Governing data products using fitness functions

Decentralized data management requires automation to scale governance effectively. Fitness functions are a powerful automated governance technique we've applied to data products within the context of a Data Mesh. Since data products serve as the foundational building blocks (architectural quanta) of a data mesh, ensuring robust governance around them significantly increases the chances of a successful data mesh transformation. In this article, we'll explore how to implement this technique. We'll start by designing simple tests to assess key architectural characteristics of a data product, and then we'll explore how to automate their execution by leveraging metadata about the data products.

05 September 2024

Kiran Prakash

Kiran is a principal engineer at Thoughtworks with a focus on data. An avid practitioner of extreme programming, he is passionate about microservices, platform modernization, data engineering, and data strategy. As a senior leader in the Data and AI service line, he helps large, strategic clients in leveraging data to achieve business success

CONTENTS

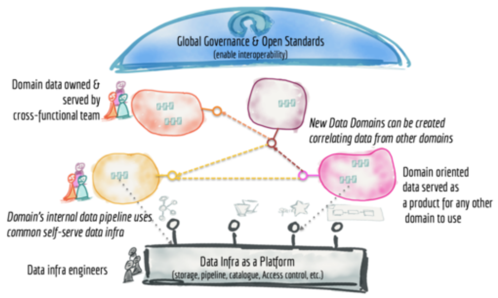
DATA MESH

The key idea behind data mesh is to improve data management in large organizations by decentralizing ownership of analytical data. Instead of a central team managing all analytical data, smaller autonomous domain-aligned teams own their respective data products. This setup allows for these teams to be responsive to evolving business

Table of Contents

needs and effectively apply their domain knowledge towards data driven decision making.

How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh



Many enterprises are investing in their next generation data lake, with the hope of democratizing data at scale to provide business insights and ultimately make automated intelligent decisions. Data platforms based on the data lake architecture have common failure modes that lead to unfulfilled promises at scale. To address these failure modes we need to shift from the centralized paradigm of a lake, or its predecessor data warehouse. We need to shift to a paradigm that draws from modern distributed architecture: considering domains as the first class concern, applying platform thinking to create self-serve data infrastructure, and treating data as a product.

by Zhamak Dehghani ARTICLE 20 May 2019

Read more...

ENTERPRISE ARCHITECTURE DATA ANALYTICS DOMAIN DRIVEN DESIGN DATA MESH

Having smaller autonomous teams presents different sets of governance challenges compared to having a central team managing all of analytical data in a central data platform. Traditional ways of enforcing governance rules using data stewards work against the idea of autonomous teams and do not scale in a distributed setup. Hence with the data mesh approach, the emphasis is to use automation to enforce governance rules. In this article we'll examine how to use the concept of fitness functions to enforce governance rules on data products in a data mesh. [1]

This is particularly important to ensure that the data products meet a minimum governance standard which in turn is crucial for their interoperability and the network effects that data mesh promises.

Data product as an architectural quantum of the mesh

The term "data product" has unfortunately taken on various self-serving meanings, and fully disambiguating them could warrant a separate article. However, this highlights the need for organizations to strive for a common internal definition, and that's where governance plays a crucial role.

For the purposes of this discussion let's agree on the definition of a data product as an architectural quantum of data mesh. Simply put, it's a self-contained, deployable, and valuable way to work with data. The concept applies the proven mindset and methodologies of software product development to the data space.

In modern software development, we decompose software systems into easily composable units, ensuring they are discoverable, maintainable, and have committed service level objectives (SLOs). Similarly, a data product is the smallest valuable unit of analytical data, sourced from data streams, operational systems, or other external sources and also other data products, packaged specifically in a way to deliver meaningful business value. It includes all the necessary machinery to efficiently achieve its stated goal using automation.

What are architectural fitness functions

As described in the book Building Evolutionary Architectures, a fitness function is a test that is used to evaluate how close a given implementation is to its stated design objectives.

By using fitness functions, we're aiming to "shift left" on governance, meaning we identify potential governance issues earlier in the timeline of the software value stream. This empowers teams to address these issues proactively rather than waiting for them to be caught upon inspections.

With fitness functions, we prioritize [2] :

- Governance by rule *over* Governance by inspection.
- Empowering teams to discover problems *over* Independent audits
- Continuous governance *over* Dedicated audit phase

Since data products are the key building blocks of the data mesh architecture, ensuring that they meet certain architectural characteristics is paramount. It’s a common practice to have an organization wide data catalog to index these data products, they typically contain rich metadata about all published data products. Let's see how we can leverage all this metadata to verify architectural characteristics of a data product using fitness functions.

