O'REILLY®

Early Release
RAW & UNEDITED

Compliments of
ACTIAN™
a division of HCLSoftware

Building Data Products

Increase Trust and Unleash Data-Driven Innovation

Jean-Georges Perrin

ACTIAN™
a division of HCLSoftware

Data Products Are How Modern Enterprises Scale AI

Building data products requires more than data.

It requires governance, metadata, quality, ownership, observability, and business context.

Actian helps organizations design, govern, publish, discover, and operate trusted, reusable data products across hybrid, cloud, and on-premises environments.

- Smart Data Products and Data Contracts
- Business Context and Semantic Governance
- Federated Knowledge Graph
- End-to-End Lineage
- Data Quality and Observability
- Unified Catalog and Marketplace
- Open Standards Support (ODPS & ODCS)

Build trusted data products that humans and AI can use with confidence.

Explore Actian Data Intelligence at actian.com

Building Data Products

Increase Trust and Unleash Data-Driven Innovation

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Jean-Georges Perrin

O'REILLY®

Building Data Products
by Jean-Georges Perrin

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Jill Leonard
Production Editor: Ashley Stussy

Interior Designer: David Futato
Interior Illustrator: Kate Dullea

November 2026: First Edition

Revision History for the Early Release

2025-09-23: First Release
2026-03-05: Second Release
2026-07-01: Third Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341629400* for release details.

979-8-341-62935-6

[FILL IN]

Table of Contents

Brief Table of Contents (*Not Yet Final*)

CHAPTER 1

Beyond Data Contracts

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *ccollins@oreilly.com*.

Data contracts, artifacts, ports, SBOMs: if your head is already spinning, don't worry. This chapter is about setting the stage, not drowning you in acronyms and concepts. Before we can build, automate, or measure, we need to agree on the basics: what a data product is, why it matters, and how the vocabulary around it gives us common ground. Without this foundation, we're just yelling "data!" at each other in different dialects.

Think of a data contract like the menu at your favorite restaurant: it sets clear expectations between chef and customer. Without it, your data product is just a mystery plate you'll get from your waiter.

Data Products

Data products are revolutionizing data management—ushering in a new era of reusable, standardized, and value-driven assets. Whether you're a data engineer, architect,

or product owner, ignoring this shift is like parking on the shoulder of the road while the rest of the industry speeds down the data highway.

Let's start with a definition.

> A data product is a reusable, active, and standardized data asset designed to deliver measurable value to its users, whether internal or external, by applying the rigorous principles of product thinking and management. It comprises one or more data artifacts (e.g., datasets, models, pipelines) and is enriched with metadata, including governance policies, data quality rules, data contracts, and, where applicable, a Software Bill of Materials (SBOM) to document its dependencies and components. Ownership of a data product is aligned to a specific domain or use case, ensuring accountability, stewardship, and its continuous evolution throughout its lifecycle. Adhering to the FAIR principles — Findable, Accessible, Interoperable, and Reusable — a data product is designed to be discoverable, scalable, reusable, and aligned with both business and regulatory standards, driving innovation and efficiency in modern data ecosystems.

This definition was finalized after an epic session at the Data Day Texas conference, in Austin, TX, at the end of January 2025. It was not the first definition ever crafted of a data product, but it was time for a refresh that would be a little more practical (and a lot less abstract). After all, we are engineers, and we like concrete stuff.

Let's break this definition down further by exploring some of the key terms within it:

Data contract
: A formal, versioned agreement between a data producer and consumers that defines the schema or structure, quality, and expectations (such as SLA) of data exchange.

Artifact or Output Port
: A concrete output of a data product—such as a dataset, a model, or a file—that delivers value and can be versioned independently.

Port
: An interface through which a data product exchanges information, including inputs, outputs, and interactions for control, discovery, and observability.

SBOM (Software Bill of Materials)
: A detailed, structured list of components, dependencies, and transformations that make up a data product, providing transparency and traceability for how data is generated, think of a manifest with a `pom.xml` file in Java/Maven project.

When we consider all of these terms together, I would set the postulate that a data product is a logical construct[1]: it supersedes the implementation.

1 You will see in the later chapters that a data product may not be only a logical construct, but let's not burn steps!

Reusable

Since I have been in IT (and one could argue I started in 1983 when I was 12, coding video games and trying to write structured Atari BASIC), I have been confronted with reusability, as in "do the work once, reuse multiple times." Although this is desirable, for a long time, it was simply wishful thinking. However, data products can achieve this goal in many ways, let's see how.

Data products can be reused by many consumers, as illustrated by Figure 1-1. I hear you, questioning my sanity: "how is that different from a database?"

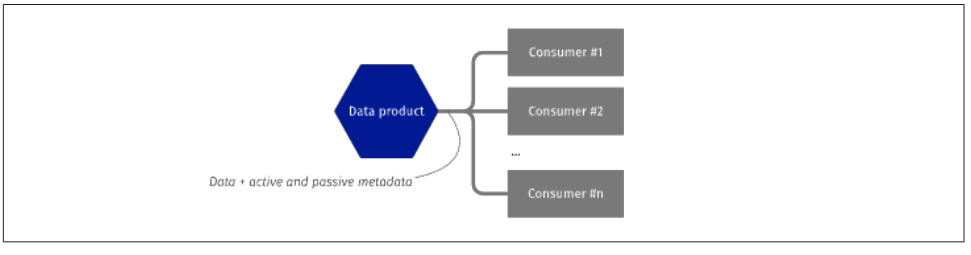

Figure 1-1. A data product and its consumers.

Usually, a consumer of a database will limit its consumption to the data from the database. Sometimes, on rare occasions, it will consume some passive metadata, such as information from the schema. However, with data products, the consumers are encouraged to consume metadata, both passive and active. This may include the results of data quality and a lot more.

Consumers can take many approaches: applications, reports, feeding systems to a large-language model (LLM), or even other data products. When data products are linked, they can exchange information with sources they trust via data contracts, as illustrated by Figure 1-2.

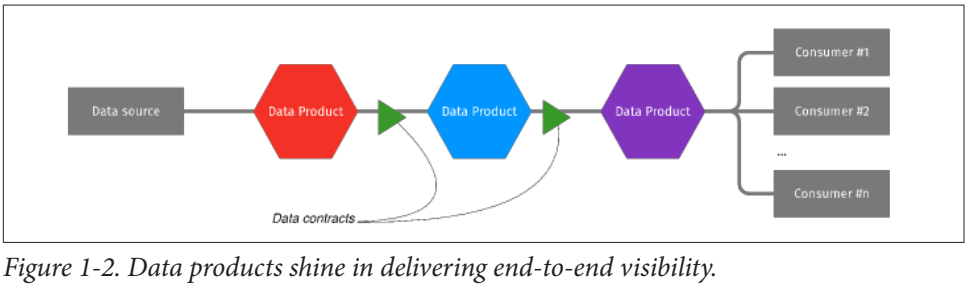

Figure 1-2. Data products shine in delivering end-to-end visibility.

Active

A data product is active, but what does it mean to be active in the data world? To answer this, let's compare a data product to a dataset. A dataset is a passive entity:

although they are useful, they are not self-aware. Let's look at Table 1-1 for a detailed comparison.

Table 1-1. Comparison of a dataset and a data product.

	Dataset	Data Product
Aligned to	A specific use case	A domain, several domains, or a specific use case
Awareness	Itself, although a small set, its passive metadata can be extrapolated	Its lifecycle and the data journey Its access control Its metadata Its quality and SLA (Data QoS) Its users (subscribers) Its maturity level
Access	Specific to the dataset, occasionally documented	Specified in its descriptor
Monitoring	External	Integrated
Content	Tabular data	Datasets, models, federated data, and more

A data product is active because of the ports that it has. Figure 1-3 details the C4 architecture, level 2. Of course, there are the obvious ports, such as the input of data and the access to data.

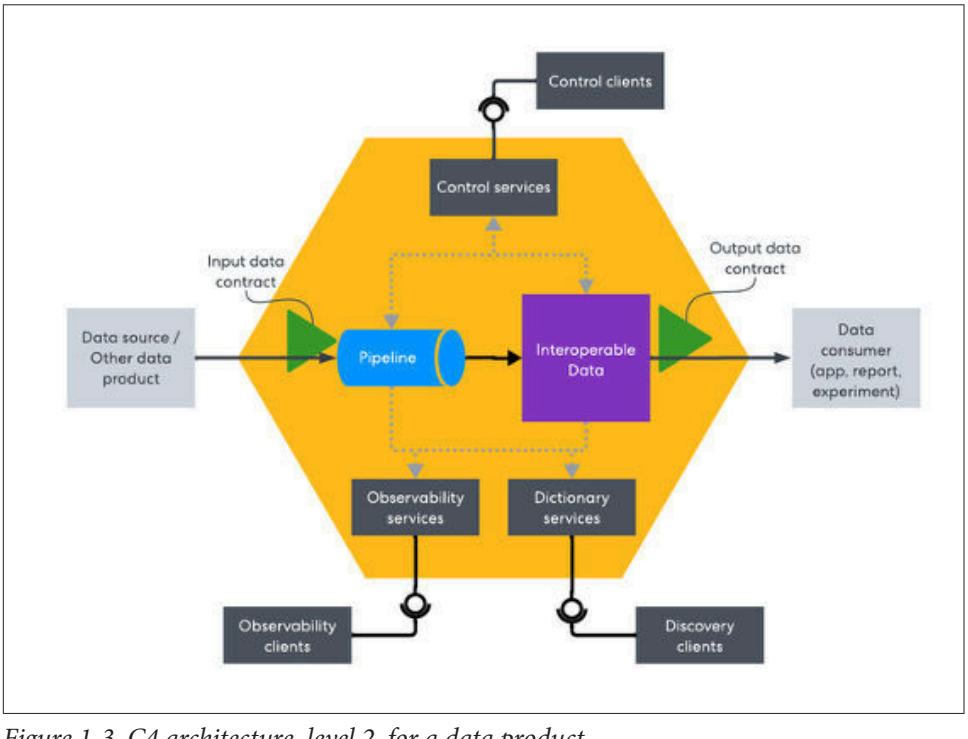

Figure 1-3. C4 architecture, level 2, for a data product.

There are also three sets of endpoints designed for interacting with the data product.

Control
: Manages lifecycle actions like configuration, access control, and deprecation. Examples: An API to update a data product, control the pipeline, deploy a new version, and more.

Observability
: Captures operational metadata, including data quality, SLA, refresh status, and pipeline health. Examples: Metrics from OpenTelemetry, Datadog, or Bitol's upcoming Open Observability Result Standard (OORS).

Discovery
: Enables search and exploration of the data product's structure, metadata, and access policies. Examples: Search for available artifacts, datasets, and schemas; retrieve metadata, including ownership, tags, and domain information.

Those endpoints could be implemented as REST endpoints or subscribers/publishers on dedicated topics/messaging queues. Chapter 4, Building Your First Data Product, will focus on architecture.

Standardized

Standardizing data products ensures consistency and interoperability across a given data ecosystem. A common structure for metadata, access controls, and lifecycle management reduces complexity for both producers and consumers. It also enables seamless cross-team collaboration and integration with various tools.

From a software vendor perspective, adhering to standards will expand the sales of your products while comforting your customers with the idea that you will not try to lock them in with proprietary tooling.

Key benefits of standardization include easier automation and managed self-service. With defined schemas and governance rules, validation, access provisioning, and metadata updates can be automated. This enables faster data discovery, reduced engineering effort, and greater operational efficiency.

Governance and compliance become easier when data products follow a standard. Security policies, access controls, and lineage tracking are uniformly applied, ensuring regulatory compliance and auditability. Additionally, a marketplace allows teams to discover, assess, and reuse data products effortlessly, eliminating duplication of work.

Bitol's ODPS (Open Data Product Standard) and ODCS (Open Data Contract Standard) are the best choices for standardization, providing an open, vendor-neutral approach. These standards enable reuse, automation, and ecosystem growth, ensuring data products are future-proof, scalable, and aligned with industry best practices.

You will learn more about those standards in Chapter 2, Adopting Standards for Data Products.

Measurable Value

Measuring the value of a data product ensures it delivers business impact, drives adoption, and justifies investment. Key metrics such as user adoption, feedback, usage frequency, data quality, and SLA compliance help optimize performance, align with business goals, and build trust among consumers . It also aids in resource allocation, compliance, risk management, and innovation, keeping data products relevant and cost-effective. Without measurement, data products risk becoming stale, wasteful, redundant, or misaligned, making continuous tracking—through automated processes—essential for maximizing their strategic and operational value.

A structured approach to measuring value combines quantitative metrics, qualitative insights, and business alignment. A scorecard method can assign weight to each metric based on its organizational importance, providing a clear evaluation framework. Additionally, dashboards integrated into the data product can offer a real-time view of these metrics, helping Data Product Owners (DPOs), monitor performance, prioritize improvements, and demonstrate value to stakeholders.

Content

A data product descriptor describes the data product content. Let's dive into its structure. It will help us understand how ODPS is designed. ODPS follows the same structure and guiding principles that ODCS.

A data product has specific properties, like:

- Fundamentals, including a name, a semantic version, kind (ODPS' structure is based on Kubernetes' resource descriptors), and more.
- Metadata endpoints: As you read earlier, a data product is an active component, as such it has endpoints to communicate with. Those endpoints can be REST endpoints or topic pub/sub.
- Service-level agreements (SLAs): Sets the default service level expectations for the data product. They can be fine-tuned at the output port levels.

Each data product can have several (data) artifacts. An artifact is defined by a few fields like an identifier, a name, or a type. Each artifact can have several versions.

Each artifact version has a set of properties:

- Details of a version: like the semantic version itself.
- List of input data contracts: leveraging ODCS' identifier and version.

- List of output data contracts: in most cases, there is only one output data contract.
- The SBOM defines how the data is being transformed for this output port (this really describes how the sausage is being done).

Each of those three levels – data product, artifact, and version – can also have the following properties:

- A team, containing both individuals and their role.
- A support channel, like email, slack, etc. like the support block in the contract.
- A list of roles and the level of access they should be provided
- Tags and custom properties to ensure extensibility.

Figure 1-4 illustrates the different sections of a data product and their attributes.

Figure 1-4. Content of a data product.

Example 1-1 illustrates what a basic data product could look like using ODPS. ODPS is described extensively in Chapter 2.

Example 1-1. Fundamentals of a data product, in YAML.

```
name: Loyalty Member Profile
version: 0.1.0
id: f7e9d245-28db-4f1d-bedf-a3fe9f458427
kind: DataProduct
apiVersion: 0.9.0
description: "Offers a unified view of each loyalty members journey"
domain: Loyalty
status: active

outputPorts:
```

```
- artifactId: a1b2c3d4-5678-90ab-cdef-123456789abc
  name: loyalty_member_profile_ds
  type: dataset
  version: 1.0.0
  contractId: 04cac846-4d1d-4634-a505-b01696acd6a3
  sbom:
  - url: "https://mysbomserver/mysbom"
  inputContracts:
  - id: a1b2c3d4-5678-90ab-cdef-123456789012
    version: 1.0.0
```

Product Thinking and Management

Now that we understand what data products are, let's dive into how product thinking and product management concepts reshape the way we build and manage data products.

The Theory

Let's start by a quick intro to the theory of product thinking and management.

Product thinking is a mindset shift, where the focus moves from simply delivering outputs (like datasets or features) in a project mode, to delivering meaningful and user-centric outcomes. It involves deeply understanding the problems, needs, and goals of the end user, and continuously iterating to create solutions that provide real value.

Product management is the practice of guiding a product—from concept to delivery and beyond—to ensure it delivers maximum value to users and the business. A product manager (PM) acts as the bridge between users, business, and engineering, defining the product vision, prioritizing features, and coordinating efforts across teams. In data, product management means treating data assets like products: with roadmaps, feedback loops, key performance indicators (KPIs), and continuous improvement. The goal is to solve real problems, drive adoption, and ensure the product evolves with changing needs—all while aligning with strategic objectives.

Focus on the customer is the most important part: methods like determining the ideal customer profile (ICP, not to be confused with Insane Clown Posse) can help (Scearce, 2019).

It's all about building the right thing, at the right time, in the right way with a real lifecycle from ideation to retirement.

Real Life Application

What does it look like when the rubber hits the road? Let's explore this by considering the idea of product management within a single data product.

In this scenario, Beth, our data product owner, received a significant number of requests to justify an evolution from her version 1.0.0 of her dataset to a version 2.0.0. In traditional data engineering, this would mean starting a new pipeline, provisioning a new storage, and more. When the work around version 2.0.0 is done, there would be a cutoff date, with potentially a scream test before decommissioning the old version.

When it comes to data products, you can imagine that the same data product offers access to both datasets at the same time. Figure 1-5 illustrates this situation. This scenario allows a smoother transition and avoids hard cutoffs, which often results in production issues.

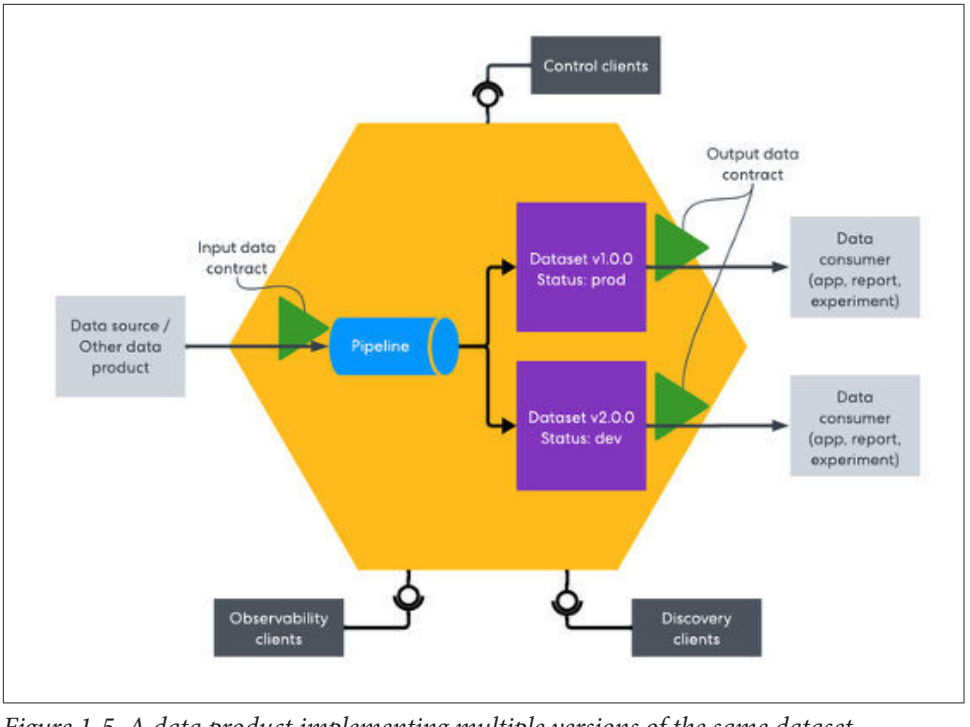

Figure 1-5. A data product implementing multiple versions of the same dataset.

