Open in app ↗ Sign up Sign In

Published in Adevinta Tech Blog

Xavier Gumara Rigol Follow

Sep 21, 2021 · 9 min read · Listen

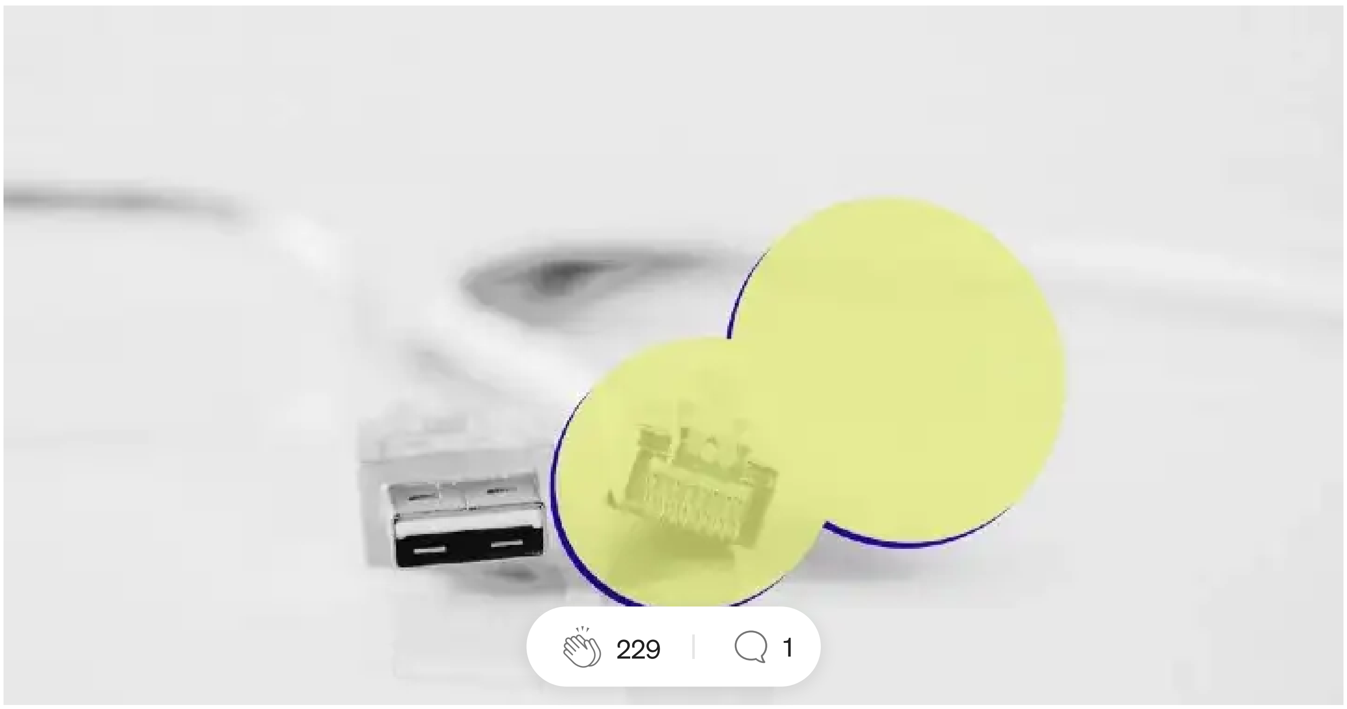
Save

Treating data as a product at Adevinta

A deep dive into the logical and technical implementation of the data mesh principle “data as a product”

In a previous post, Building a data mesh to support an ecosystem of data products at Adevinta, we explained our journey towards a federated data architecture for analytics (or something very similar to a data mesh) in the Global Capabilities teams at Adevinta.

In this article, we’ll deep dive into our journey towards treating “data as a product”, one of the principles of the data mesh paradigm. We’ll explain why we did it, where we are in the implementation and will give you a concrete example.