Architectural characteristics of a Data Product

In her book Data Mesh: Delivering Data-Driven Value at Scale, Zhamak lays out a few important architectural characteristics of a data product. Let's design simple assertions that can verify these characteristics. Later, we can automate these assertions to run against each data product in the mesh.

Discoverability	Assert that using a name in a keyword search in the catalog or a data product marketplace surfaces the data product in top-n results.
Addressability	Assert that the data product is accessible via a unique URI.
Self Descriptiveness	Assert that the data product has a proper English description explaining its purpose Assert for existence of meaningful field-level descriptions.
Secure	Assert that access to the data product is blocked for unauthorized users.
Interoperability	Assert for existence of business keys, e.g. `customer_id`, `product_id`. Assert that the data product supplies data via locally agreed and standardized data formats like CSV, Parquet etc. Assert for compliance with metadata registry standards such as “ISO/IEC 11179”

Trustworthiness	Assert for existence of published SLOs and SLIs Asserts that adherence to SLOs is good
Valuable on its own	Assert - based on the data product name, description and domain name - that the data product represents a cohesive information concept in its domain.
Natively Accessible	Assert that the data product supports output ports tailored for key personas, e.g. REST API output port for developers, SQL output port for data analysts.

Patterns

Most of the tests described above (except for the discoverability test) can be run on the metadata of the data product which is stored in the catalog. Let's look at some implementation options.

Running assertions within the catalog

Modern day data catalogs like Collibra and Datahub provide hooks using which we can run custom logic. For eg. Collibra has a feature called workflows and Datahub has a feature called Metadata Tests where one can execute these assertions on the metadata of the data product.

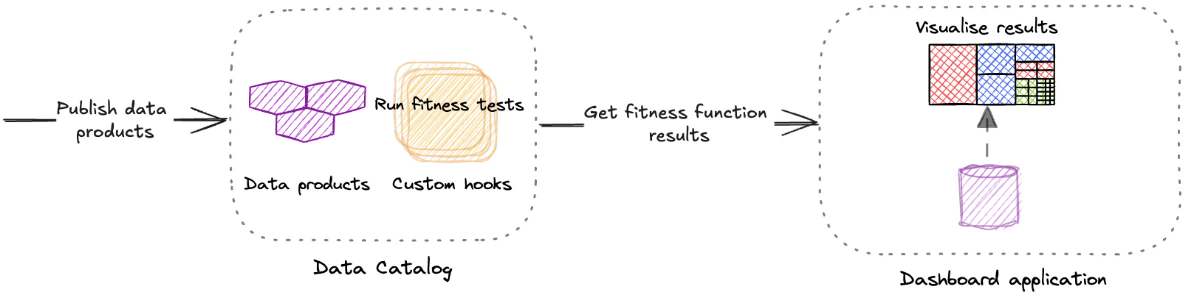


Figure 1: Running assertions using custom hooks

In a recent implementation of data mesh where we used Collibra as the catalog, we implemented a custom business asset called "Data Product" that made it straightforward to fetch all data assets of type “data product” and run assertions on them using workflows.

Running assertions outside the catalog

Not all catalogs provide hooks to run custom logic. Even when they do, it can be severely restrictive. We might not be able to use our favorite testing libraries and frameworks for assertions. In such cases, we can pull the metadata from the catalog using an API and run the assertions outside the catalog in a separate process.

Figure 2: Using catalog APIs to retrieve data product metadata and run assertions in a separate process

Let's consider a basic example. As part of the fitness functions for *Trustworthiness*, we want to ensure that the data product includes published service level objectives (SLOs). To achieve this, we can query the catalog using a REST API. Assuming the response is in JSON format, we can use any JSON path library to verify the existence of the relevant fields for SLOs.

```
import json
from jsonpath_ng import parse


illustrative_get_dataproduct_response = '''{
  "entity": {
    "urn": "urn:li:dataProduct:marketing_customer360",
    "type": "DATA_PRODUCT",
    "aspects": {
      "dataProductProperties": {
        "name": "Marketing Customer 360",
        "description": "Comprehensive view of customer data for marketing.",
        "domain": "urn:li:domain:marketing",
        "owners": [
          {
            "owner": "urn:li:corpuser:jdoe",
            "type": "DATAOWNER"
          }
        ],
        "uri": "https://example.com/dataProduct/marketing_customer360"
      },
      "dataProductSLOs": {
        "slos": [
          {
            "name": "Completeness",
```

```
          "description": "Row count consistency between deployments",
          "target": 0.95
        }
      ]
    }
  }
}
}'''


def test_existence_of_service_level_objectives():
    response = json.loads(illustrative_get_dataproduct_response)
    jsonpath_expr = parse('$.entity.aspects.dataProductSLOs.slos')
    matches = jsonpath_expr.find(response)

    data_product_name = parse('$.entity.aspects.dataProductProperties.name').find(response)[0].value

    assert matches, "Service Level Objectives are missing for data product : " + data_product_name
    assert matches[0].value, "Service Level Objectives are missing for data product : " + data_product_name
```

Using LLMs to interpret metadata

Many of the tests described above involve interpreting data product metadata like field and job descriptions and assessing their fitness, we believe Large Language Models (LLMs) are well-suited for this task.