Nothing forbids you from having several versions in production, as shown in Figure 1-6.

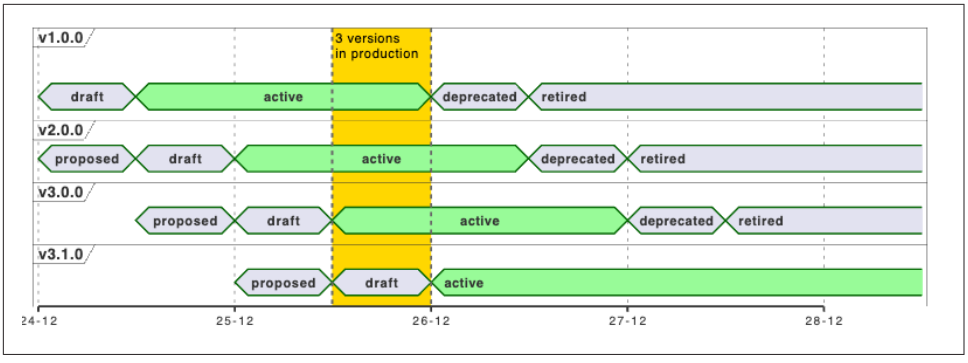

Figure 1-6. Lifecycle of output ports with different versions within a data product.

If I extend the data product described in Figure 1-3 by adding a second dataset, it would look like Figure 1-7: two data artifacts, which will result in two pipelines, two datasets, and two output data contracts. In some literature, you will also see output ports instead of data artifacts. I like output port less as it does not represent the entire chain, from pipeline to the effective port.

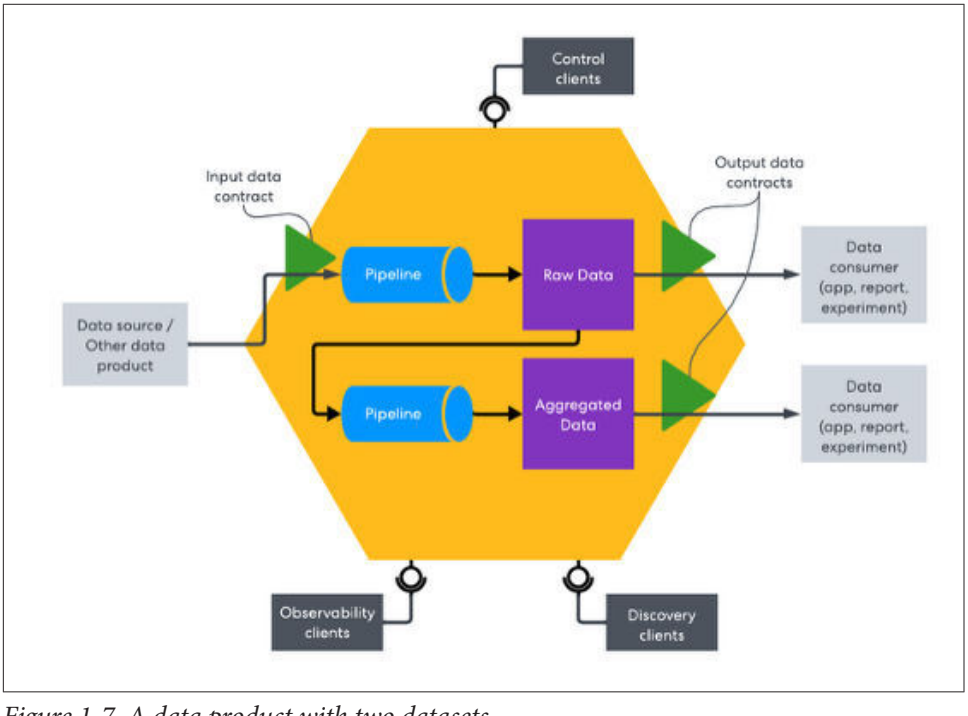

Figure 1-7. A data product with two datasets.

A data product does not have to focus on a single dataset (or single artifact), it can group artifacts in domains or by use cases.

Specific Domain or Use Case

I'm often asked, *Should I align to a specific domain or just focus on my use-case?* Let's consider a few scenarios.

If you are a data producer, you will most likely oversee building data products. Your data product will be the visible part of your production system, as shown in Figure 1-8. Domain aligned data product is a common term, but you will also find producer-aligned or source-aligned data products in some documentation

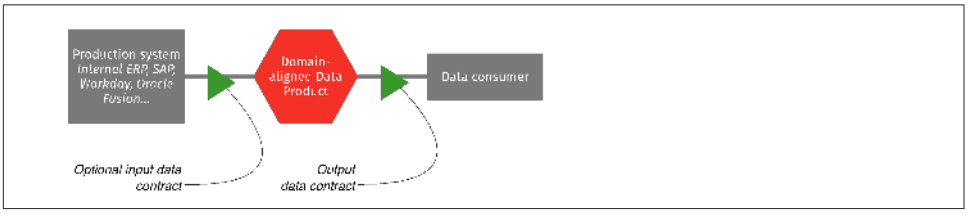

Figure 1-8. Domain-aligned data product offering access to production data.

Now, if you are combining two systems, let's say an air reservation system and a hotel system to build a trip, are you aligning to a domain or a use case (or a set of use cases)? In this scenario, your trip data product is a cross-domain data product and its strength is based on the strength of the upstream system, hence the importance of the data contracts. Figure 1-9 illustrates this combination. Cross-domain data product is a popular term, but you will also find consumer-aligned data products in some documentation.

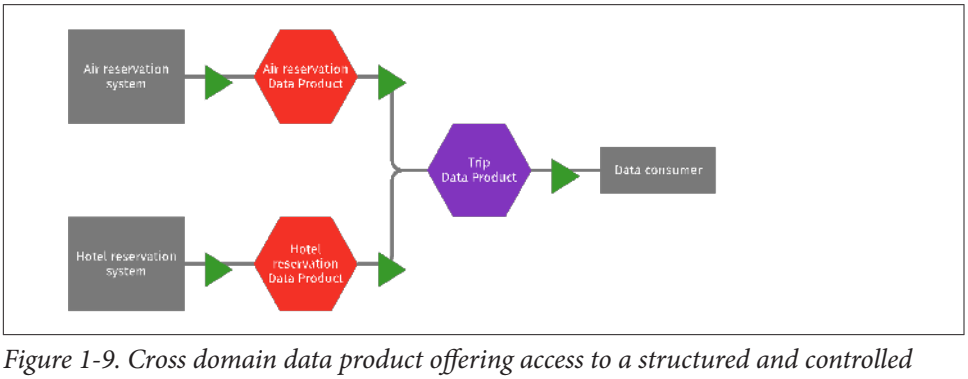

Figure 1-9. Cross domain data product offering access to a structured and controlled data product.

Zhamak Dehghani aligns data products to domains, which can create some confusion when fit-for-purpose data products do not align on an enterprise domain (Dehghani, 2022). Therefore, many practitioners align domain data product to the source data and combination of data products which are not all in the same domain to a cross-domain category.

Findability, Accessibility, Interoperability, and Reusability (FAIR)

The FAIR principles (Collective 2016) are a set of guidelines designed to improve the findability, accessibility, interoperability, and reusability of data. Introduced to promote responsible data management and sharing, they help ensure that data can be easily discovered and used—not just by humans, but also by machines. Let's break this acronym down:

Findable
: Data should be easy to locate using rich metadata and persistent identifiers.

Accessible
: Data should be retrievable using standard protocols, with clear authentication and authorization where needed.

Interoperable
: Data should use standardized formats and vocabularies to integrate with other datasets and tools.

Reusable
: Data should include detailed provenance and licensing to support long-term use and reproducibility.

FAIR is not a standard, but a framework for thinking about how to make data more useful and valuable across domains. It is normal that data products build on top of this framework.

Summary

Data products aren't a buzzword—they're a foundational shift in how we design, consume, and manage data. By applying product thinking, standardization, and a relentless focus on usability, we create assets that aren't just technically sound but also deliver tangible and measurable business value.

Think like a product manager
: Understand your users and evolve with their needs.

Be active, not passive
: Your data product should self-report, be observable, and interact via data and metadata ports, not just sit there like a dusty dataset.

Reuse is real now
: Design once, use many times, and track its impact.

Standardize to scale
: ODPS and ODCS offer frameworks for governance, exchange, automation, and ecosystem interoperability.

Measure value continuously
: Adoption, data quality, SLA compliance, and user satisfaction are the KPIs that matter.

You now have the framework: the future belongs to teams that build data like they build software—with ownership, accountability, and care.

Further Reading

FAIR Principles, from GO FAIR: *https://www.go-fair.org/fair-principles/*.

Data Mesh, by Dehghani, Z. (2022). O'Reilly.

Defining Data Products: A Community Effort., by Perrin, J.-G. et all (2025, January 28): *https://medium.com/data-mesh-learning/defining-data-products-a-community-effort-77363611e5c5*.

The Framework for Ideal Customer Profile Development, by Scearce, T. (2019, July 10): *https://www.gartner.com/en/articles/the-framework-for-ideal-customer-profile-development*.

CHAPTER 2

Adopting Standards for Data Products

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *ccollins@oreilly.com*.

Imagine a world where every website required a different browser, and each browser used its own proprietary language to display content. That world almost happened. In the early days of the internet, companies like Netscape (with Netscape tags) and Microsoft (with VBScript and ActiveX) pushed their own formats. If they had won, the web would be a fragmented mess.

Thankfully, HTML, an open standard, prevailed. It became the lingua franca of the web, allowing anyone to build, anyone to consume, and everyone to innovate.

The same tipping point is happening with data.

Without shared standards, each team becomes a silo, building incompatible formats, reinventing wheels, and shipping fragile integrations. But with the rise of open, community-driven standards, like the Open Data Contract Standard (ODCS) and the Open Data Product Standard (ODPS) from the Bitol project at the Linux Foundation, we finally have the HTML moment for data.

These standards don't just define how data products should look. They define how they work. How they can be governed, discovered, trusted, and reused.

Want your data ecosystem to scale like the web? Start by speaking a common language.

A Brief History of Bitol

Bitol wasn't exactly planned. It started as a practical fix during the early days of implementing data mesh at PayPal. We needed a generic way to describe data resources. Enter the data contract template. In May 2023, PayPal released it as open source under the Apache 2.0 license.

Shortly after, I left the company.The appetite to maintain an open standard faded. So I took the idea to the Linux Foundation, who saw the opportunity to bring open, lightweight standards to the fast-evolving world of data engineering. Bitol was born.

The Foundation had already backed solid projects like Egeria and OpenLineage, each with a strong focus: metadata management and data lineage, respectively. Bitol came in as a complement, not a competitor, offering a set of minimal, interoperable, and implementation-agnostic standards for data contracts and data products.

Governance was key. Over several months, I gathered a global, cross-industry group, 15 people from 13 companies, to form Bitol's first Technical Steering Committee, blending voices from software vendors, service providers, and enterprise users. This group, with additional volunteers, became a powerful engine to standardize data contracts through the Open Data Contract Standard (ODCS) and, a little later, data products with Open Data Product Standard (ODPS).

More standards are on the roadmap. But the mission remains simple: codify the building blocks of modern data engineering, without dictating the architecture.

Bitol ODPS and ODCS

Now that I've covered the why, let's dive into the how. In this section, I'll open up the hood on the Open Data Product Standard (ODPS) and the Open Data Contract Standard (ODCS). They're concrete, structured, and ready-to-code principles.

You'll see how each standard is designed to be:

- Readable by humans
- Consumable by machines
- Friendly to governance

We'll walk through the anatomy of both ODCS and ODPS, field by field, with a full example. By the end of this section, you'll not only understand how these standards work—you'll be ready to implement them, extend them, and maybe even contribute back.

Diagram of a Data Product

Figure 2-1 should remind you s us what a customer data product could look like.

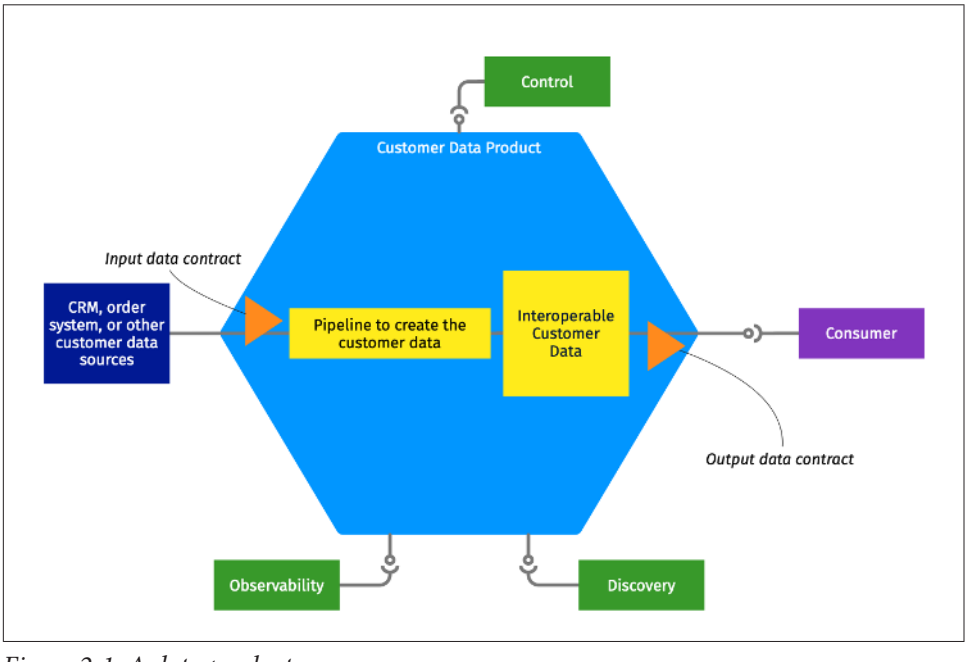

Figure 2-1. A data product

The data product exposes the data contract; it is its guarantee or promise. For this first example, I will turn a simple customer dataset into a data product. My dataset uses three tables: `Customer`, `Address`, and `AddressType`, as illustrated by Figure 2-2.

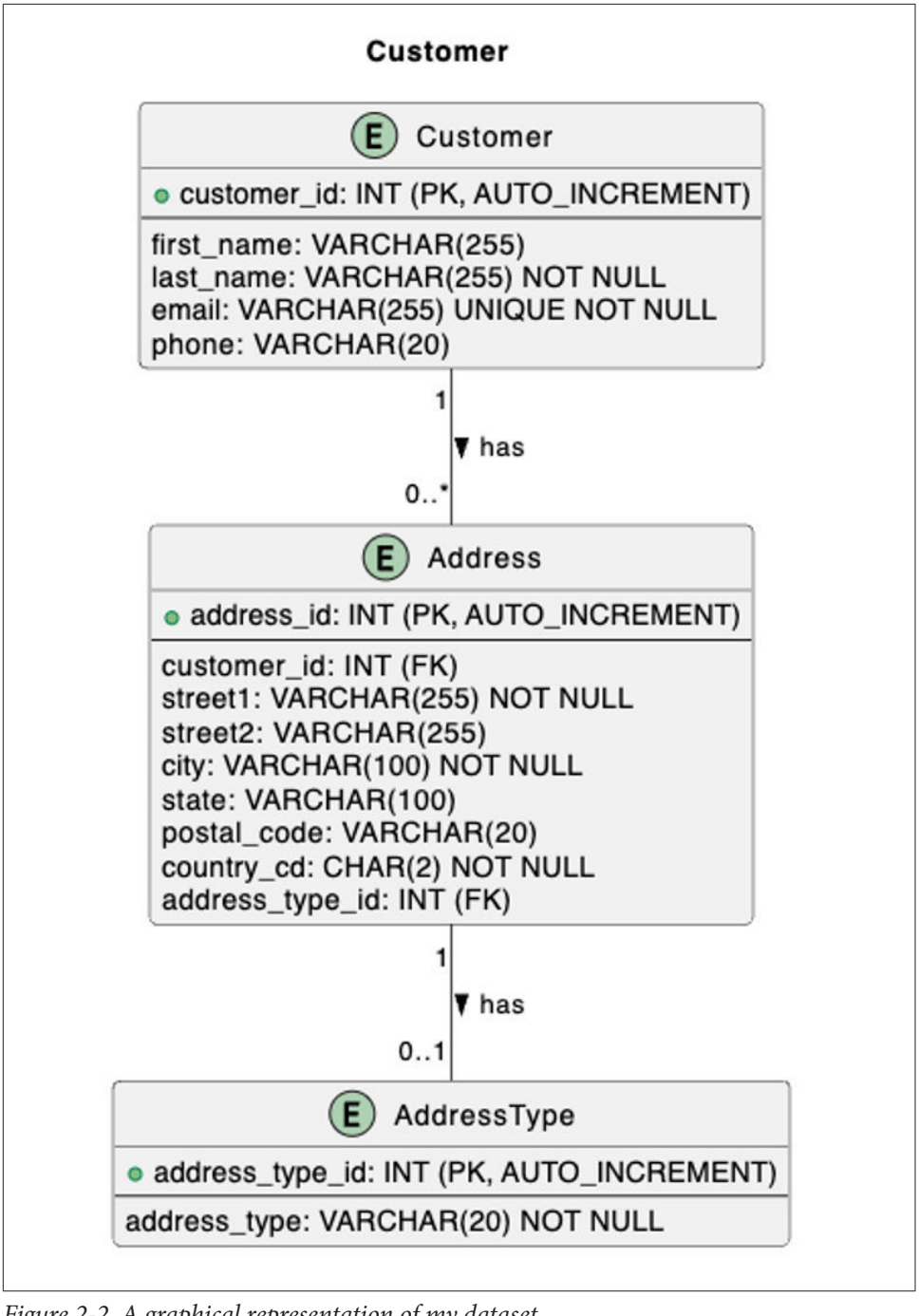

Figure 2-2. A graphical representation of my dataset

You can also represent these tables as a SQL script, more specifically Data Definition Language (DDL), as shown in Example 2-1. This file can be found here.

