros, it sends the numeric value `48.50`. After a change, it sends `4850`, now expressed in cents. The field name is unchanged and the schema still accepts a number.

A check that only verifies that the field is present and has the right type may accept both values. Yet if the next process keeps reading euros, it gets 4,850 euros. In this example, the interpreted amount is a hundred times too high. The EUR currency code is not enough to remove the ambiguity: you also need to know the unit of representation.

A reconciliation with the source invoice or a well-chosen business rule can detect this gap. The problem is the scope of the check that actually runs. A technical “completed” status tells us that the job ran; whether the amount is correct also depends on the checks that were planned.

The dbt documentation describes this difficulty: a calculation can change the results of a column while keeping its name and type. The team that maintains the model has to assess the consequences for its users. [1]

Find every use before changing the field

To prepare the change, we suggest starting from one piece of data and following its path. Who produces it? Which processes transform it? Which applications, exports or decisions depend on it? A dependency view is a good starting point, but its coverage has to be checked. A file that is exported and then reworked by hand can escape the technical map.

In our example, imagine two consumers: a financial monitoring dashboard and a collections export. The first displays the totals; the second is used to prepare a list of invoices to process. The change has to be tested in both uses. Validating only that the data arrives in the warehouse would leave the final reading unchecked.

For each consumer, record an owner, the process to adapt and the expected evidence. The owner of the source application confirms the new representation. The integration team checks the transformation. The owner of each use confirms the result obtained. One person leads the transition and knows which points remain open.

This list should stay proportionate to the flow. For rarely used data, a few lines may be enough. For an amount distributed to several systems, a dependency without an owner should be resolved before the cutover is scheduled.

Make the definition and expectations explicit

A data contract is an agreement between those who produce a piece of data and those who use it. Confluent distinguishes within this agreement the structure, constraints, metadata, rules and evolution. [2] For our review, we keep the information needed to decide whether the change is acceptable.

Points to specify, applied to the fictional example

Point to specify	Applied to our fictional example
Business definition	Total invoice amount including VAT, one invoice per record.
Unit and currency	Euros before the change; euro cents after.
Special cases	Negative credit notes, zero amount, cancellation and missing value handled according to explicit rules.
Versions and period	Version received, effective date and handling of historical invoices.
Consumers	Financial monitoring and collections export, each with an owner.
Checks and response	Reconciliation with the source invoices; stop or set aside the flow if a gap is unexplained.

This content can be kept in a repository of definitions, a code repository or a versioned specification. Its location must allow the teams involved to find it and review the change. The document then has to translate into checks that run and sign-offs that are assigned. A written rule that never takes part in acceptance testing or monitoring detects no gap.

The Open Data Contract Standard also distinguishes rules described in text from executable rules, for example in SQL or with a data quality engine. [3] This distinction helps show what is already checked and what remains to be implemented.

Test the result in a common unit

A useful test compares values that have the same meaning. If the new value is `4850` cents, its normalised amount is `48.50` euros. That is the amount to compare with the reference in euros. Comparing the two raw numbers directly would produce an expected alert without checking the conversion.

Choose a small set of invoices whose expected results are known. Include the cases the business allows: an ordinary invoice, a credit note, a zero amount and the relevant cancellation or missing-value situations. For each case, keep the input data, the version used, the expected result and the result observed in each consumer.

Checks and expected results in the fictional example

Check	Expected result in our example
Invoice of 48.50 euros	`4850` cents becomes `48.50` euros after normalisation.
Credit note of 12 euros	If credit notes are negative, `-1200` becomes `-12.00` euros.
Missing unit or unknown version	The process applies the agreed rule and flags the case; it does not guess the unit.
Same batch of invoices	The normalised total matches the reference, with a documented rounding rule.
Replay of a batch	The result contains no additional duplicate.

The total is useful, but it can hide errors that cancel each other out. Also check the rows, their count and their identity. A tolerance should have a business reason, for example a defined rounding rule; a threshold chosen only to make acceptance pass makes the check uninformative.

For amounts, also specify where rounding applies and how many decimal places are kept. The same calculation performed per row or after aggregation can give different results. This choice must be shared by the consumers concerned.

Choose the transition according to dependencies

If all consumers can be adapted together, a coordinated cutover may be suitable. You then need to define the sequence of work, the post-deployment tests and the decision to take if one of them fails. A short window does not remove the need to know who can stop the process.

If teams have different schedules, a period of coexistence may be necessary. In our example, the source could expose a new version or a new field whose unit is explicit. Consumers then migrate on a known schedule. Avoid letting the same piece of data become “euros” and then “cents” with no reliable way to tell the two representations apart.

For its versioned models, dbt lets you keep parallel versions and set a deprecation date. Its documentation also points out the cost of this coexistence. [1] The principle is useful beyond this tool: each version kept must have identified users and a condition for removal. Multiplying copies without a schedule simply moves the problem.

The transition must also deal with history. Do invoices already recorded stay in the old representation? Will they be converted? How do you recognise an old batch replayed after the cutover? A processing date alone may be insufficient if the batch contains older data. An explicit version, or another unambiguous rule, must guide interpretation.

Prepare recovery before giving the go-ahead

Rollback has to cover the code and the data already processed. Putting an old process back into service does not automatically correct the amounts distributed during the incident. Define the batch concerned, the outputs to suspend and how to recalculate or correct the results without creating duplicates.

Before the cutover, the team must be able to answer a concrete question: if the check on an invoice fails, which action stops the propagation and who triggers it? Depending on the flow, this may be a hold, a data quarantine or stopping the consumer. The answer depends on the cost of incorrect data and the cost of an interruption.

Also set the post-deployment checkpoint. The owners of each use must confirm their results on the same data scope. Closing the change can then rely on identified evidence, and removing the old version on the absence of consumers that still depend on it.

Prepare your next change review

The accompanying worksheet follows this approach: a completed fictional example, then a grid to document your own flow. It does not replace test results: it shows what must be checked, by whom, and where to find the evidence.

Start with a piece of data whose change is already planned. Bring together the source team and the owners of each use. If you cannot yet name a consumer, its expected unit or its owner, you have found a point to resolve before delivery.

Are you changing data used by several applications? Let's discuss one specific flow to scope its dependencies, the checks to plan and the transition.

→ Discuss my data flow with DATANALYSIS: https://datanalysis.re/contact?lang=en

→ Fillable worksheet (PDF): https://datanalysis.re/downloads/data-n2-fiche-remplissable.en.pdf

→ HTML worksheet: https://datanalysis.re/ressources/data-engineering/data-changes-meaning-across-applications/worksheet

Related expertise

Data Engineering (available in French): https://datanalysis.re/expertises/data-engineering?lang=fr

Data Governance (available in French): https://datanalysis.re/expertises/data-governance?lang=fr

Sources

Sources rechecked on 10 November 2026. The method and examples are for teaching purposes; no client results are presented.

[1] dbt Labs | Model versions (calculation change without a change of name or type; versions and deprecation)
https://docs.getdbt.com/docs/mesh/govern/model-versions

[2] Confluent | Data Contracts for Schema Registry (structure, rules, metadata, evolution)
https://docs.confluent.io/cloud/current/sr/fundamentals/data-contracts.html

[3] Bitol | Open Data Contract Standard, Data Quality, version 3.2.0 (textual description versus executable check)
https://bitol-io.github.io/open-data-contract-standard/v3.2.0/data-quality/

Fictional teaching example; no client results.