229 1

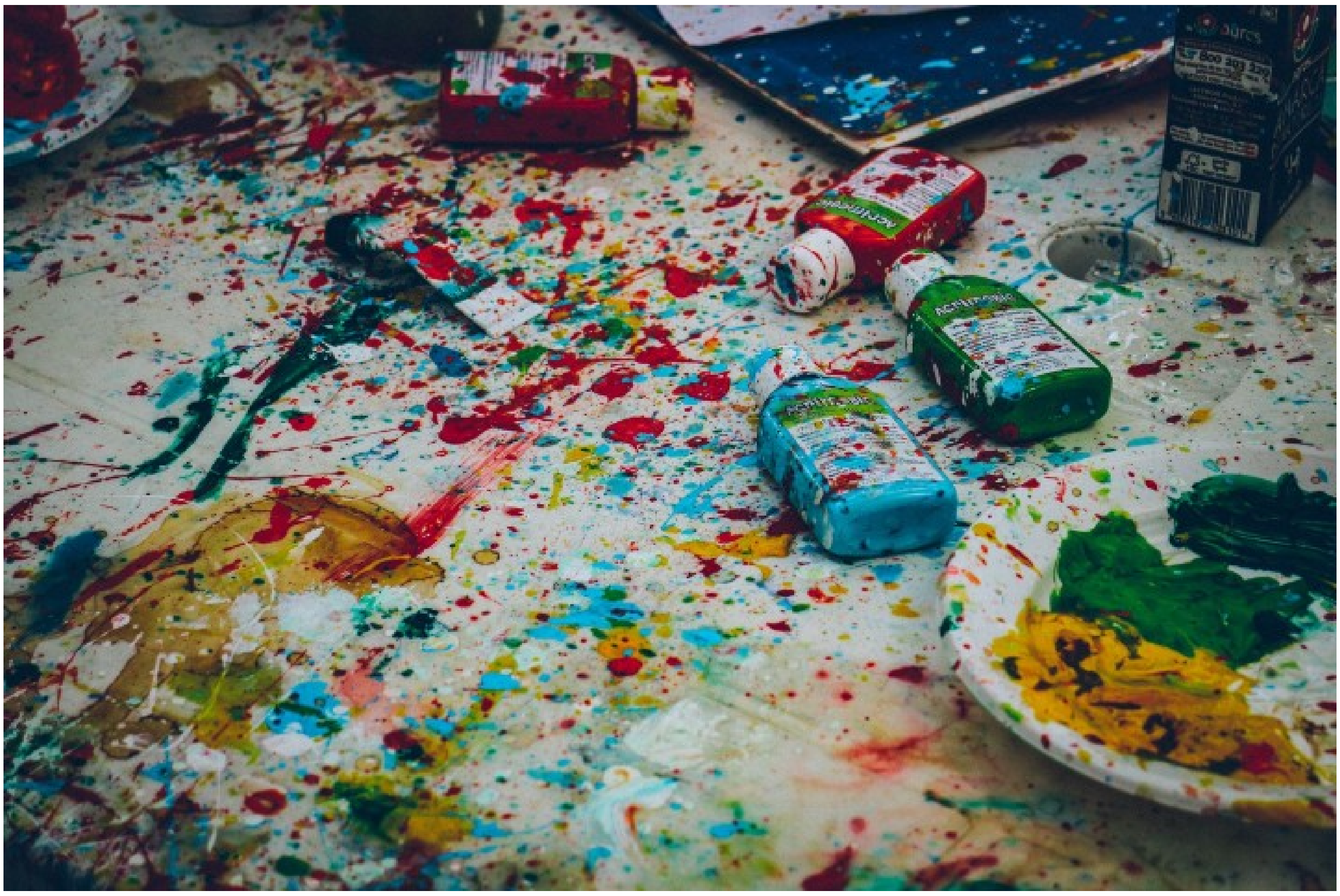
Before we start, here are a few considerations:

- Our journey of decentralising our architecture for analytics began before the data mesh concept was born. At that time, we didn't have the words for it and because of that, some aspects don't follow what the book advises.
- If you're new to the concept of data as a product, we recommend reading the data as a product principle first, from the original data mesh articles and also the article: Data as a product vs data products. What are the differences?.
- Applying product thinking to datasets is valuable to any data project (data warehousing, recommender systems, experimentation platforms,…) and is independent of other data mesh principles. You can apply product thinking without modifying the operating model your analytics data architecture sits on.
- The purpose of the datasets described in this blog post is purely analytical and belongs to Adevinta's Global Capabilities teams that provide a wide range of data products that our marketplaces can leverage.