Example 2-1. The SQL script for creating our small customer database (or DDL)

```
-- Table: Customer
CREATE TABLE Customer (
    customer_id SERIAL PRIMARY KEY,
    first_name VARCHAR(255),
    last_name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    phone VARCHAR(20)
);

-- Table: AddressType
CREATE TABLE AddressType (
    address_type_id SERIAL PRIMARY KEY,
    address_type VARCHAR(20) NOT NULL
);

-- Table: Address
CREATE TABLE Address (
    address_id SERIAL PRIMARY KEY,
    customer_id INT NOT NULL,
    street1 VARCHAR(255) NOT NULL,
    street2 VARCHAR(255),
    city VARCHAR(100) NOT NULL,
    state VARCHAR(100),
    postal_code VARCHAR(20),
    country_cd CHAR(2) NOT NULL,
    address_type_id INT,

    CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES Customer(cus
tomer_id) ON DELETE CASCADE,
    CONSTRAINT fk_address_type FOREIGN KEY (address_type_id) REFERENCES Addres
sType(address_type_id)
);
```

The schema from Example 2-1, represents the data part in our product. It is the highlighted part of our data product, as shown in Figure 2-3.

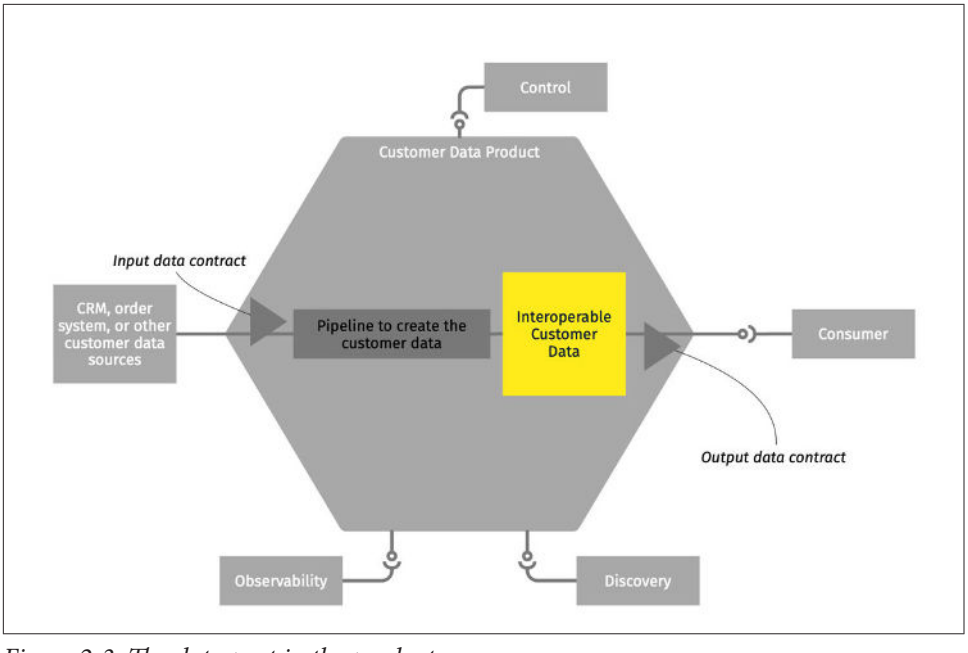

Figure 2-3. The data part in the product

Start with the Promise: The Data Contract

You have the schema. Now make a promise about it.

A data contract describes the data before anyone consumes it — what it contains, what quality to expect, who owns it. Misunderstandings should happen in YAML, not in production at 2 a.m. (note that it can still happen, but it will provide so much information that you will be back in bed at 2:05 a.m.).

Example 2-2 shows the most minimal valid contract for our customer schema. The contract is deliberately sparse (I will enrich it shortly), but even this skeleton names the objects, documents the structure, and gives the contract an identity and a lifecycle.

Example 2-2. A minimalist data contract defining three tables

```
kind: "DataContract" [A]
apiVersion: "v3.1.0" [A]
version: "v0.1.0" [B]
status: "proposed" [C]
id: "34cae6d7-7648-38b2-8f66-8db79e1e2ce4" [D]

schema: [E]
- name: "Customer" [F]
```

```
  properties: [F]
  - name: "customer_id" [G]
  - name: "first_name" [G]
  - name: "last_name" [G]
  - name: "email" [G]
  - name: "phone" [G]
- name: "AddressType"
  properties:
  - name: "address_type_id"
  - name: "address_type"
- name: "Address"
  properties:
  - name: "address_id"
  - name: "customer_id"
  - name: "street1"
  - name: "street2"
  - name: "city"
  - name: "state"
  - name: "postal_code"
  - name: "country_cd"
  - name: "address_type_id"
```

From this example, what can you learn?

- [A] kind and apiVersion: You can see the type (`kind`) of artifact and the version of the standard it matches. Each Bitol standard is based on a similar YAML structure. `apiVersion` refers to the version of the standard. This minimalist example uses ODCS v3.1.0.
- [B] `version`: Indicates the version of the data contract, using semantic versioning; here it is v0.1.0.
- [C] `status`: Can be any value, but recommended values are `proposed`, `draft`, `active`, `deprecated`, and `retired`.
- [D] `id`: The identifier can be any value, but I highly recommend using UUIDs as they should be unique.
- [E] The `schema` is an essential part of the data contract. Since ODCS v3, it can contain a hierarchy with objects and properties.
- [F] The first element, `customer`, is an object as indicated by the properties key. If you are thinking in a relational world, customer would be a table.
- [G] All the properties of the customer object. In a relational world, they would be columns.

Although this minimalist data contract provides some value, it *is* minimal. Let's enrich it and unfold more value. For clarity, I did not dump the entire enriched contract in Example 2-3. The full contract is available on GitHub.

In Example 2-3:

- I increased the `version` to v0.2.0 ([A]) and changed the `status` to `active` ([B]),
- The `id` ([C]) is the same: Example 2-3 is an evolution of Example 2-2, not a different contract.
- Documentation is enhanced in different areas ([D]).
- The primary key is identified ([E]).
- Required fields are mentioned ([F]). Note that the fact that they are mentioned does not affect data quality ([G]).
- I use two different rules ([H]) with parameters. The first rule [I] is based on the `nullValues` that are being counted. There can be a tolerance expressed in percent or rows. The second rule [J] focuses on invalid values and has a very low threshold for tolerance.
- Another block of information is the team [K], where you can learn that I have been the Data Product Owner (DPO) for this contract since July 7th, 2025.
- Finally, this contract has only one SLO (service-level objective) [L]: the retention period is three years. However, as you can see, there is no way to enforce this value as there is no date field to measure on. Enforcement requires a field, such as a creation or ingestion timestamp, that an external process can evaluate at runtime. Without that anchor, the SLA is a promise written in YAML, not a guarantee.

Example 2-3. A more complete data contract on the same tables

```yaml
apiVersion: "v3.1.0"
kind: "DataContract"
version: "v0.2.0" [A]
status: "active" [B]
id: "34cae6d7-7648-38b2-8f66-8db79e1e2ce4" [C]
description:
  usage: "Customer information including address for analytics" [D]
name: "CustomerContract"
tenant: "ClimateQuantum"

schema:
- logicalType: "object"
  name: "Customer"
  physicalName: "Customer"
  physicalType: "table"
  description: "Defines a customer according to the standard enterprise model." [D]
  properties:
  - logicalType: "number"
    name: "customer_id"
```

```
    physicalName: "customer_id"
    physicalType: "SERIAL"
    primaryKey: true [E]
    required: true [F]
    description: "Customer id is always required." [D]
  - logicalType: "string"
    name: "first_name"
    physicalName: "first_name"
    physicalType: "VARCHAR (255)"
  - logicalType: "string"
    name: "last_name"
    physicalName: "last_name"
    physicalType: "VARCHAR (255)"
    required: true [F]
    quality: [G]
    - metric: nullValues [H] [I]
      mustBeLessThan: 1
      unit: percent
      description: "There must be less than 1% of null values in the last name col
umn."
  - logicalType: "string"
    name: "email"
    physicalName: "email"
    physicalType: "VARCHAR (255)"
    required: true [F]
    quality: [G]
    - metric: nullValues [H] [I]
      mustBeLessThan: 5
      unit: percent
      description: " There must be less than 5% of null values in the e-mail col
umn. Historically e-mails were not collected."
    - metric: invalidValues [H] [J]
      mustBe: 0
      description: "The value must be a valid email, there is no tolerance."
      arguments:
      pattern: |
        "/^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]
{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/"
  - logicalType: "string"
    name: "phone"
    physicalName: "phone"
    physicalType: "VARCHAR (20)"
- name: "AddressType"
  properties:
    - name: "address_type_id"
    - name: "address_type"
- name: "Address"
  properties:
    - name: "address_id"
    - name: "customer_id"
    - name: "street1"
    - name: "street2"
```

```
    - name: "city"
    - name: "state"
    - name: "postal_code"
    - name: "country_cd"
    - name: "address_type_id"

team: [K]
  name: "Customer Team"
  description: "This team is in charge of maintening the customer model" [D]
  members:
  - username: "jgp"
    dateIn: "2025-07-07"
    name: "Jean-Georges Perrin"
    role: "DPO"

slaProperties: [L]
  - property: retention
    value: 3
    unit: y
```

As you can see, version 0.2.0 of the contract provides a lot more information (and value). You can add more:

- Data quality rules
- SLA expectations.
- Documentation through description and other fields.
- Add security.
- Add authoritative definitions to ease governance.

As you write your first contracts, you should keep in mind the following:

- It's fairly easy: the minimalist data contract provides value with minimal effort (here's another, even easier example you can share).
- A data contract is never finished: you keep enriching it.
- The more information you put in the contract, the more value you will get out of it.

You have your output data contact, as shown in Figure 2-4.

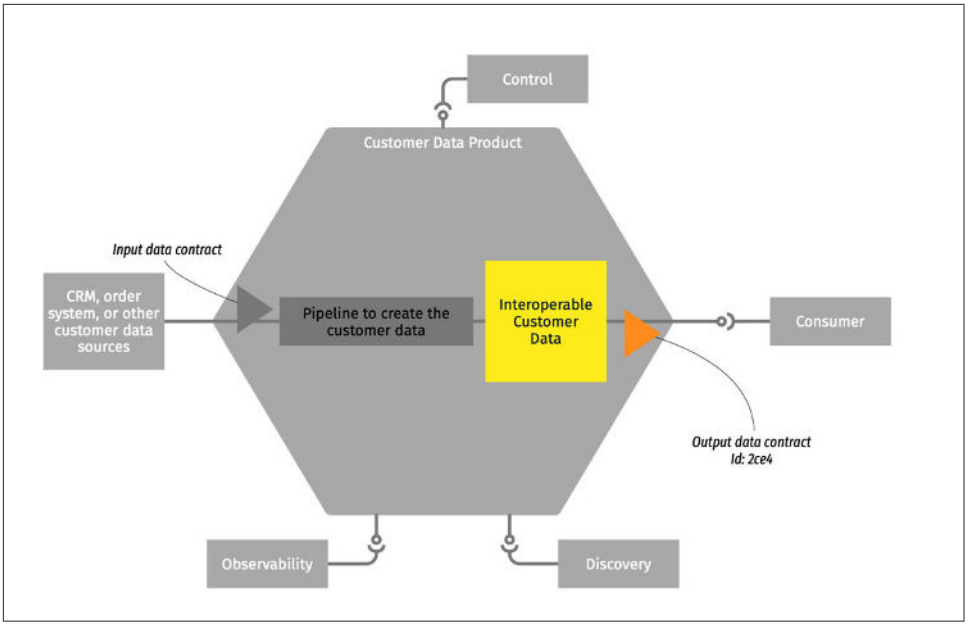

Figure 2-4. The output data contract

In the next section we'll build a data product.

Create a Data Product

Now that we have the output data contract, let's wrap it in a data product following the ODPS standard from Bitol.

In a similar approach, Example 2-4 builds a simple version, and then you will add more value to it.

Example 2-4. A proposed version of a data product

```
apiVersion: v1.0.0 [A]
kind: DataProduct [A]

version: "v0.1.0" [B]
status: "proposed" [C]
id: 2abf58db-8672-43a2-88fa-787e6c37be46 [D]
name: Customer Data Product [E]

description:
  purpose: Enterprise view of a customer. [F]

outputPorts: [G]
- name: customerTable [H]
```

```
  description: "Customer Table" [I]
  contractId: 34cae6d7-7648-38b2-8f66-8db79e1e2ce4 [J]
```

Let's dive into that code:

- [A] `kind` & `apiVersion`: Type (`kind`) of artifact and the version of the standard it matches. Each Bitol standard is based on a similar YAML structure. `apiVersion` refers to the version of the standard. This first data product uses ODPS v1.0.0.
- [B] `version`: Indicates the version of the data product, using semantic versioning; here it is `v0.1.0`.
- [C] `status`: Can be any value, but recommended values are `proposed`, `draft`, `active`, `deprecated`, and `retired`.
- [D] `id`: The identifier can be any value; however, like for data contracts, I highly recommend using UUIDs as they should be unique.
- A data product can have a `name` [E].
- Description is encouraged ([F]). It follows the same structure as ODCS: `purpose`, `limitations`, and `usage`.
- A data product must have at least one output port [G].
- In the output port, you have a `name` ([H]), a `description` ([I]), and a contract id ([J]).

The link between the data product and the data contract is based on the contract id. Let's enrich this version, as shown in Example 2-5.

Example 2-5. A proposed version of a data product

```
apiVersion: v1.0.0
kind: DataProduct

version: "v0.2.0" [A]
status: "active" [B]
name: Customer Data Product
id: 2abf58db-8672-43a2-88fa-787e6c37be46

description:
  purpose: Enterprise view of a customer.

inputPorts: [C]
  - name: customerCrm
    description: "Customer as defined in the CRM"
    contractId: dbb7b1eb-7628-436e-8914-2a00638ba6db

outputPorts:
  - name: customerTable
```

```
    description: "Customer Table"
    contractId: 34cae6d7-7648-38b2-8f66-8db79e1e2ce4
    sbom: [D]
      - type: "external" # default
        url: "https://mysbomserver/mysbom"
    inputContracts: # [E]
      - id: dbb7b1eb-7628-436e-8914-2a00638ba6db

team:
  name: "Customer Team"
  members:
    - username: "ceastwood"
      role: "DPO"
```

Let's analyze the additions:

- I increased the version to v0.2.0 ([A]) and changed the status to active ([B]).
- I added an input port, `customerCrm` ([C]).
- I referenced the external SBOM [D].
- I linked the output port to the specific input port ([E]).

It may seem redundant to have the input ports mentioned twice. It is not. In the section [C], you list all the input ports and you can enrich their information with a description, specific versions, custom properties, tags, and authoritative definitions. In the output port's section on input port [D], you only reference the specific contract linked to this output port. It allows to fine grain your lineage.

Wrapping Up Manual Creation

Manual creation of data contracts and data products, and their maintenance, is not fun: you need to be precise and it's easy to make a mistake, especially with those long UUIDs. However, remember that you are in an iterative process where you extend them based on your needs. Those needs could be:

- Updated SLAs.
- Different data quality rules.
- Schema change (or drift).
- Adding contracts (output and input ports) to the products.
- And many more operations…

In the next section, let's use some tools to build data contracts and data products.

Deep Dive into the Standards

ODCS and ODPS are mature but evolving standards. This section is not a reference manual for the standards, focusing on a few elements that may not be obvious at first.

Using tags to specify PII, PHI, and other regulations

Data is becoming more and more regulated, especially in certain industries, and varying among different governments' desires to protect consumers. You've most likely heard of the following:

PII (personally identifiable information)
: Any data that can be used to identify a specific individual, such as name, address, social security number, email, or phone number.

HIPAA (Health Insurance Portability and Accountability Act)
: A U.S. federal law (1996) that sets national standards for protecting sensitive patient health information (see also PHI) from being disclosed without the patient's consent or knowledge.

PHI (protected health information)
: Any health-related information (medical records, diagnoses, treatment data) that is linked to an identifiable individual, protected under HIPAA.

GDPR (General Data Protection Regulation)
: A European Union regulation (effective 2018) that governs how organizations collect, store, and process personal data of EU residents, granting individuals strong privacy rights.

CCPA (California Consumer Privacy Act)
: A California state law (effective 2020) that gives California residents the right to know, delete, and opt out of the sale of their personal data collected by businesses.

To avoid a plethora of fields that would take a true/false value, Bitol recommends using tags, as shown in Example 2-6.

Example 2-6. Tagging a field

```
- name: "last_name"
  tags:
  - pii
  - gdpr
```

Version management

Semantic versioning (semver) is the backbone of both ODCS and ODPS. A version number follows the <major>.<minor>.<patch> pattern:

Major
: A breaking change. A column is removed, renamed, or its type changes in an incompatible way. Consumers need to adapt.

Minor
: A backwards-compatible addition. A new column is added, a new output port appears, or a new quality rule is declared, for example. Most of the time, consumers can ignore it safely.

Patch
: A backwards-compatible fix. A description is corrected, a tag is added, documentation improves. No structural change.

When no version is specified in a reference to a contract or a product, the service (see Experimentation) always returns the latest version. This is convenient for exploration but dangerous in production: pin your versions.

You can specify a version for an output port reference in a data product, as shown in Example 2-7. This matters when a producer offers multiple concurrent versions (see also Chapter 1, Figure 1-6) and you want your data product to consume a specific, stable version rather than whatever the latest happens to be on a given Monday morning.

Example 2-7. Pinning an output port to a specific contract version.

```
outputPorts:
  - name: customerTable
    description: "Customer Table"
    contractId: 34cae6d7-7648-38b2-8f66-8db79e1e2ce4
    version: v0.2.0   # pin to a specific version
```

The compare endpoint (discussed in the next section) can help you decide what the next version number should be: it computes the diff between two versions and suggests a semver bump automatically.