Let's take one of the trickier fitness tests, the test for *valuable on its own* and explore how to implement it. A similar approach can be used for the *self descriptiveness* fitness test and the *interoperability* fitness test for compliance with metadata registry standards. 3

I will use the *Function calling* feature of OpenAI models to extract structured output from the evaluations. For simplicity, I performed these evaluations using the OpenAI Playground with GPT-4 as our model. The same results can be achieved using their API. Once you have structured output from a large language model (LLM) in JSON format, you can write assertions similar to those described above.

```
System Prompt

You are a data product evaluator. Your job is to look at the meta data about a data
product provided and evaluate if certain architectural properties of the data product
holds true or not.
```

Functions: 4

Tip: Asking LLMs to provide reason for it's evaluation improves accuracy of the evaluations and has a nice side effect that we can pass on this reason to data product owner when a fitness function fails

Functions

```
{
  "name": "get_data_product_fitness",
  "description": "Determine if data product is fit for purpose",
  "strict": false,
  "parameters": {
    "type": "object",
    "required": [],
    "properties": {
      "valuable_on_its_own": {
        "type": "object",
        "properties": {
          "is_fit": {
            "type": "boolean",
            "description": "True if the data product is valuable on it's own, false otherwise"
          },
          "reason": {
            "type": "string",
            "description": "Reason why the data product is valuable on it's own or not"
          }
        },
        "description": "Determines if data product represents a cohesive
                        information concept in its domain. Has value on its own
                        and can be used independent of other data products"
      }
    }
  }
}
```

We can then send the data product metadata to the LLM to evaluate them. Here's a couple of results.

Customer data product: We expect this data product to pass the test for *valuable_on_its_own*

```
User:
{
  "entity": {
    "urn": "urn:li:dataProduct:marketing_customer360",
    "type": "DATA_PRODUCT",
    "aspects": {
      "dataProductProperties": {
        "name": "Marketing Customer 360",
        "description": "Comprehensive view of customer data for marketing.",
        "domain": "urn:li:domain:marketing"
      }
    }
  }
}

Assistant:

 get_data_product_fitness({
  "valuable_on_its_own": {
```

```
      "is_fit": true,
      "reason": "The 'Marketing Customer 360' data product presumably provides a
                 comprehensive view of customer data, which is valuable for marketing
                 purposes. It could offer insights into customer behaviors, preferences,
                 and engagement with marketing campaigns, all of which can be used to
                 improve marketing strategies."
    }
  })
```

Product_Id data product: We expect this data product to fail the test for *valuable_on_its_own*. As product id needs to be always joined with product or customer to have any use.

```
User:
{
  "entity": {
    "urn": "urn:li:dataProduct:product_id",
    "type": "DATA_PRODUCT",
    "aspects": {
      "dataProductProperties": {
        "name": "Product_Id",
        "description": "Table representing product ids per customer",
        "domain": "urn:li:domain:marketing"
      }
    }
  }
}

Assistant:

 get_data_product_fitness({
  "valuable_on_its_own": {
    "is_fit": false,
    "reason": "The 'Product_Id' data product may not be valuable on its own. It simply
               represents product ids per customer and lacks contextual information
               about what those products are. For it to be meaningful, it would
               likely need to be used in conjunction with other data products that
               provide details about the products themselves."
  }
})
```

Publishing the results

Once we have the results of the assertions, we can display them on a dashboard. Tools like Dashing and Dash are well-suited for creating lightweight dashboards. Additionally, some data catalogs offer the capability to build custom dashboards as well.