Benefits observed

The following situations reflect what it was like to work with analytical data before applying product thinking to it:

- When coworkers used to say "give me access to data", they were provided with a connection to a database (always depending on a human) full of tables that lacked up-to-date documentation. This led to analysts being used to having to go back to the raw data, clean it and transform it themselves, which resulted in a disparity of conclusions in analyses.
- The code to generate all the datasets was written under the same repository. Controlling different versions, updates and dependencies became unmanageable once the amount of pipelines grew past a dozen.
- There was a perception that the quality of datasets wasn't guaranteed because there wasn't any available information about them.
- Knowledge sharing around data, datasets and its attributes was tribal (or Slack-based).
- We had heterogeneous semantics within domains and no aligned metrics.



A big mess can happen when product thinking isn't applied to datasets. Photo by Alice Dietrich from Unsplash

Below are the changes in behaviour that we identified when treating data as a product:

- Each dataset has up-to-date metadata and is accessible via a web interface where anyone can request and get secure access to it; so no more human or Slack-based access management.
- Each data as a product has its own repository of code isolated from the rest, which allows more control and domain ownership.
- Analysts have a single point of entry and landing page when it comes to getting acquainted with the most important datasets of the company and their domain.
- Domain-specific customers now trust their domain-specific datasets because they are the product owners of them. Also, since they see the benefits of data, they're now more prone to fix quality issues at the source systems (operational/transactional).
- The return on investment of analytics has increased: datasets are more widely used (13% increase in weekly querying users), customers are more efficient

(time to analyse A/B tests has decreased) and they can dedicate more time to product discovery and run more A/B tests (7% increase in the number of experiments created by domains teams).

All these benefits have been achieved by putting in place the right processes. Obviously you need to have good technologies to make sure query time is low and access management is self-served, but making sure every dataset is properly documented and decision-makers are provided with a consistent implementation across domains is more important in our view.

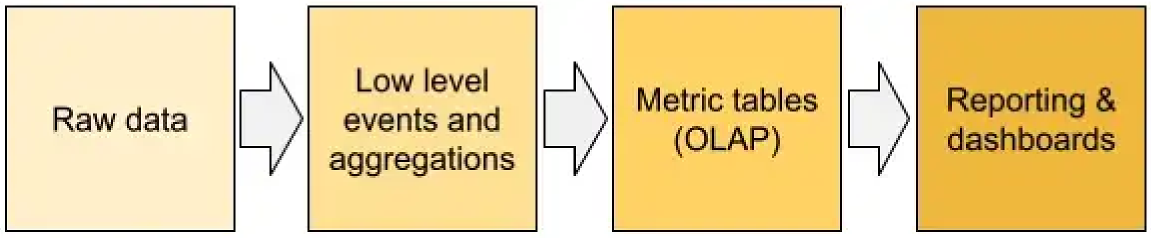
Logical implementation

At Adevinta, inside a domain-data product we differentiate different servings of the data, which we call layers. We've standardised the use of these logical layers for all the domains so customers have the same experience when using data from different domains.

A common characteristic is that each layer only uses data from datasets in the previous layer. The union of all these layered datasets makes up our domain data product.

For people familiar with other analytics architectures, this layered approach is very similar to the data warehouse logical architecture as understood and explained by Bill Inmon in his book *Building the Data Warehouse*.

Here's a diagram of the different layers, which are explained in the sections below.



Logical layers of data

Raw events

The first serving of data is the **raw events** shared both in batch and real-time. These events aren't cleaned, they contain lots of attributes with raw values and are used for multiple purposes in the organisation.

This is the layer we want the analytics customers to avoid because data is too raw.

Low-level events and aggregations

The second serving is the batch **datasets with low-level events and aggregations** that contain the source of facts for the domain data product. These facts are obtained by filtering the raw data with the events and attributes that are relevant for the domain. Some transformations and mappings are also applied.

Besides the filtered and cleaned events, some low-level aggregations also belong to this layer; for example, a summarised daily/weekly/monthly activity table by user id.

As the datasets in this layer contain Personal Identifiable Information (PII) and due to privacy regulations, data takeout and data deletions services are implemented in this layer.