Experimenting

If you are reading this book, it probably means that you learn through reading. Some people learn better through videos, podcasts, blog posts, and so on. Whichever your favorite way of learning is, experimentation is the ideal complement.

However, it is not easy to build experiments on abstract concepts, so I built a set of services that you can use as a companion to this book. Let's dive in.

Exploring Some of the Services

To facilitate your work with data contracts and data products, I've created a set of services available online. This is not a required step, but the idea is to greatly simplify experimenting with data contracts and data products using ODCS and ODPS.

The examples in this section use a UNIX-like syntax and have been tested on a Mac running macOS Tahoe v26.3, but no specific feature of this version of macOS has been used. Familiarity with the `curl` command and the command line are recommended.

I highly recommend you copy/paste the example from an online source. If you are not reading this book on the O'Reilly e-learning platform, I recommend this article.

In the rest of this section, you will:

- Create an API key to access the services.
- Create a data contract using from a DDL source.
- Create a data product from a data contract.

Creating an API Key

For simplicity, I will export all key values as environment variables, so it will be easier to copy/paste the curl calls (which can be tedious), as shown in Example 2-8.

Example 2-8. First set of environment variables

```
export BITOL_URL=https://cloud.jgp.ai/api
export BITOL_USER_PW=BitolRu7ez!
export BITOL_USER_EMAIL=jgp@example.com
```

You will need a real email address, as it needs to be validated for recovery purpose.

The first step is to create an API key for you by registering to the service. Replace the values between the chevrons in Example 2-9.

Example 2-9. Creating an API key

```
curl -X POST "$BITOL_URL/v1/users" \
  -H "Content-Type: application/json" \
  -d '{
    "email": "'"$BITOL_USER_EMAIL"'",
    "password": "'"$BITOL_USER_PW"'",
    "firstName": "<<Your first name>>",
    "lastName": "<<Your last name>>",
    "company": "<<Your company>>",
    "dob": "<<your date of birth>>",
    "code": "I<3BDP",
    "comment": "<<Something nice about this book is always appreciated.>>"
}'
```

Only email and passwords are required. The code will unlock the service for readers of this book. Note that you will need a real email address as an email will be sent to you for validation.

The reply should look like Example 2-10.

Example 2-10. Receiving the API key

```
{"email":"jgp@jgp.net","firstName":"Jean-Georges", "lastName":"Perrin","com
pany":"Bitol","dob":"1971-10-05",
"comment":"I love Building Data Products.",
"createdAt":"2025-04-25T15:50:29.650725",
"updatedAt":"2025-04-25T15:50:29.650736",
"apiKey":"97d09209-9021-4b24-89a9-620aae063d40"}
```

The important part is the API key, which I recommend exporting as described in Example 2-11.

Example 2-11. Exporting the API key for easy reuse

```
export BITOL_API_KEY=97d09209-9021-4b24-89a9-620aae063d40
```

Of course: do not use this key, as it will not work. Use the one you just generated.

Creating a Contract from a DDL Script

Since we want to build a data product, let's first start with our promise, or the data contract. I will be using the basic example of a client and their addresses as we saw earlier in the chapter. You can find the files I use in the book's associated repository on GitHub.

Example 2-12 sends the DDL of my database to the service and creates the embryo of the data contract for me.

Example 2-12. Creating a data contract from a DDL script

```
cat resources/customer.sql | \
curl -X POST "$BITOL_URL/v1/contracts?sourceFormat=DDL&version=0.1.0&name=Customer
Contract&domain=Customer&tenant=QuantumClimate" \
     -H "X-API-KEY: $BITOL_API_KEY" \
     -H "X-USER-PASSWORD: $BITOL_USER_PW" \
     -F "file=@-"
```

Let's analyze the call:

- `sourceFormat=DDL` specifies that the format of the source the service is getting is the DDL.
- `version=0.1.0` specifies a semantic version (semver) number.
- `name=CustomerContract` is the name of the contract.
- `domain=Customer` is the business domain associated to this contract.
- `tenant=QuantumClimate` a specific tenant (or brand, or business unit, or department…).

The return is shown in Example 2-13.

Example 2-13. Result of the creation

```
{
  "domain":"Customer",
  "name":"CustomerContract",
  "id":"34cae6d7-7648-38b2-8f66-8db79e1e2ce4",
  "version":"0.1.0",
  "tenant":"QuantumClimate",
  "status":"draft"
}
```

The contract id is `34cae6d7-7648-38b2-8f66-8db79e1e2ce4` and you should have the same one, as the service is designed to be idempotent. Like for the other elements, let's export it to the environment, as shown in Example 2-14.

Example 2-14. Exporting the contract's id

```
export BITOL_CONTRACT_ID=34cae6d7-7648-38b2-8f66-8db79e1e2ce4
```

A quick note on UUIDs. Yes, UUIDs are ugly, unreadable, and complicated. But they are also a very powerful tool. Nevertheless, to make them a little more bearable, I will, as much as possible, use only the four last characters in the examples. So, when possible, instead of `34cae6d7-7648-38b2-8f66-8db79e1e2ce4`, I will use `2ce4`.

Example 2-15 asks the service for the contract.

Example 2-15. Retrieving the contract

```
curl -X GET "$BITOL_URL/v1/contracts/$BITOL_CONTRACT_ID" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" \
  --output $BITOL_CONTRACT_ID-0.1.0.odcs.yaml
```

If you omit the --output, you may get a long listing at times. You can now use your favorite editor (mine remains vi on the command line, but VSCode is a great alternative).

Example 2-16. Viewing and editing the contract

```
apiVersion: "v3.0.2"
contractCreatedTs: "2025-05-09 19:14:02.911 UTC"
dataProduct: ""
domain: "Customer"
version: "v0.1.0"
id: "34cae6d7-7648-38b2-8f66-8db79e1e2ce4"
kind: "DataContract"
name: "CustomerContract"
description:
  usage: "DDL upload for contract generation"
  purpose: "Defines schema based on uploaded DDL"
  limitations: "None"
schema:
- logicalType: "object"
  name: "Customer"
  physicalName: "Customer"
  physicalType: "table"
  properties:
  - logicalType: "Numeric"
    name: "customer_id"
    physicalName: "customer_id"
    physicalType: "SERIAL"
    primaryKey: true
    required: true
…
```

You can now use your favorite editor to edit and enrich it. Save it as version 0.1.1. You can then save it using Example 2-17.

Example 2-17. Uploading the contract

```
curl -X POST "$BITOL_URL/v1/contracts?version=0.1.1" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -F "file=@./$BITOL_CONTRACT_ID-0.1.1.odcs.yaml"
```

You will explore more services around contracts soon, but now let's build a first data product, leveraging the data contract we just generated and enhanced, as represented by Figure 2-5.

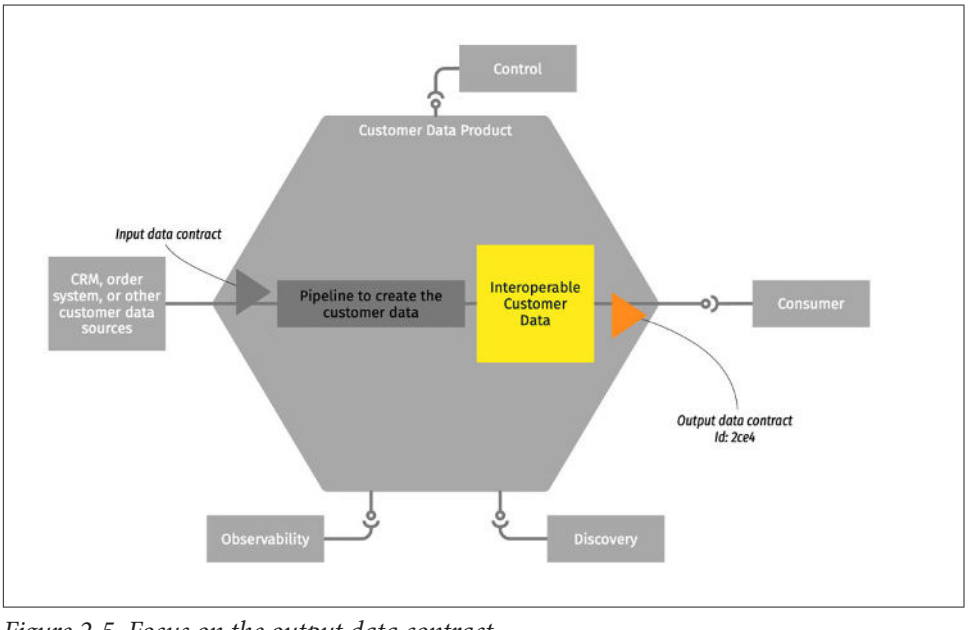

Figure 2-5. Focus on the output data contract

Creating a Product from a Contract

Now that we have a data contract, building a data product is also a matter of only a few API calls. Let's start by creating a product from the contract, as shown in Example 2-18.

Example 2-18. Creating a data product

```
curl -X POST "$BITOL_URL/v1/products?contractId=$BITOL_CONTRACT_ID&contractVer
sion=0.1.1" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW"
```

And the service should reply with the data product's id, just as in Example 2-19.

Example 2-19. Result of the creation, including the product id

```
{"name":"Default Data Product",
"id":"3153baf0-0c77-351d-b10c-a98578e10806","version":"0.1.0","status":"draft"}
```

As before, export the product id as a variable, as shown in Example 2-20.

Example 2-20. Exporting the product id

```
export BITOL_PRODUCT_ID=3153baf0-0c77-351d-b10c-a98578e10806
```

And now, you can retrieve it via another quick curl call, like in Example 2-21.

Example 2-21. Getting the product

```
curl -X GET "$BITOL_URL/v1/products/$BITOL_PRODUCT_ID" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" \
  --output $BITOL_PRODUCT_ID-0.1.0.odps.yaml
```

Finally, we get out data product, as illustrated in Example 2-22.

Example 2-22. Our first data product

```
apiVersion: "v1.0.0"
kind: "DataProduct"
# NOTE:
# The service is operational but only works with an old version
# of Bitol ODPS - This listing will be updated
```

Finally, you have completed your first data product with all the required elements, as illustrated by Figure 2-6.

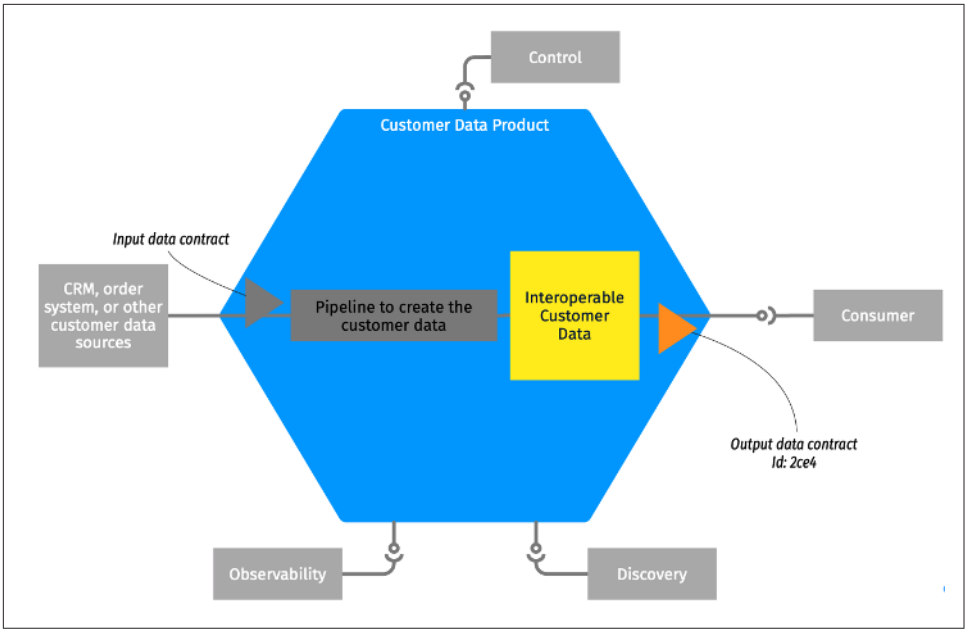

Figure 2-6. Your data product with a contract and its data

You can modify it manually, save it as version 0.1.1, and resubmit it via the service, as in Example 2-23.

Example 2-23. Uploading our modified data product

```
curl -X POST "$BITOL_URL/v1/products?version=0.1.1" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" \
  -F "file=@$BITOL_PRODUCT_ID-0.1.1.odps.yaml"
```

Other Useful API Calls

You have the basics: create, upload, list, and retrieve contracts and products. The service has a few more tricks worth knowing about before you close the terminal.

Listing all the data contracts

Example 2-24 lists all the data contracts stored by a user.

Example 2-24. Listing all the data contracts available to a user

```
curl -X GET "$BITOL_URL/v1/contracts" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" | jq
```

The `jq` utility formats the JSON output. If you do not have it on your system, you can remove the `| jq` at the end of the command. Output of the call is displayed in Example 2-25.

Example 2-25. All the data contracts

```
[
  {
    "id": "34cae6d7-7648-38b2-8f66-8db79e1e2ce4",
    "version": "v0.1.0",
    "name": "CustomerContract",
    "domain": "Customer",
    "tenant": "QuantumClimate",
    "status": "draft"
  },
  {
    "id": "34cae6d7-7648-38b2-8f66-8db79e1e2ce4",
    "version": "v0.1.1",
    "name": "CustomerContract",
    "domain": "Customer",
    "tenant": "QuantumClimate",
    "status": "draft"
  }
]
```

You can also list all the data products stored by a user, as shown in Example 2-26.

Example 2-26. Listing all the data products available to a user

```
curl -X GET "$BITOL_URL/v1/products" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" | jq
```

The output is displayed in Example 2-27.

Example 2-27. All the data products

```
[
  {
    "id": "3153baf0-0c77-351d-b10c-a98578e10806",
    "version": "v0.1.0",
    "name": "Default Data Product",
    "domain": "",
    "tenant": "",
    "status": "draft"
  },
  {
    "id": "3153baf0-0c77-351d-b10c-a98578e10806",
    "version": "v0.1.1",
```

```
    "name": "Default Data Product",
    "domain": "",
    "tenant": "",
    "status": "draft"
  }
]
```

Recovering a lost API key

Lost your API key? It happens, especially when you forget to export it and close the terminal before you should have. No drama: you can retrieve it as long as you know your email and password. Note that there is no way of recovering your email or password. Example 2-28 shows how.

Example 2-28. Recovering a lost API key

```
curl -X GET "$BITOL_URL/v1/users/key?email=$BITOL_USER_EMAIL" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW"
```

The response gives you back your key as a JSON object. Re-export it and carry on.

Comparing two contract versions

One of the most useful operations when maintaining contracts over time is understanding exactly what changed between two versions, and, most importantly, whether that change is breaking. The compare endpoint in Example 2-29 does precisely that: it returns a structured diff with a semantic versioning impact level (major, minor, or patch) for each difference, and suggests the next version number.

Example 2-29. Comparing two versions of the same contract

```
curl -X GET "$BITOL_URL/v1/contracts/compare \
  ?id1=$BITOL_CONTRACT_ID&version1=0.1.0 \
  &id2=$BITOL_CONTRACT_ID&version2=0.2.0" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" | jq
```

A trimmed response might look like Example 2-30.

Example 2-30. The diff result showing a minor change and the suggested next version

```
{
  "diffs": [
    {
      "type": "ADD",
      "section": "schema",
      "field": "quality",
      "path": "/schema[0]/properties[2]/quality",
```

```
      "message": "Quality rule added",
      "impactLevel": "MINOR"
    }
  ],
  "major": 0,
  "minor": 1,
  "patch": 0,
  "suggestedVersion": "0.3.0"
}
```

The service is telling you that adding a quality rule is a backwards-compatible addition (`MINOR`), and that your next version should be 0.2.0. A removed or renamed column, on the other hand, would trigger a `MAJOR` bump — your consumers' pipelines will break, and the version number should warn them.

Assessing contract maturity

All data contracts are created equal, but they can evolve differently. A contract with a name and three column names is technically valid ODCS, but it delivers less value than the one with all description filled, SLA set, and a few data quality rules in place. The maturity endpoint in Example 2-31 scores your contract on a scale from 1 to 5 and tells you exactly what you need to do to reach the next level.

Example 2-31. Measuring the maturity of a contract

```
curl -X GET "$BITOL_URL/v1/contracts/maturity-level \
  ?contractId=$BITOL_CONTRACT_ID&version=1.2.0" \
  -H "X-API-KEY: $BITOL_API_KEY" \
  -H "X-USER-PASSWORD: $BITOL_USER_PW" | jq
```

The response includes the level (say, 3 — Governed), a Markdown explanation of what criteria were satisfied, and concrete advice on what you need to add to reach level 4. Think of it as a rubric for your contract: useful to run after each enrichment cycle to make sure you are making meaningful progress and not just adding noise. More details will come later, when we tackle the lifecycle in Chapter 9.

Other Standards

Bitol's ODCS and ODPS are not the only kids in town. Four other specifications are worth knowing: *DataContract.com* (Harrer & Christ), the *Data Product Descriptor Specification (DPDS)* from Open Data Mesh, the *Open Data Product Specification* from opendataproducts.org, and the *Data Products Ontology (DPROD)* from the Object Management Group. Each took a different angle. Table 2-1 summarizes the key differences; the notes in the following subsections cover what the table cannot.

The space has been consolidating. DataContract.com's and DPDS's authors are now on the Bitol TSC. The Open Data Product Specification and DPROD serve distinct enough use cases that they complement rather than compete. For internal data engineering, ODCS and ODPS are the clear choice. For some use cases, the others are worth a look and are not incompatible with Bitol's contract layer.