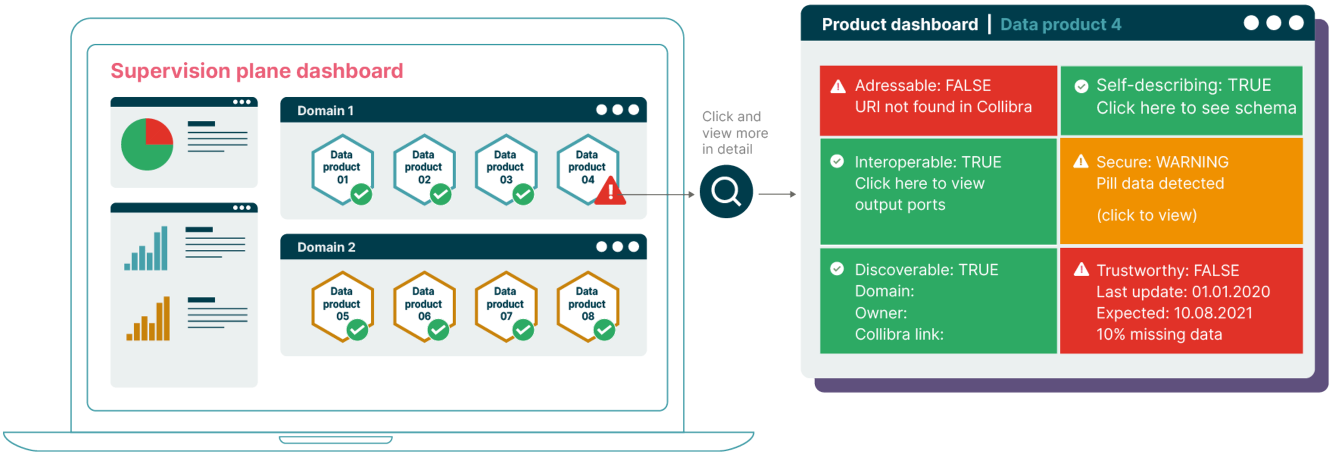


Figure 3: A dashboard with green and red data products, grouped by domain, with the ability to drill down and view the failed fitness tests

Publicly sharing these dashboards within the organization can serve as a powerful incentive for the teams to adhere to the governance standards. After all, no one wants to be the team with the most red marks or unfit data products on the dashboard.

Data product consumers can also use this dashboard to make informed decisions about the data products they want to use. They'd naturally prefer data products that are fit over those that are not.

Necessary but not sufficient

While these fitness functions are typically run centrally within the data platform, it remains the responsibility of the data product teams to ensure their data products pass the fitness tests. It is important to note that the primary goal of the fitness functions is to ensure adherence to the basic governance standards. However, this does not absolve the data product teams from considering the specific requirements of their domain when building and publishing their data product.

For example, merely ensuring that the access is blocked by default is not sufficient to guarantee the security of a data product containing clinical trial data. Such teams may need to implement additional measures, such as differential privacy techniques, to achieve true data security.

Having said that, fitness functions are extremely useful. For instance, in one of our client implementations, we found that over 80% of published data products failed to pass basic fitness tests when evaluated retrospectively.

Conclusion

We have learnt that fitness functions are an effective tool for governance in Data Mesh. Given that the term “Data Product” is still often interpreted according to individual convenience, fitness functions help enforce governance standards mutually agreed upon by the data product teams . This, in turn, helps us to build an ecosystem of data products that are reusable and interoperable.

Having to adhere to the standards set by fitness functions encourages teams to build data products using the established “paved roads” provided by the platform, thereby simplifying the maintenance and evolution of these data products. Publishing results of fitness functions on internal dashboards enhances the perception of data quality and helps build confidence and trust among data product consumers.

We encourage you to adopt the fitness functions for data products described in this article as part of your Data Mesh journey.

Acknowledgements

Thanks Ammara Gafoor, Ian Murdoch, Omar Khawaja and Wolfgang Mai for putting these ideas into action. Thanks Syed Atif Akhtar, Pablo Porto, Chris Ford, Markus Buhmann and Jim Gumbly for proof reading the early drafts. Special thanks to Martin Fowler for gently nudging me to write this.

Footnotes

1: Fitness functions are only one aspect of automated data governance. The wider topic of the varied techniques that are available is outside the scope of the this article. For an outline of the broader topic of governance, see Dehghani.

2: Expressed using the style of Comparative Values

3: At Thoughtworks, we’ve developed an as-yet internal tool called Pendant, currently in use with a few healthcare clients, to assist data product owners in creating glossary definitions. Pendant leverages a LLM to apply ISO 11179 Part 4 requirements and recommendations, evaluating whether a provided definition meets these standards. If a definition falls short, Pendant suggests improvements to ensure compliance

4: In the context of the OpenAI API, *functions* are predefined operations or tasks that a model can reference during a conversation. These functions are included in the prompt and guide the model in generating structured responses, including all necessary parameters for function invocation. For example, we’ve defined a function called `get_data_product_fitness`, which takes in a parameter named `valuable_on_its_own`. This parameter is an object with two properties: `is_fit`, a boolean indicating if the data product is independently valuable, and `reason`, a text field explaining the model's rationale.

▸ Significant Revisions