Datasets in this layer are never used in long-lived reports and visualisations. They're used instead as a source for metrics generation (next layer) and for exploratory analyses via notebooks or SQL directly.

Metric tables

The third serving is the batch **metric tables**, very similar to the concept of dimensional data marts in Inmon's architecture. In this layer, datasets are modeled using dimensional modeling and they materialise thousands of metrics that are used for dashboarding.

No PII is stored in this layer (because of the aggregations) and we usually have a metric table per domain.

Indicator layer

Lastly, we have an **indicator** layer that provides reports and dashboards using the calculations provided in the metric tables.

Different servings of the data are provided depending on the job to be done. Photo by Yehor Milohrodskyi from Unsplash.

Physical implementation

Note: we use Amazon Web Services (AWS) and Tableau at the moment, so some technical decisions have been made with these choices in mind.

Storage

We store **raw events, low-level events and aggregated** batch datasets in S3. The metadata of the datasets is retrieved by the Glue Data Catalogue and is loaded later by Athena to allow interactive queries using SQL. The setup is perfect to perform exploratory analysis via notebooks or SQL.

For the next layer, we store the **metric tables** in Redshift because of the ease of manually populating metric tables with historical data or modifying some metric values in Redshift compared to S3 (although we're transitioning to delta and we'll need to review these assumptions).

For each metric table, we also create a Tableau Server data source that makes data available in Tableau for dashboarding without having to worry about the underlying storage system. We use live connections to the table itself so there are no business transformations done between the metric table stored in Redshift and what you see exploring the Tableau data source.

Infrastructure

The necessary pipelines to build the domain datasets as products run on Kubernetes. Our Common Runtime Environment (CRE) team takes care of managing this piece of infrastructure. Luckily they've written an awesome blog post that explains all the details of the platform if you're interested: Introducing Unicron, our big data and Machine Learning platform.

Code

As of today, within all the domains combined, we have more than 50 pipelines that generate data for more than 30 datasets (some pipelines append data to the same dataset, as in the Messaging case explained in the next section). Pipelines are built using Scala and we manage the dependencies between them with code (we like to call it *orchestration as code*).

Going through the code of a pipeline deserves a post on its own so we'll save that for another time.