Table 2-1. Comparison of data product and data contract specifications

Aspect	Bitol ODCS / ODPS	DataContract.com	DPDS	Open Data Product Spec	DPROD (OMG)
Primary focus	Contracts + products, governance & lifecycle	Contracts + minimal product descriptor	Full-stack: interface, app & infra	Products incl. commercial & monetization	Semantic ontology for data product discoverability
Format	YAML	YAML	JSON / YAML	YAML	RDF / OWL / SHACL (W3C Linked Data)
License	Apache 2.0	MIT	Apache 2.0	Apache 2.0	OMG standard
Governance	Linux Foundation (Bitol TSC)	Sunsetted; authors on Bitol TSC	Open Data Mesh; author on Bitol TSC	Linux Foundation	Object Management Group (OMG)
Current version	ODCS v3.1.0 / ODPS v1.0.0	v1.2.1 / v0.0.1	v1.0.0-DRAFT	v4.1 (Oct 2025)	Beta 1 (Feb 2025)
Best for	Internal data engineering & governance	Migration path → ODCS / ODPS	Mesh platforms with automated deployment	External / commercial data products & marketplaces	Enterprise knowledge graphs, federated data marketplaces
Community status	Active, cross-industry TSC	Legacy; still widely referenced	Small; converging into Bitol	Active and growing	Early stage; formal OMG process
Relationship to Bitol	—	Authors joined Bitol TSC	Author joined Bitol TSC	Supports ODCS by reference	Complements; same contributor (Gioia)

DataContract.com

Dr. Simon Harrer and Jochen Christ created the DataContract.com website, as well as two specifications, one for data contracts and one for data products. Harrer and Christ were early contributors to Bitol through the TSC and, with the release of ODCS v3.1.0 and ODPS v1.0.0, they decided to sunset their specifications in favor of Bitol's. The migration path is short. If you are still on their specs, I'd recommend you move on.

DPPS

As of now, Quantyca's DPDS goes deeper than ODPS: it describes not just what a data product exposes, but how it is built including deployed interface, application, and infrastructure components in a single Apache 2.0-based descriptor. It aligns with

OpenAPI and AsyncAPI, which feels natural to engineering teams already in those ecosystems.

The story of DPDS, however, is a convergence story. In May 2025, its author Andrea Gioia joined the Bitol TSC:

> "Now, I'm happy to join the TSC at Bitol, where I'll contribute to driving the evolution of ODCS, promoting harmonization across data specifications, and fostering a more interoperable data ecosystem that emphasizes value creation over tool integration."

DPDS remains available, and its ODM Platform is worth studying for full-lifecycle automation. For new projects, ODPS is the forward bet, one Gioia is now actively shaping.

Open Data Product Specification

Fair warning: there are two specifications called ODPS. The Open Data Product Specification, led by Dr. Jarkko Moilanen and Linux Foundation-hosted since May 2024, has used that acronym longer than Bitol. The conversation about who owns those four letters is, let's say, irrelevant.

At version 4.1 (October 2025), it serves a genuinely different use case: external monetization. Its 120+ attributes cover technical, business, legal, and ethical dimensions, with 12 pricing plan templates. It also supports ODCS contracts by reference, so the two are not mutually exclusive. If you're building an internal engineering platform, Bitol wins. If you're selling data externally, this one is worth a serious look.

Data Products Ontology (DPROD)

DPROD operates at a different layer entirely. Published by the OMG as a proposed specification in September 2024 (Beta 1, February 2025), it defines data products as a formal ontology built on W3C Linked Data standards (RDF, OWL, SHACL), extending the W3C DCAT vocabulary. If your toolchain is YAML and REST APIs, DPROD will feel distant. If your organization runs on enterprise knowledge graphs, federated data marketplaces, or linked-data infrastructure, DPROD offers semantic interoperability that YAML specs cannot match (yet). Andrea Gioia is a contributor here too, which at this point is less surprising than inevitable.

Summary

Did data finally get its HTML moment? Here's what you learned in this chapter:

- The HTML moment for data: Data standards are at the same inflection point the web was in the 1990s. Without them, every team builds incompatible silos. ODCS and ODPS are data's HTML.

- Bitol was almost an accident. It started as a PayPal data mesh resource descriptor, and was open-sourced in May 2023. The Linux Foundation is now hosting it, and a 15ish-person TSC from almost as many companies turned a template into a proper standard.
- Two standards, one mission. ODCS defines what a dataset promises (schema, quality, SLAs). ODPS wraps it into a deployable data product with ports. Both are YAML, human-readable, and machine-consumable.
- Data contracts: start small, keep enriching. A bare-minimum contract (kind, version, id, schema) already provides value. Add quality rules, ownership, and SLAs over time. A contract is never finished.
- SLAs without anchor fields are just poetry. A *three-year retention* SLA is unenforceable without a date field to measure against. No timestamp, no guarantee.
- Semantic versioning everywhere: Major breaks, minor adds, patch fixes. Always pin contract versions in production. "Latest" is fine for exploration. It is a trap on Monday morning (or Friday evening).
- Tags beat Boolean fields for compliance. Use tags (`pii`, `gdpr`, `phi`…) instead of a `true`/`false` field per regulation. Extensible and future-proof.
- Version references can be pinned on output ports. A data product can reference a specific contract version per output port. No surprises when a producer ships a new version on a random Wednesday (Fridays are worst).
- The standards landscape is consolidating. DataContract.com is sunsetted; its authors joined Bitol TSC. DPDS is converging too. DPROD is an RDF ontology for enterprise knowledge graphs.
- Trust the API to do the heavy lifting. A free companion service at *cloud.jgp.ai* lets you generate a contract from a DDL script, create a product from a contract, and manage versions via `curl`.
- The compare endpoint tells you what broke. Diff two contract versions, get a semver-based label per change, and receive a suggested next version number. Useful before every release.

By now, you should be so excited that you can't wait to read Chapter 3!

Further Reading

Niederst, J. (1998). Web Design in a Nutshell. Sebastopol, CA, USA: O'Reilly Media, Inc.

Roff, J. T. (2001). ADO: ActiveX Data Objects. Sebastopol, CA, US: O'Reilly Media, Inc.

CHAPTER 3

Imagining a Data Product

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *ccollins@oreilly.com*.

Imagine trying to describe a car to someone who's only ever seen bicycles. That's where we are with data products. We're used to datasets, dashboards, maybe a pipeline or two, but the idea of a product built from data still feels fuzzy to many. In this chapter, you'll explore what makes a data product a product, how to imagine and architect one, and how it fits into a larger system of contracts and Mesh. Whether you're building from scratch or renovating a legacy monster (brownfield alert!), this chapter will sharpen your mental model and show how reuse, contracts, and architecture come together to form something… hopefully useful.

Building Your Mental Model

There is a lot of confusion about picturing what a data product is. I hope that Chapter 1 helped. However the versatility of a data product calls for a little more in-depth discussion, which I wanted to have with you.

Location of the Data

You: where is the data in a data product?

Jean-Georges (jgp): starting with the tough question right away! I like that.

Remember that a data product is a logical construct above your data, so the data can be anywhere. Let's look at a few scenarios.

In Figure 3-1, the data lives inside the data product. This is the most common representation practitioners have. However, when working with brown field projects (more details in this chapter), you may realize that the data is outside of the data product, like illustrated by Figure 3-2.

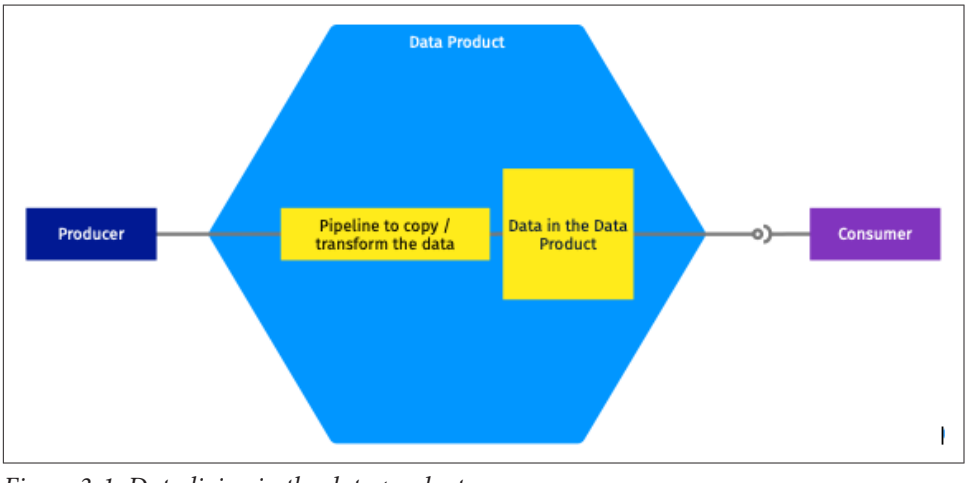

Figure 3-1. Data living in the data product

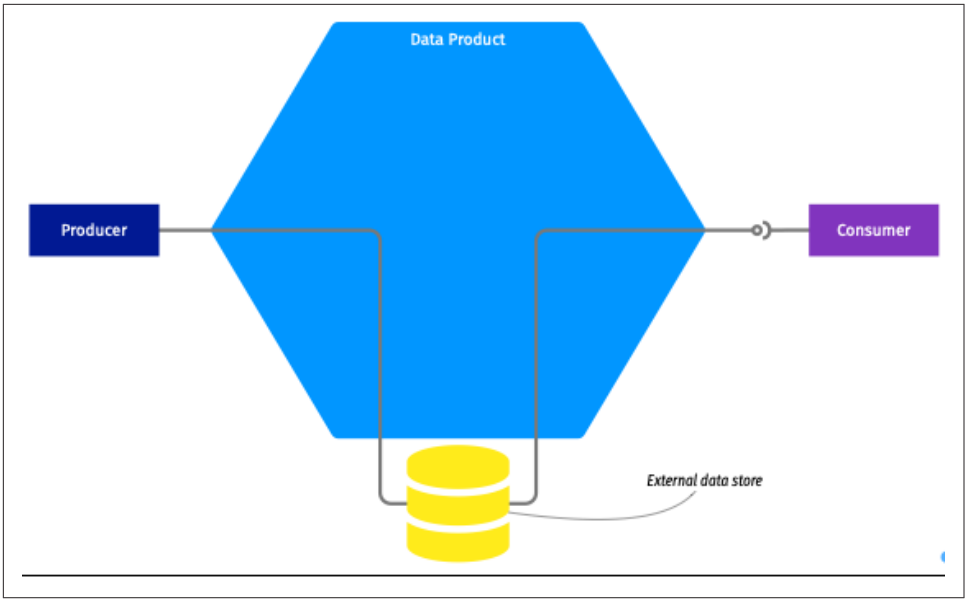

Figure 3-2. Data living outside the data product

It may trigger a visceral reaction questioning my sanity but remember that the data product is a logical construct. It offers similar services whether the data is within the physical scope of the product or not. Figure 3-3 illustrates this idea with data available within the product, externally, and virtualized.

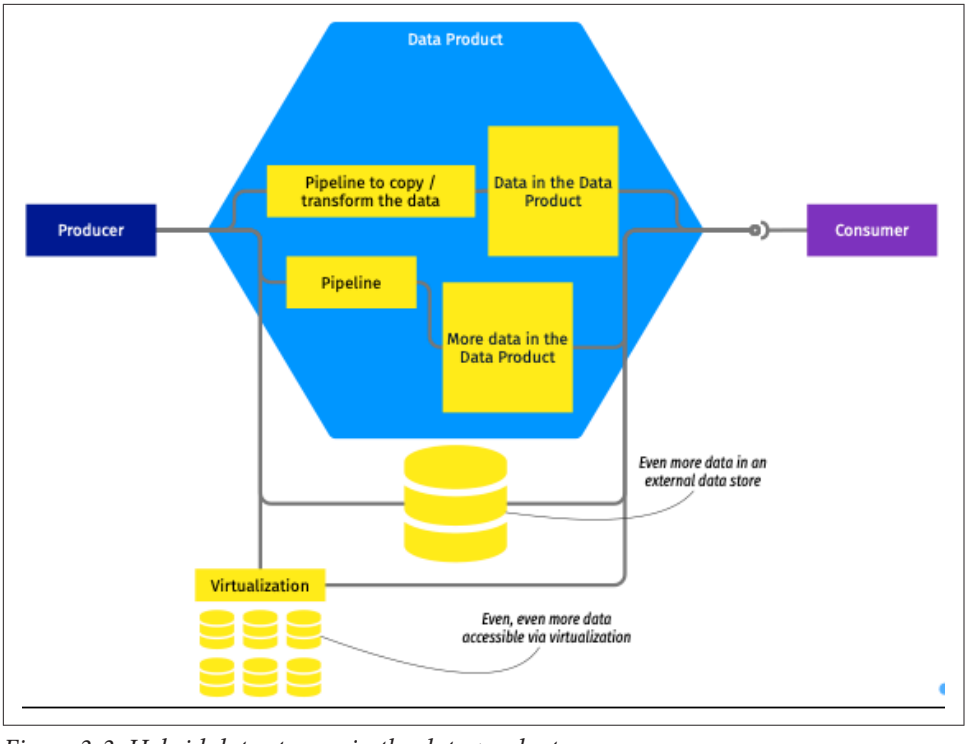

Figure 3-3. Hybrid data storage in the data product

Types of Data Products

You: are there different types of data products?

jgp: Of course there is, why would we keep things easy?

Let's preface by saying that the types are more a logical placement than a real difference in the data product itself. The two main types are producer aligned and consumer aligned.

Producer-Aligned Data Products

Producer-aligned (or source-aligned) data products reflect the data in a very similar as the system the data is coming from. Data may be reorganized for easier usage, but it is not a requirement.

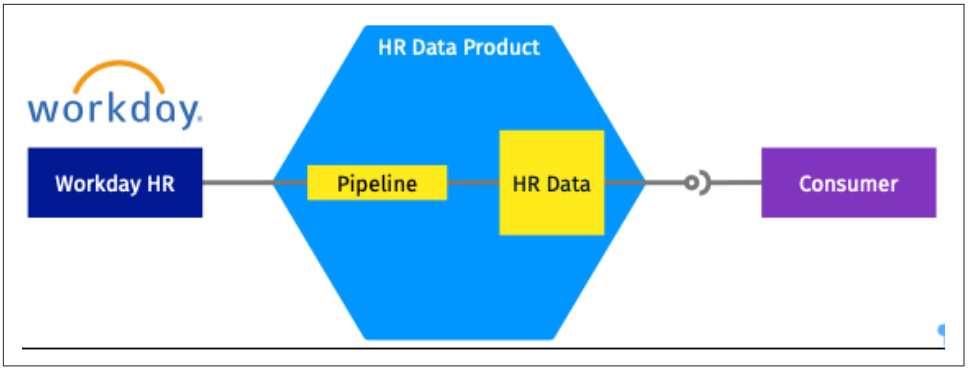

Figure 3-4. An HR data product, aligned with the source system, here Workday.

You can use a single data product, like in Figure 3-4 where the entire data is accessible via a single data product. You can also decompose your source system in multiple data products like illustrated by Figure 3-5.

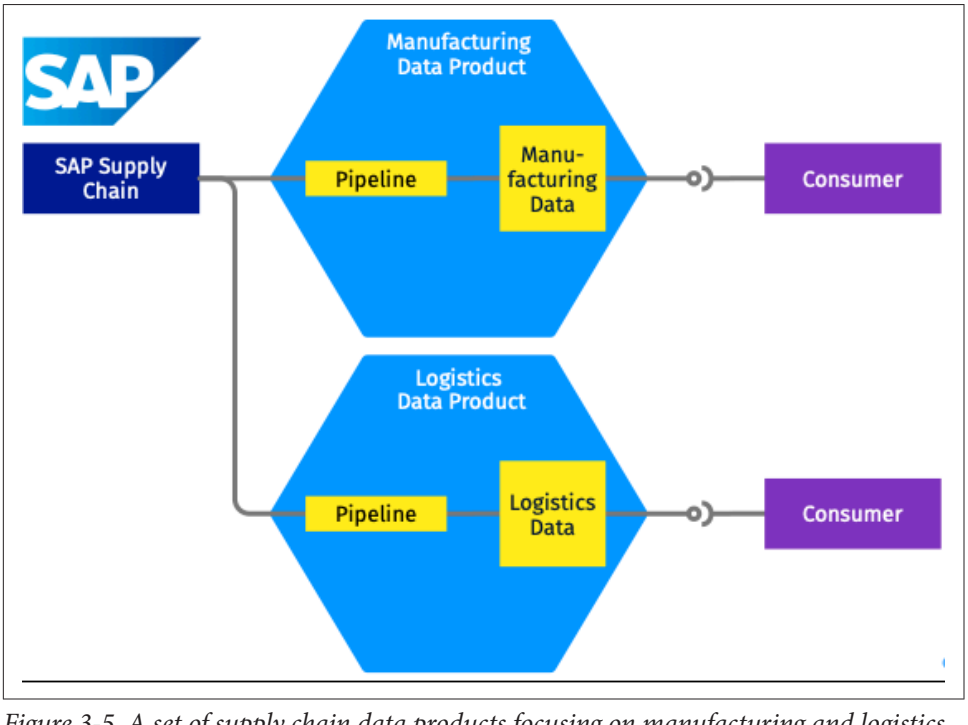

Figure 3-5. A set of supply chain data products focusing on manufacturing and logistics, aligned with the source system, here SAP Supply Chain.

Those data products can be considered foundational as everything comes after them.

Consumer-Aligned Data Products

Consumer-aligned (or use case-aligned, domain-aligned) data products are not only closer to the consumer, but they also align to a domain or use case.

In Data Mesh Radio #151, Vista's Jillian Maffeo confirmed that it was ok to align a use-case to a data product. It was my experience as well, but she was the first time I heard someone sharing it publicly. It made sense then; it still makes sense now. I classify the consumer-aligned data products as use-case oriented, domain-oriented, or cross-domain.

Figure 3-6 illustrates a consumer-aligned data product based on a use case that needs fleet management data and supply chain data. The consumer-aligned data product's input data are defined by the upstream data products' data contracts.

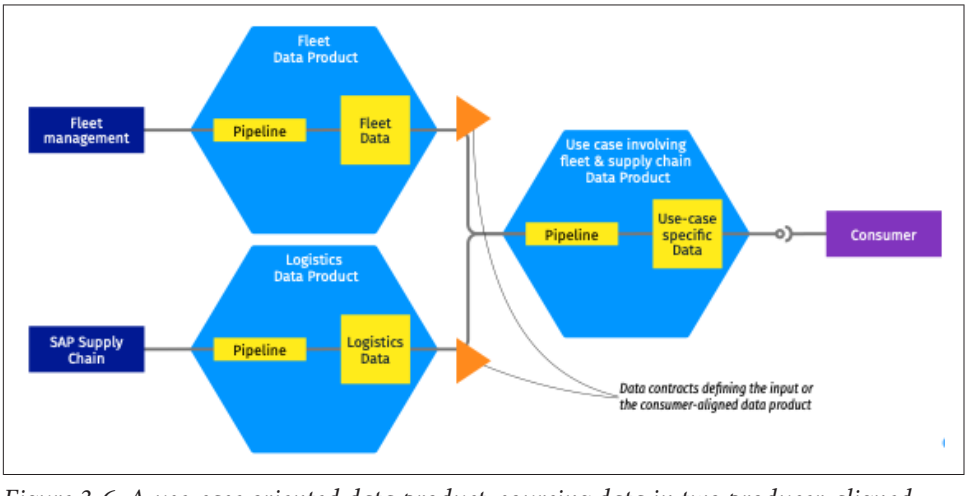

Figure 3-6. A use-case oriented data product, sourcing data in two producer-aligned data products.

AI & ML Data Products

Are you surprised to see AI & ML data products? I am sure you are not as you probably suspect that AI & ML have special needs like models, versions of models, training datasets, and more. Do they all belong in a data product?

All those assets are part of a data product, as shown by Figure 3-7. As with other data products, the data can be sourced from a producer or upstream data product. The differences start to appear with data. Your AI & ML data product can contain the training data alongside the data used by the model (you could even imagine having the validation data). The pipeline should naturally support this design. Finally, you can imagine having multiple versions of the models, as you know the saying, a dead model is a model that is not maintained.

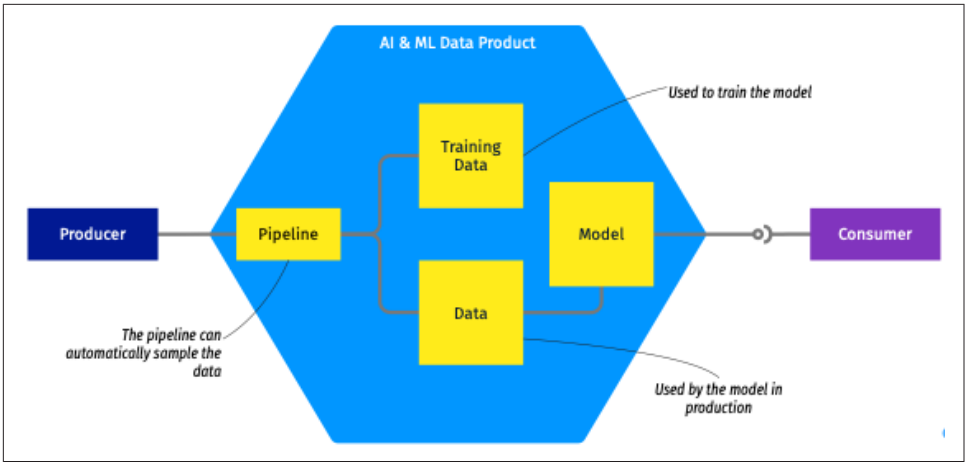

Figure 3-7. Focus on the specifics on an AI & ML data product.

Data Contract vs. Data Product

You: Are data contracts and data products the same thing?

jgp: Nope.

Data contracts and data products are like inseparable cousins—always working together, always aligned, and always making sure things run smoothly. One ensures that data is structured, reliable, and consistent, while the other packages and delivers that data for valuable consumption. If data were a factory, data contracts would be the quality control process, ensuring that only properly formatted, high-quality materials enter production. Meanwhile, data products would be the final packaged goods, ready for consumers to use and trust.

Data Contract

A data contract is a formal agreement between (usually) one data producer and data consumers that defines expectations around data structure, quality, meaning, and availability. It ensures that data meets technical guarantees and aligns with business definitions, reducing inconsistencies across teams.

Following these characteristics, a data contract is the source of truth for your metadata.

Data contracts are represented by a 90°-rotated equilateral triangle, symbolizing its three foundational components, as illustrated by Figure 3-8.

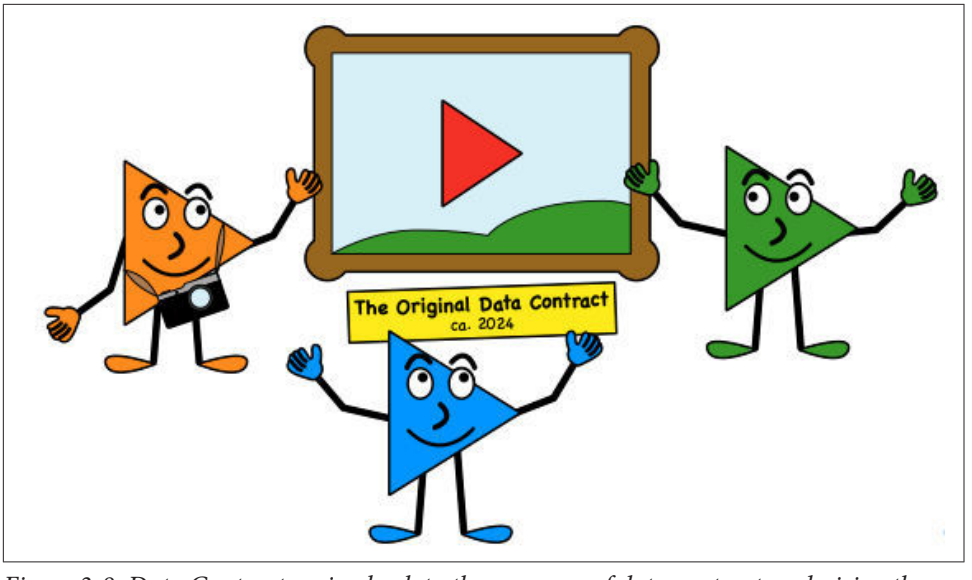

Figure 3-8. Data Contracts going back to the museum of data contracts, admiring the first documented contract.

Data Product

As we discussed in Chapter 1, a data product is a reusable, active, and standardized data asset designed to deliver measurable value by applying product thinking to data management. Unlike a data contract, which is a formal agreement, a data product is an actual deliverable — something users interact with and derive value from.

Here are some characteristics of a data product:

Composed of data artifacts
: Includes datasets, models, pipelines, and probably more. Artifacts can have multiple versions.

Output ports
: A data product exposes its data through well-defined output ports, ensuring downstream users or systems can easily consume it.

One (output) data contract per output port
: As a data product may expose multiple output ports, it will require one data contract per output port.

Input data contracts
: They are recommended to ensure that what is coming in the data product can be trusted.

Software Bill of Material (SBOM)
: Designed to document precisely how the transformation is performed.

Its shape? A horizontal hexagon represents the interconnected nature of hexagonal architecture. Figure 3-9 shows an open interpretation to data products.

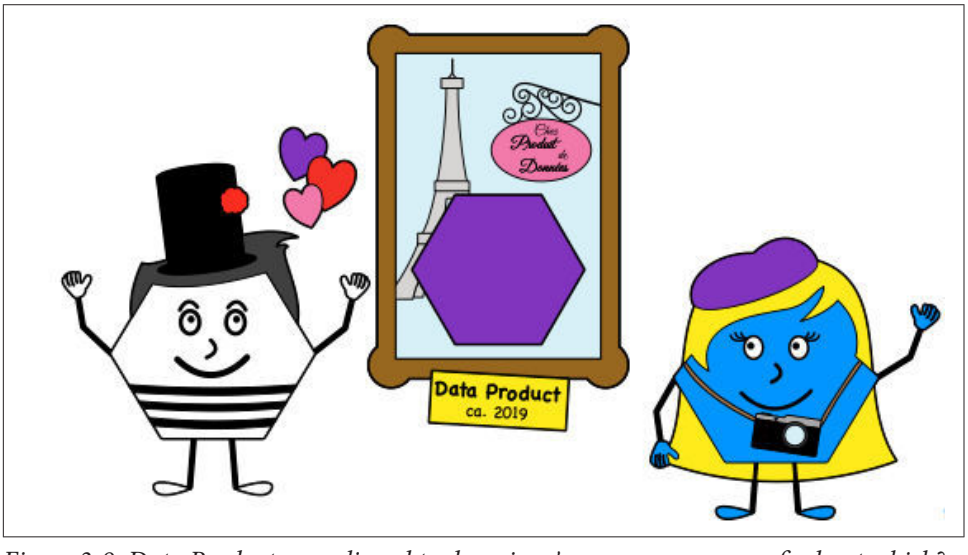

Figure 3-9. Data Products are aligned to domains & use-cases, can you find out which?

Table 3-1 summarizes the difference between a data contract and a data product.

Table 3-1. Comparison of a data contract & a data product.

Aspect	Data Contract	Data Product
Quick Definition	A formal agreement defining expectations for data structure, quality, and delivery.	A packaged, standardized, and reusable data asset delivering business value.
Purpose	Ensures reliability, trust, and consistency in data pipelines.	Provides actionable insights, analytics, or operational value to users.
Scope	Defines guarantees for a dataset.	Exposes data through output ports with a single associated data contract.
Activity	Passive: used by applications as a resource descriptor.	Active: offers API for discovery, control, and observability.
Components	Schema, SLAs, ownership, validation rules.	Data artifacts, metadata, governance policies, output ports.
Standardization	Bitol ODCS	Bitol ODPS
Lifecycle	Core is typically static once agreed upon, with occasional updates. Data QoS (quality rules and SLA) evolve.	Continuously refined and improved based on feedback.

Aspect	Data Contract	Data Product
Shape		
	Equilateral Triangle (rotated 90°) – symbolizing schema, business meaning, and SLAs.	Horizontal Hexagon – symbolizing interconnectedness, multiple output ports, and usability.

Role of the Data Contract

You: What is the role of a data contract in a data product?

jgp: Data contracts prevent misalignment between technical teams and business stakeholders by explicitly defining what the data represents and how it should behave. They reduce schema drift, ensure compliance, and foster trust in data pipelines.

> Imagine an analytics team relying on a database's `customer_status` field. A data contract guarantees that the field exists and defines what each status means from a business perspective, ensuring consistency across teams.

Data contracts are defined using the Open Data Contract Standard (ODCS), a mature standard that ensures structured agreements between data producers and consumers.

In a way, the contract is the promise while the product is the delivery.

Lifecycle

You: How should I think about the lifecycle?

jgp: Aren't you relentless?

The most important to remember is that both a data contract and a data product, and their YAML description, are living elements. If I ever tell you that "this data contract is finished," you may question my sanity.

Data contracts are the source of truth for your meta data. As such, they need to evolve as your meta data evolve: documentation evolves all the time, additional data quality rules, different SLAs for various stakeholders or drivers, and more.

For both data contracts and data products, Bitol recommends five values for the status of the lifecycle:

Proposed
: Early stage of a data contract or product, at the ideation stage of the process. This is mostly seen in the "green field" data contract/product. A data product owner is not required at this stage.

Draft
: As your data contract/product is development, it is not yet active.

Active
: When the data contract or product is ready to go to production. It becomes active. In this state, the data contract/product will evolve, and its version will really matter. The version follows semantic versioning.

Deprecated
: At this stage, the data contract/product will solely be on maintenance. The SLA fields should indicate when the data contract/product enters the various stages using the `generalAvailability`, `endOfSupport`, and `endOfLife` properties. It can still be used but I would not recommend building new application or report on this data.

Retired
: The data contract/product does not exist anymore. Its documentation should indicate which data contract/product replaced them.

Figure 3-10 shows the process graphically. There is no notion of how much time a contract or product should stay in each phase.

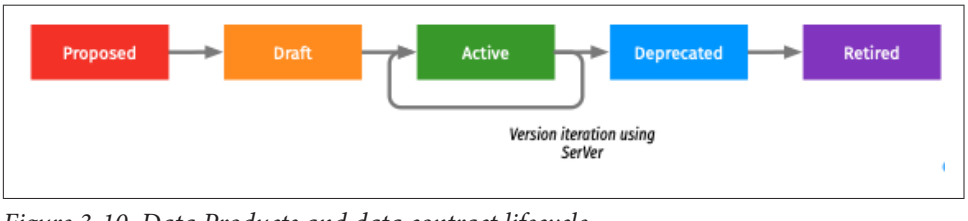

Figure 3-10. Data Products and data contract lifecycle

Thinking Reusable in Reusable Components

Data products are designed for reusability. It is true for data consumers, but it is also true for their internal components. Contrary to a dataset, a data product is an active data asset, as I described it in Chapter 1: this means that it can actively interact through its data and its metadata. Figure 3-11 illustrates them.

A sidecar in software engineering is a set of helper services that run alongside your main application, like a buddy sitting in the sidecar of a motorcycle. It's separate from your app but works closely with it—common examples are logging, metrics, or service mesh proxies. Unlike a library, which is code you import into your app and call directly, or a framework, which often takes control and tells your code when to run (inversion of control), a sidecar does its job independently, communicating with your app over the network or through shared files. Think: library = tool in your hand, framework = machine you step into, sidecar = companion riding next to you.

In a data product, there are two sets of components:

- The mutable components which require adaptation every time you build a new data product. They include: data contracts (input and output, ODCS files), data product description file (ODPS file), the pipeline, and the design of the data store. Note that in the case of retrofitting an existing dataset and pipeline (brown field), you will only need the data contract(s) and the data product description.
- The immutable components are reusable across every data products implementation. They include the control, observability, and discovery services, with their associated sidecar and clients. I will detail the implementation in Chapter 4, but you can already imagine, at the lower level, a (Docker) container with the necessary services and storage that lives in your infrastructure.

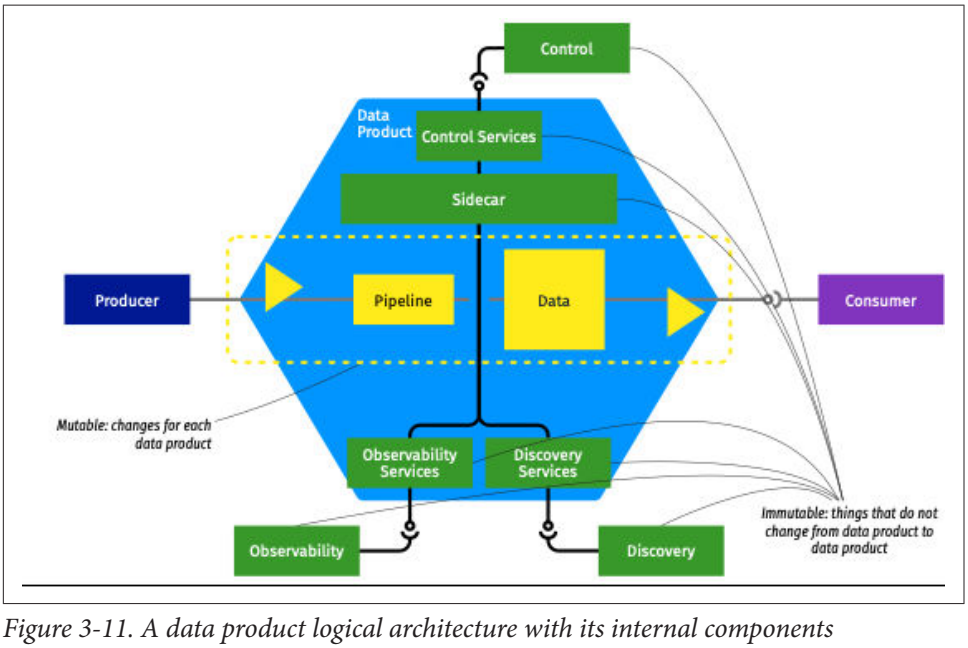

Figure 3-11. A data product logical architecture with its internal components

Following the Holy Trinity

In this context, the Holy Trinity is composed of data contracts, data products, and Data Mesh. At times, you may want to remind yourself why we are doing all this work. Hopefully your answer is the constant journey for extracting value from your data. True, it should have been the case for all data projects, but how much time does it take?

Although the average data warehouse or data lake project can take months or years to get value, following a product-oriented data engineering & management (PODEM) methodology can reduce the time to value to weeks.

You start by implementing data contracts. You will then have:

- Better documentation, as you centralize the documentation. Engineers, subject matter experts (SMEs), and owners can collaborate incrementally on a single version of the documentation.
- Better data quality and service levels, aka Data QoS. I cover Data Quality of Service more extensively in Chapter 8.
- Faster discoverability, as your data contracts can easily imported in various tools supporting ODCS, like Actian's Data Intelligence Platform or Entropy Data's Data Contract Manager.

You can then switch to data products. You will have more value:

- Better isolation to a domain or a use-case.
- Faster time to market than with traditional datasets.
- Easier evolution as you are thinking product and not projects.
- Higher security thanks to better isolation.
- Standardized way of consuming data using open standards and API.
- Consistent view of observability, not depending on the plethora of tools you use.

The last element is Data Mesh. The additional value Data Mesh provides is:

- Better cross-domain communication thanks to the data contracts flowing between the data products.
- Enhanced lineage thanks to the rich metadata contained in each data product and data contract. Lineage comes naturally in this condition and should always be accurate.
- Enterprise standard to access & consume data.

- Federated data governance, thanks to centralized policy and distributed expertise.

In addition to this horizontal progression in value, you can also increase value as you increase the information inside and deploy more data contracts and products. Figure 3-12 illustrates the value growth.

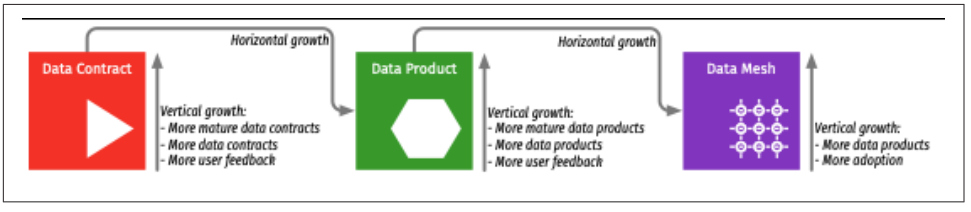

Figure 3-12. Value growth with data contracts, data products, and Data Mesh

Table 3-2 summarizes the purpose, benefits, and outcome of data contracts, data products, and Data Mesh. It also reminds you of the standards behind each element and the shapes.