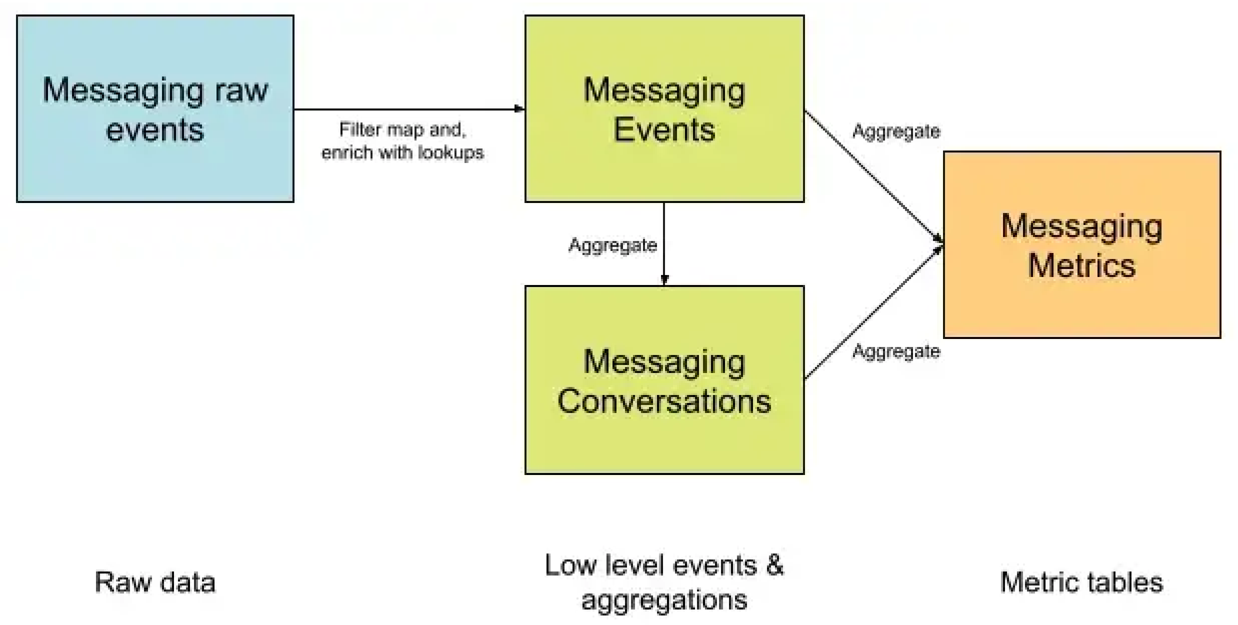
A real life example: Messaging

In order to better understand our take on data as a product, here's an example using one of our domains: Messaging. The messaging domain handles the metadata of the communication between any two users in our marketplaces (not the content of the messages).

Let's get our analysts' hands dirty by taking a look at an example. Photo by Alice Dietrich from Unsplash

We have a specific team owning the product, therefore it's easy for them to control the analytical data that the domain generates. That doesn't mean that the transition to decentralise the ownership has been easy (more on the challenges involved in our first blog post).

In the following diagram we can see the logical split of the domain data product:



Logical split of the analytical datasets for the Messaging data as a product

The Messaging product sends an event to the data platform for every change and user interaction with the product. From a logical perspective, these events land in the "raw data" layer and can be used for multiple purposes.

From there, we apply some enrichments and lookups and materialise them again, forming the **Messaging Events** dataset.

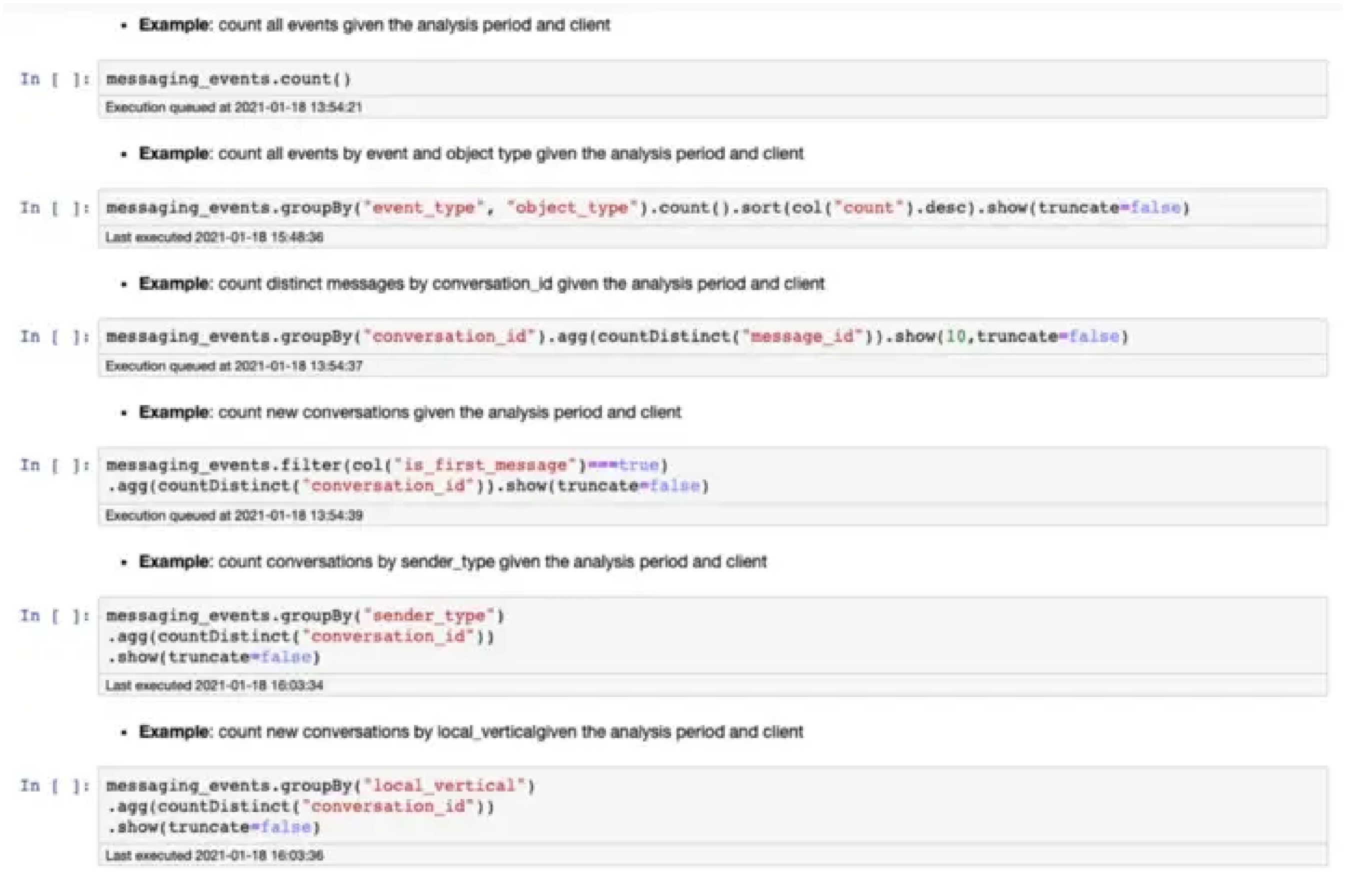
As you can see in the diagram, the **Events Messaging** dataset is the source for the **Messaging Conversations** dataset; a dataset that contains aggregated information about the conversation between two users.

These two datasets are used by our customers for exploratory analyses and for training Machine Learning models, but also to generate metrics for reporting and dashboarding in the next layer.

This final dataset with the metrics is called **Messaging Metrics.** This dataset is generated using the cube() function in Spark that allows us to materialise multidimensional aggregations, making it easy to explore the data in Tableau.

To help analysts understand the domain data product, documentation and examples (notebook and SQL queries) are provided.

See below a screenshot of an example notebook that is provided to analysts for this domain:



- **Example**: count all events given the analysis period and client

```
In [ ]: messaging_events.count()
        Execution queued at 2021-01-18 13:54:21
```

- **Example**: count all events by event and object type given the analysis period and client

```
In [ ]: messaging_events.groupBy("event_type", "object_type").count().sort(col("count").desc).show(truncate=false)
        Last executed 2021-01-18 15:48:36
```

- **Example**: count distinct messages by conversation_id given the analysis period and client

```
In [ ]: messaging_events.groupBy("conversation_id").agg(countDistinct("message_id")).show(10,truncate=false)
        Execution queued at 2021-01-18 13:54:37
```

- **Example**: count new conversations given the analysis period and client

```
In [ ]: messaging_events.filter(col("is_first_message")===true)
        .agg(countDistinct("conversation_id")).show(truncate=false)
        Execution queued at 2021-01-18 13:54:39
```

- **Example**: count conversations by sender_type given the analysis period and client

```
In [ ]: messaging_events.groupBy("sender_type")
        .agg(countDistinct("conversation_id"))
        .show(truncate=false)
        Last executed 2021-01-18 16:03:34
```

- **Example**: count new conversations by local_verticalgiven the analysis period and client

```
In [ ]: messaging_events.groupBy("local_vertical")
        .agg(countDistinct("conversation_id"))
        .show(truncate=false)
        Last executed 2021-01-18 16:03:36
```

Example of queries that can be executed on top of the Messaging Events dataset

Final words

We’ve said this before but it’s worth reinforcing to close this article: the processes and policies you implement in your organisation are far more important than the technology you choose in order to deliver datasets as products.

We wouldn’t be treating data as a product if we hadn’t invested early in people that have good data modelling and product skills.

On one hand, Business Intelligence, Analysts/Engineers, Data Modelers and (more generally) people that are used to translate business requirements for a dashboard into a data model, have been key to succeed in our journey of treating data as a product.

On the other hand, incorporating a Product Manager in a Business Intelligence project can help provide product thinking to the team. An example of product thinking would be to update the “definition of done” and the “acceptance tests” of the user stories that generate analytical datasets to be more customer centric.

When we put the right people at the right seat, they bring the right processes and policies into the mix.

Data Mesh Data Product Management Data Architecture Business Intelligence

Data And Machine Learning