Table 3-2. Benefits & outcome of data contracts, data products, and Data Mesh.

Aspect	Data Contract	Data Product	Data Mesh
Purpose	Define clear, formal agreements on data structure, meaning, and SLAs between producers and consumers	Package, deliver, and maintain reusable, value-driven data assets	Federate data ownership across domains to scale and democratize data access
Benefits	Reduces schema drift, improves trust, enables automation, and observability	Faster time to market, better domain alignment, API-driven data reuse	Organizational scalability, domain autonomy, and enterprise-wide governance
Outcome	Shared understanding, improved data quality, metadata-first thinking	Self-serve, well-documented, production-grade data with lifecycle management	Streamlined collaboration, standardized access, and lineage at scale
Standardization	**Bitol ODCS**	**Bitol ODPS**	**Bitol ODMS**
Shape	 Equilateral Triangle (rotated 90°) – symbolizing schema, business meaning, and SLAs.	 Horizontal Hexagon – symbolizing interconnectedness, multiple output ports, and usability.	 The Aegean symbol representing 90,000, illustrating the vast quantity of interconnected data products you can have in your instance of Data Mesh.

Harvesting Value from Green & Brown Fields

In the previous section, I reminded the fundamentals of why we engineer and manage data: extract value. Many compare that work to the oil extraction and oil is extracted from fields, right?

Greenfield means you're starting from scratch—no legacy code, no constraints, just a blank canvas (and a lot of hope). It's like building a shiny new app on a fresh plot of land. Brownfield, on the other hand, means you're working within an existing system—there's already data, code, architecture, maybe even tech debt—and you need to build on top of it or alongside it. It's like remodeling an old house: expect surprises in the plumbing.

In my career, and in many of those bearing the color of wisdom in their beard or hair, there is this sentiment that greenfield is utopia, while brownfield is realism. So why are most vendors trying to sell you solutions that only work in greenfield? Because it is easier, much easier. However, it is far from the reality.

Figure 3-13 describes the greenfield approach to building data products, while Figure 3-14 focuses on the brownfield.

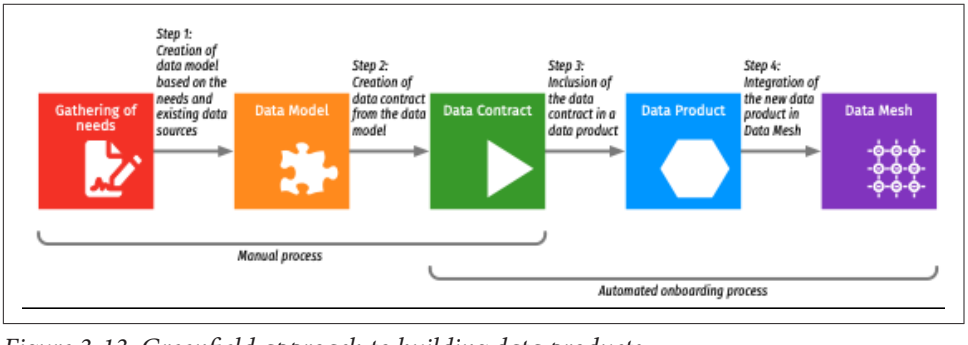

Figure 3-13. Greenfield approach to building data products

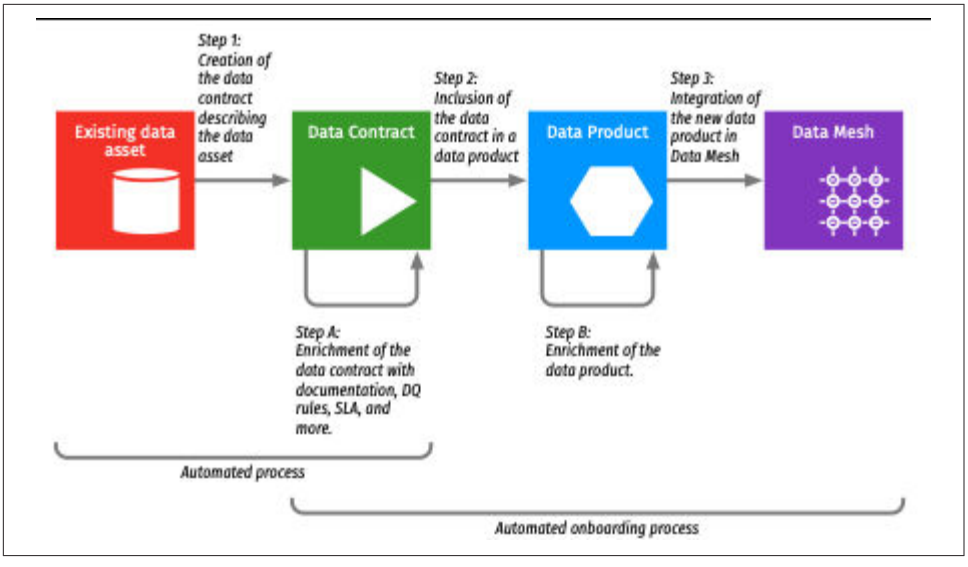

Figure 3-14. Brownfield approach to building data products

As you can see, the greenfield approach is very similar to a waterfall project, where the team needs to gather requirements and design a data model. Those tasks can be heavy time consumers.

The brownfield approach relies on interpreting automatically what the previous teams wanted and onboards existing assets automatically, drastically reducing the time to delivery and value.

Of course, the world is not only black and white (or should I say green and brown?) and there are some middle grounds to be found. A bank I worked with will not automatically convert their 35,000 pipelines or a fintech its 9 million tables… However, if you are a CDO and want to show value quickly, there is a way.

Then, as you are in product mode, enhance your product.

Summary

In this chapter, you should have been able to create or update a mental model for data products, include their stake holders and lifecycle:

- A data product is a logical construct-it doesn't own the data, but it does own how that data is exposed and managed.
- There are multiple types of data products: producer-aligned, consumer-aligned, and even AI/ML-centric, each tailored for specific use cases and data sources.

- Data contracts and data products are different but complementary: think quality control vs. finished goods.
- Lifecycle matters! Both contracts and products evolve: they're living metadata with stages like Proposed, Draft, Active, Deprecated, and Retired.
- The architecture of a data product includes mutable (custom) components and immutable (reusable) ones. Think plug-and-play meets tailored craftsmanship.
- The "Holy Trinity" is composed of data contracts, data products, and Data Mesh. They form the backbone of product-oriented data engineering and management.
- Whether you're building greenfield (clean start) or brownfield (legacy chaos), there's value to unlock-and faster than you think.

CHAPTER 4
Building Your First Data Product

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *ccollins@oreilly.com*.

So, you want to build your first data product. Congratulations! You're about to move from theory into practice, from hand-wavy "mesh talk" to something you can actually poke at with an API call. This chapter is your first lap around the track: I'll define who you're building for, what roles they play, and how to organize the work so your product doesn't end up like a group project where only one person does the homework.

You'll meet your main personas (Cindy, Juliette, Beth), map out responsibilities with a RACI (the adult version of "calling dibs"), and split the work into four pragmatic streams: platform, runtime, infrastructure, and consumer experience. You'll also get a tour of the sidecar (the brain of your product), its internal database, and the first endpoints that make your data product feel alive.

By the end, you'll understand not just how to sketch a data product, but how to get a minimal, running version into the world—whether you're starting fresh (green field) or reshaping what you already have (brown field).

Who Are We Building For?

The primary group of personas includes the people you are dealing with daily. As a data product builder, they are your main customers. Treat them well. There is also a secondary group of users that may appear here and there.

Primary Actors

Let's start by examining who we are building data products and their infrastructure for. Since I started my journey on product-oriented data engineering and management (PODEM), three strong women have constantly been by my side.

Under her pure New England flannel shirt, Cindy is an experienced data engineer. She discovered programming through Perl and, for sanity purposes, quickly moved to Python. She is now mastering ETL and ELT pipeline, as well as tools like dbt, Talend…

Her pet peeves? She barely has time to learn as she is always firefighting damage from upstream and downstream teams. She'd like to invest in data quality but between the home-grown framework and the multitude of attempts at implementing open source frameworks, she's feeling like at the bottom of the Everest.

You want to make her explode? Ask her for clean data without defining *clean*.

Juliette understands data. She knows that a customer id should not be null, that a birth year in the 18th century has no place in a Social Security system, that it's occasionally ok to have duplicate values… She gets it as a data scientist or business analyst. She needs clean data even if she might not be always able to define what clean data is.

Her pet peeves? Finding data and making sure it is (half) decent. She does not trust those data people, so first thing she does when she opens a notebook: copy the data somewhere she can access it. Even if it is 2 TB, her cloud engineer told her storage was cheap.

You want an angry Juliette? Tell her you need AI.

The thing that frustrates Beth the most is how organizations are doing data governance. This former data steward saw that she was going nowhere and rose up to adopt product thinking, data service-levels, lifecycle management.

She understands that documentation matters, what service levels are bringing to data, and writes data quality rules.

What annoys her is the lack of willingness to embrace change. She helps design new tools to manage data products at scale because she is passionate about the topic and desperate about current offerings.

You want Beth to stop talking to you forever? Ask her how many dashboards are in her data products (hint: zero).

Table 4-1 summarizes the primary personas and adds the focus on data products.

Table 4-1. Primary personas

	Cindy	Juliette	Beth
Role	Data Engineer	Data Scientist / Business Analyst	Data Product Owner (ex–Data Steward)
Superpower	Master of ETL/ELT pipelines, Python, dbt, Talend	Intuitive sense for data quality, analytical sharpness	Product mindset, SLAs, lifecycle governance, documentation
Pet Peeves	Firefighting, fragmented quality tools, messy upstream/downstream	Finding trustworthy data, hoarding 2 TB due to trust issues	Governance theater, bad tooling, status quo defenders
Trigger Warning	"Can you give me clean data?"	"We need AI."	"Where are the dashboards in this data product?"
Role in Data Product Lifecycle	Builder – implements pipelines and contracts	Consumer – uses data for analysis, ML, and AI	Owner – defines scope, quality, SLAs, and value, manages lifecycle
Impact on Data Product	Shapes structure, performance, and data quality	Surfaces quality issues, drives usability expectations	Orchestrates collaboration, ensures lifecycle and governance adherence

Secondary Personas

We can distinguish between two types of secondary personas: sub-personas from the primary ones and occasional stakeholders.

Juliette, our data consumer, can be a data scientist, a business analyst, or even a BI engineer. Whatever cap she's wearing, her needs with regard to the data product are the same: what differs is how she is going to consume the data. The data scientist will probably use notebooks and as such would like a Python API to interact with the data product. The BI engineer using Cognos or Tableau to design dashboards would like to see the catalog in her user interface. As I will study the consumption in more detail in Chapter 6), I will detail the differences there.

- The **Domain Data Owner** is ultimately accountable for the data in their domain. You may hear them rambling about "Is this going to get us sued?" and they probably forgot the term *data* in *data contract*.

- The **System Administrator** (aka Platform Engineer in 2015 and aka Infra SRE in 2025) builds and maintains the self-serve platform and its infrastructure, and dreams of terraforming everything.
- The **Enterprise Architect e**nsures architectural consistency across domains. Their focus is standards, interoperability, and future-proofing. They usually like the idea of data products as long as you align with the north star.
- The **Data Governance Lead** (or team, depending on the size of the company) defines and enforces policies (privacy, retention, and more). You may hear them asking, do we we have a policy for that?
- The **Data Quality Engineer** is the person in charge of building tests, monitoring data SLAs, and troubleshooting bad data (with Cindy, the data engineers). Although this role is crucial, for the sake of simplicity, I will consider that data quality engineering is a skill of the data engineer.
- The role of the **Executive Sponsor** (aka CDO, CIO, (S)VP Data (and AI)) is crucial. Although their title may differ, they clearly are part of the leadership team. They own the money and they are responsible for it. Don't underestimate the second part of this statement. In your role, you will have to help justify the value that data products are bringing so they can… give you more money. As data products often involve a culture change, make sure the Executive Sponsor is on board, that the vision is aligned, and that the vocabulary is not too technical. You will need their support when you hit roadblocks. You will need their help to set the right priorities.

Racing to Success

I strongly believe in a lightweight organization that empowers people. This does not mean anarchy, and it does not mean the French[1] government either, but somewhere in the middle. That's why I suggest starting any project with a RACI—a responsibility matrix that clarifies who is Responsible, Accountable, Consulted, and Informed for each task or decision. In many ways, a RACI is the adult version of "who's doing what" in group projects. Each stakeholder is either a R, A, C, or I, as in:

- *Responsible*: The person who actually does the work.
- *Accountable*: The one who has the final say and owns the result.
- *Consulted*: The folks you ask for input before acting.
- *Informed*: The people who just need the FYI after it's done.

An optimal RACI has one and only "A" per row, as you can see in Table 4-2

1 Although it sounds negative, they have their reasons and so do have many other governments.

Table 4-2. A RACI matrix that aligns with a practical implementation of and product-oriented data thinking

Activity / Role	Data Product Owner (Beth)	Data Engineer (Cindy)	Data Scientist (Juliette)	Domain Data Owner	System Admin	Governance Lead	Enterprise Architect	Exec Sponsor
Define product scope & ownership	A	C	C	R	I	C	C	I
Design data contracts	A	R	C	C	C	C	C	I
Implement pipeline & infrastructure	C	A	I	I	R	I	C	I
Ensure data quality rules	R	C	C	C	C	A	I	I
Test & validate data	C	A	R	I	I	C	I	I
Document & publish metadata	A	R	I	R	C	C	I	I
Monitor SLAs / SLOs	A	R	I	C	R	C	I	I
Approve product release / versioning	A	C	I	R	I	C	C	I
Ensure compliance & policy alignment	C	I	I	C	I	A	C	I
Provide strategic direction / funding	I	I	I	I	I	I	C	A

Obviously, this table is an illustration, and you should adapt it to your needs. In most organizations I've worked with, very often the "A" (accountable) is also the "R" (responsible).

Designing Around Four Workstreams

Let's see how you are going to split the work and understand the different workstreams.

If the experience planes are the blueprint, the workstreams are the construction crews. Each one tackles a distinct part of the house: infrastructure is the foundation, runtime builds the rooms, platform connects the utilities, and consumer experience makes sure Juliette (our ever-demanding consumer) gets a comfy couch and a good view. These four workstreams give you a pragmatic way to organize responsibilities and investment so your data products and their supporting services feel designed, not improvised.

Starting with Experience Planes

In Chapter 7 of *Implementing Data Mesh* (O'Reilly, 2024), Eric Broda and I describe the three experience planes to keep teams sane (and coffee budgets under control). Sections of that chapter are so essential that I adapted them for this chapter. Experience planes are split into three stacked planes:

Infrastructure experience plane
: The foundation. Think databases, data marts, compute, storage, and all the "boring but essential" stuff. Managed by sysadmins, DBAs, and SREs, it's where the raw machinery lives.

Data product experience plane
: The home of independent, isolated data products. Each product builds on the infrastructure but stays self-contained, communicating through data contracts. This ensures evolvability without messy interference. Data engineers and data product engineers live here.

Mesh experience plane
: The synergy layer. This is where data products discover and connect with each other via contracts, and where enterprise-level services, such as catalogs, marketplaces, monitoring applications, and security controls, reside. It's the place where individual products become a real ecosystem.

Together, these planes reduce cognitive overload, clarify responsibilities, and create a scalable structure: infrastructure as the bedrock, data products as modular units, and the mesh as the connective tissue that turns isolated parts into a thriving data network.

Figure 4-1 shows the three planes stacked together. In the next section, we'll explore each plane individually.

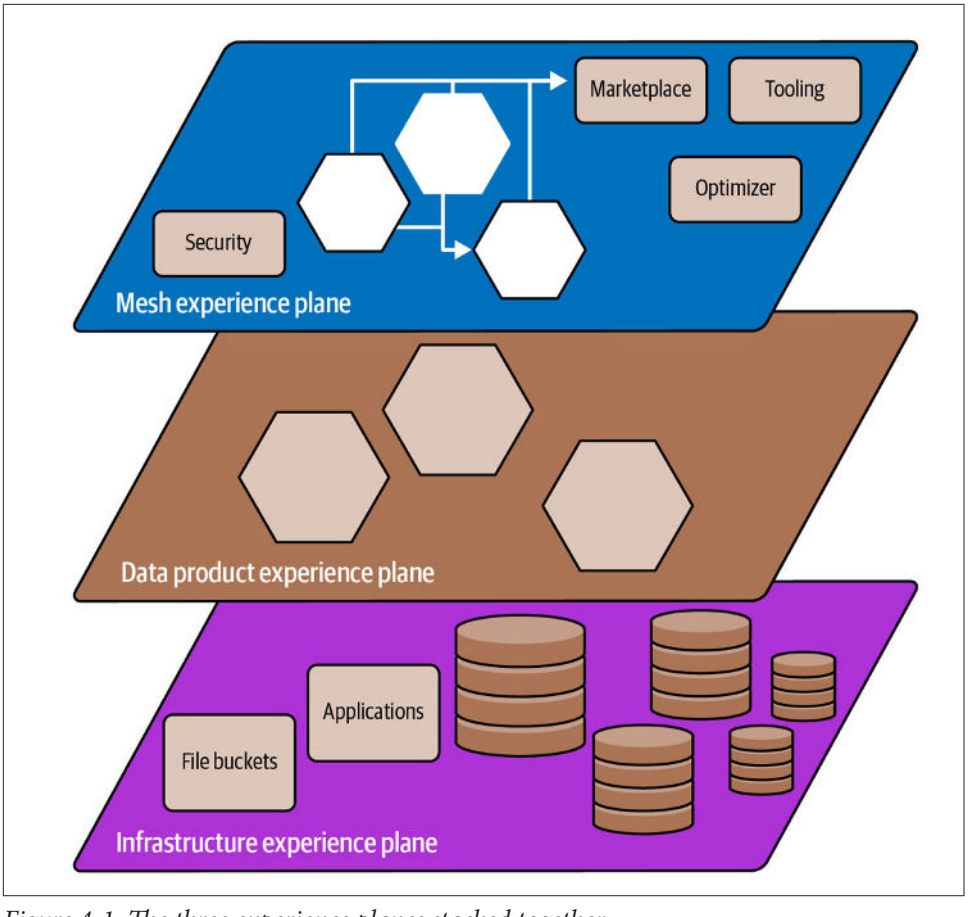

Figure 4-1. The three experience planes stacked together

The infrastructure plane remains key

The infrastructure experience plane contains all the elements you would consider to be your data infrastructure (pre–data products). As a data person, you are naturally aware of these elements: databases, data marts, and other applications that produce data. This plane also contains lower-level elements like compute, storage, orchestration, networking, operating systems, and more. Figure 4-2 represents this plane.

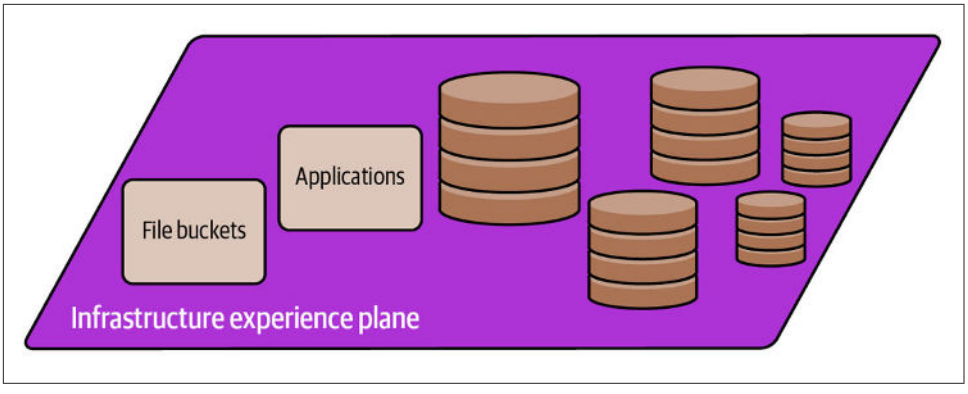

Figure 4-2. The infrastructure experience plane contains your data stores

Users of the infrastructure experience plane are your traditional system administrators (or system reliability engineers), database administrators, security engineers, and so on. You will build your data product and data mesh capabilities in other planes on top of this plane.

The data product experience plane is independent

The data product experience plane is the home of your unlinked data products. Let's have a look at how this plane interacts with the infrastructure plane. You build and deploy your data products independently of one another. They live on the data product experience plane, as illustrated in Figure 4-3.

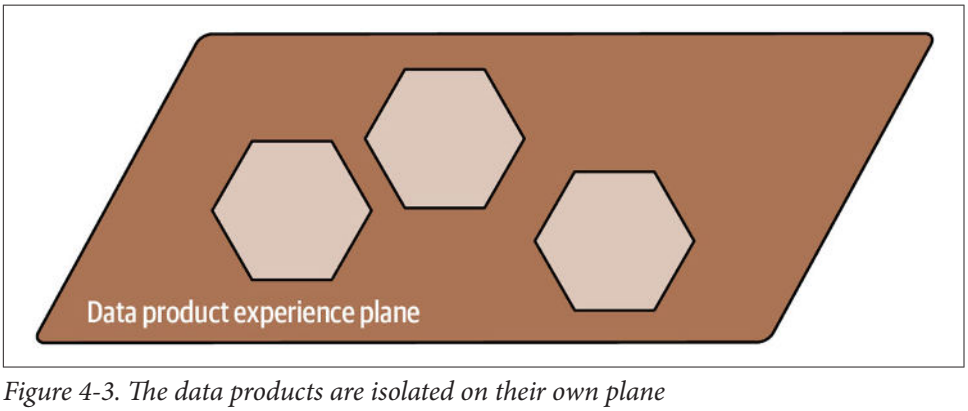

Figure 4-3. The data products are isolated on their own plane

A data product can feed itself from any source of data. This data product uses three data sources, illustrated in Figure 4-4: two databases and one application. The link between the planes is guaranteed by data contracts.

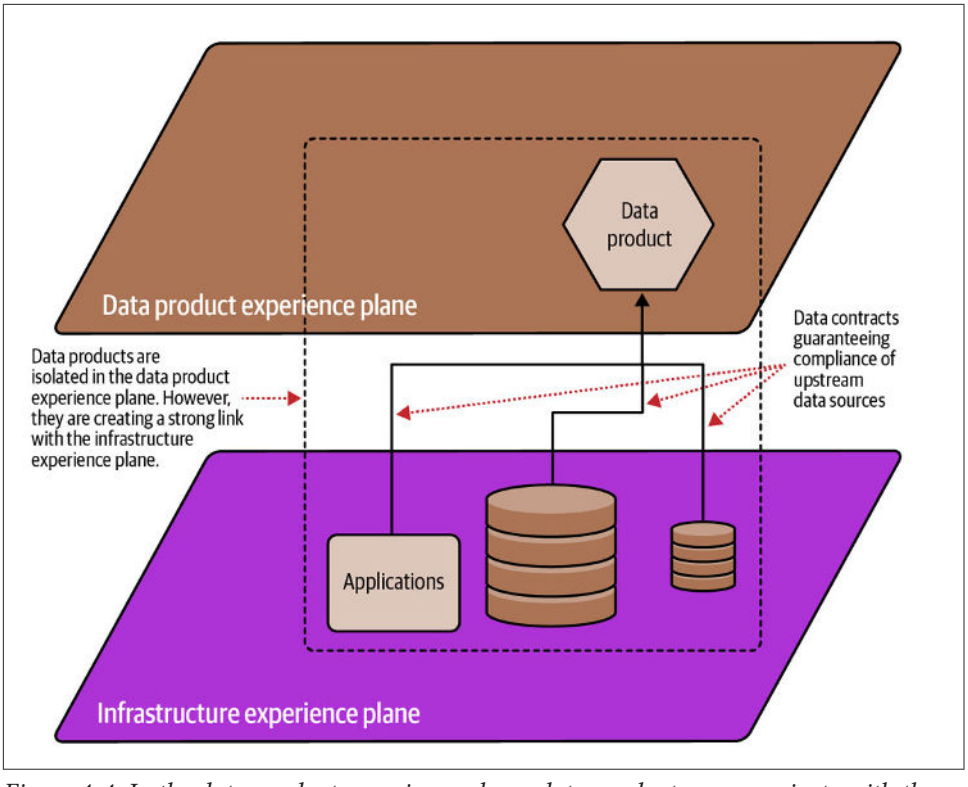

Figure 4-4. In the data product experience plane, data products communicate with the infrastructure plane via data contracts

The typical users of this plane are the data product engineers and data engineers.

The mesh experience plane is about synergy

As you can imagine, the mesh experience plane is where you are going to connect all the components together. Let's dive in. Figure 4-5 shows the enterprise-level components that the mesh experience plane will offer. Your enterprise data catalog (or data marketplace) sits here. You will also find a variety of tooling, monitoring, security management, and more. If you were building an instance of data mesh, this is where the data products would materialize their connections (or mesh).

Let's analyze two data products. Data product B uses data from data product A. Both report their information to a marketplace. Figure 4-6 focuses on data product A. The way that B can leverage information from A is through the data contract shared by A. A shares the same data contract with the marketplace.

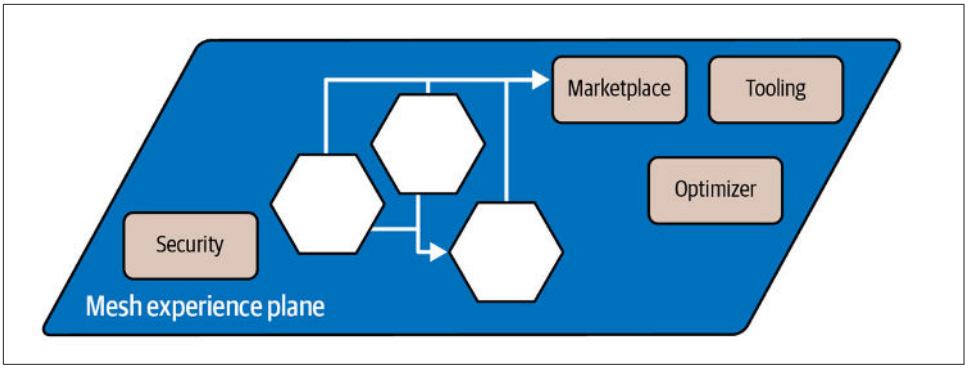

Figure 4-5. In the mesh experience plane, data products talk to one another as well as with mesh-level tooling

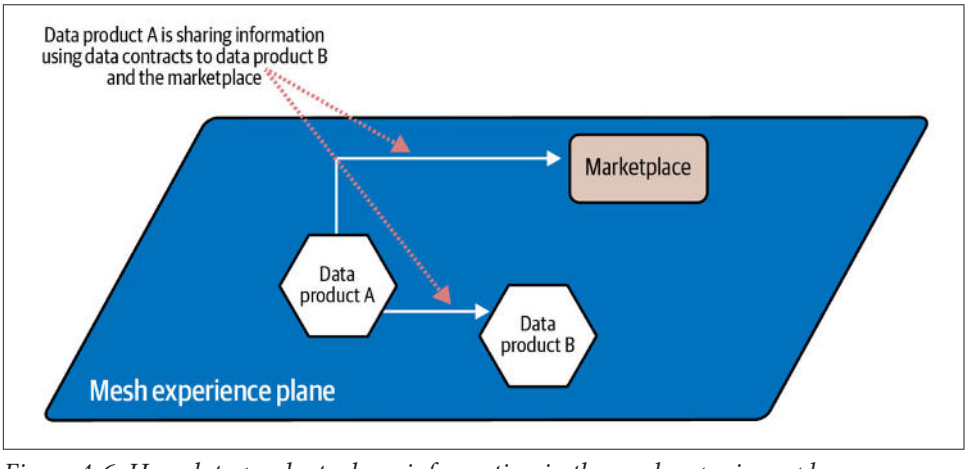

Figure 4-6. How data products share information in the mesh experience plane

For example, data product A could contain temperatures of all cities in France, whereas data product B contains only the temperatures of metropolitan areas (with a population between 500,000 and 1.5 million). Data product A exposes, through a data contract, its metadata (including its SLOs) to the marketplace where a user can find it. The same contract is shared with data product B so that it knows, for example, when the data in data product A is updated, the units that are used, and so on.

Data contracts can be transmitted in a few ways. The most common way is in a source control manager like GitHub, but it can also be in a Kafka topic, accessible via a set of REST services, or through other technology.

Finally, Figure 4-7 explains the link between the tools in the mesh experience plane and the data product itself. Those links are using the following:

- OOCS (Open Orchestration and Control Standard) allows each data product to send commands and information to the tool. Such tools could be measuring observability, running data quality rules, defining policies, and more.
- OORS (Open Observability Results Standard) returns the results of the execution of those tools back to the data product. Alternatively, OpenTelemetry (OTel) is a great protocol to bring similar data back.

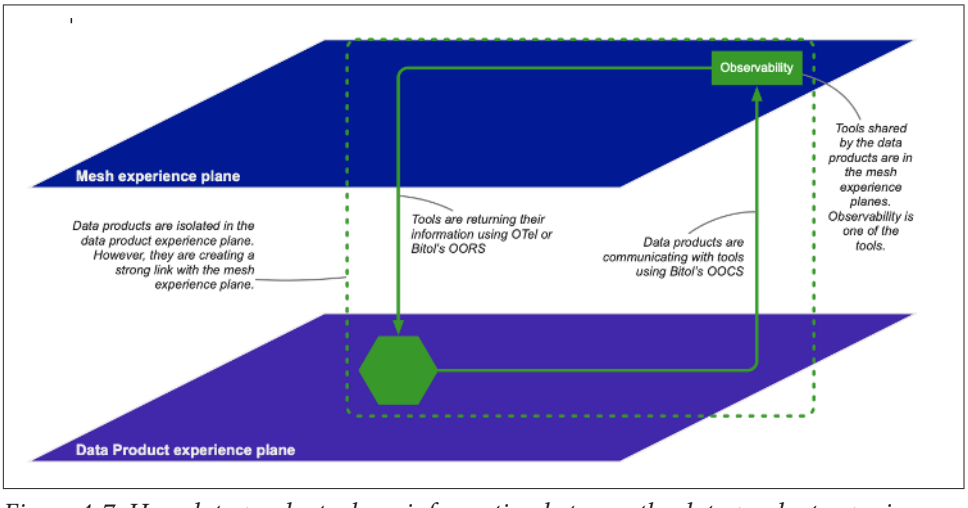

Figure 4-7. How data products share information between the data product experience plane and the mesh experience plane.

Based on the results of those commands, the data products can make decisions such as informing its users, shutting down, restarting pipelines…

The experience planes deliver a comprehensible structure and organization, similar to an architecture. They can easily be mapped to the people in your company. We can see Beth (your data product owner) managing products in the data product experience plane. In a similar way, Cindy (your data engineer) builds the pipelines dedicated to the data products, establish contracts, and defines IaC (Infrastructure as Code). She mainly works with elements in the infrastructure experience plane and link it to the data product experience plane. In our key personas, everyone has a role except Juliette, who uses the data.

Delivering a Fantastic Consumer Experience

The data consumer experience is often the poor cousin, treated as an afterthought. It should not be this way. Data consumers, Juliette in my personas, are the reasons we build this thing. They need faster data discovery, clear indicators on whether the quality is trustworthy or not, and guarantees that the data will not disappear at the

next refresh. Juliette needs dedicated tools and integration in the tools she uses. This requires a dedicated workstream.

Following Four Workstreams

The four workstreams can be defined around the three experience planes and the consumer experience, as summarized in Table 4-3.

Table 4-3. Workstreams aligned to experiences

Workstream	Plane / Area	Description
Platform	Data Mesh Experience	All the supporting tools and frameworks.
Runtime	Data Product Experience	The data product as an independent entity.
Infrastructure	Infrastructure Experience	The lower-level tooling and drivers.
CX	Consumer Experience	Dedicated tooling and integration for the data consumer.

The logical next step is to identify what you will need to build in those workstreams: let's establish a capability map.

Establishing Your Capability Map

Now that you have defined your four workstreams, let's add the capabilities we need. For the sake of simplicity, I define a *capability* as a set of features. As an example, *Marketplace* is a capability, while *faceted filtering* is a feature. Let's go through the four workstreams: Platform, Runtime, Infrastructure, and CX.

Platform Workstream

The platform provides services shared by all the data products. Here are eight of them:

Marketplace
: Data discovery is a pain. The data marketplace is essential: it is the entrance door. A marketplace can expose the data product's lineage, represent the data product as a knowledge graph, allow search on passive and active metadata, and more.

Feedback loops
: There are two kinds of feedback loops: the user feedback loop and the system feedback loop. The *user* feedback loop allows any type of user to report incidents or suggest improvements. The *system* feedback loop captures system events such as the result of the execution of data quality rules, SLA misses, and other observability elements. Some systems rate the user feedback on five stars (like Amazon) while they rate data quality on a scale of 1-100 (like a test score in a US school). Think about your users, and be consistent.

Data product (DP) lifecycle management
: Each data product is responsible for its own lifecycle; however, the management of the lifecycle is done at the platform level. It consists of basic activation of data products within the eco system as well as advertising new versions, optimizing duplicates, and more.

Policy management
: Federated governance is the way of doing governance in larger organization. Governance includes enterprise-level policies as well as line-of-business (business units) and country-specific policies. Their management takes place at the platform level since you will need consistency across your data products.

Governance
: The platform's governance includes reporting, monitoring, and auditing SLAs, data quality, and usage.

User notifications
: Users may use more than one data product, so facilitating communication with them is performed at the platform level. Notifications can be sent via email, Microsoft Teams, Slack, or any way you want (including homing pigeons).

Monitoring
: Although each data product knows its state, you are not going to ask them individually about their state. Monitoring is brought up to the platform.

Security
: Security is a growing concern and should be treated at the platform level.

Runtime Workstream

The runtime workstream ensures data products don't just exist. It's about making them run, behave, and stay accountable in production:

Sidecar
: The sidecar is the data product enabler. It will handle the communication with the external world. The sidecar handles the control, observability, and discovery services. The sidecar also manages local orchestration of the data product components (like scheduling data quality checks).

Control, observability, and discovery services
: Those services are essential for the data product to behave. You can fold them in the sidecar capability if it's easier for you to manage.

Support for SBOM
: The SBOMs define the transformation within the data product. As ODPS does not specify a specific SBOM, support for the one you pick is required.

Audit and security
: Logging is an important consideration to keep in mind. What happens when someone connects to a data product? User management is also a responsibility of the data product.

Infrastructure Workstream

The infrastructure workstream provides the plumbing (pipelines, storage, and IaC) that keeps data products flowing and grounded:

Data pipeline framework
: Interface with the tools that loads and transforms the data from the sources to the interoperable model.

Data storage
: Interface with the data storage: the data product needs to know where the data and share it with its users.

Infrastructure as Code (IaC)
: As part of its deployment, the data product needs to be able to interact with its infrastructure, and nothing says "as code" as a data product.

CX Workstream

The consumer experience (CX) workstream can be as easy as you want or as complex as your organization is: it's about integrating with the tools your data consumer use:

REST API
: The data product is a communication machine: it gets data, metadata, requests and returns information. An important part of the communication is the payload, which Bitol is standardizing.

Python SDK
: Making the REST APIs palatable to data scientists (Juliette) and data engineers (Cindy) goes through building a solid Python SDK.

BI tools
: Business users (less technical Juliettes) will use reporting and BI tools with your data product. It makes sense to bring required information in the tool, such as the catalog or (sample) data.

Building the First Data Product

Do you need to build all the capabilities before you start deploying (and seeing value from) your data products? No! We are not in waterfall anymore. This section explores the minimum capabilities you need and adds some data.

Starting with Basic Capabilities

Let's identify a minimal viable set of capabilities and their features that your data products should have. For this first data product, I want to emphasize the ease of discovery. My user story could go like this:

> *As a software engineer, I want to consume the ODPS and ODCS descriptors stored in the data product, so I can expose them to my application.*

The implementation will share the ODPS file(s) on an endpoint called `/products`. I would also share its contracts, and I know there could be many, so I'll have a `/contracts` endpoint as well. Both endpoints are part of the discovery category.

Although a data product has only one active ODPS file at a given time, the endpoint's name can safely be called `/products`.

In Figure 4-8, you see the data product with its management ports and internal components.

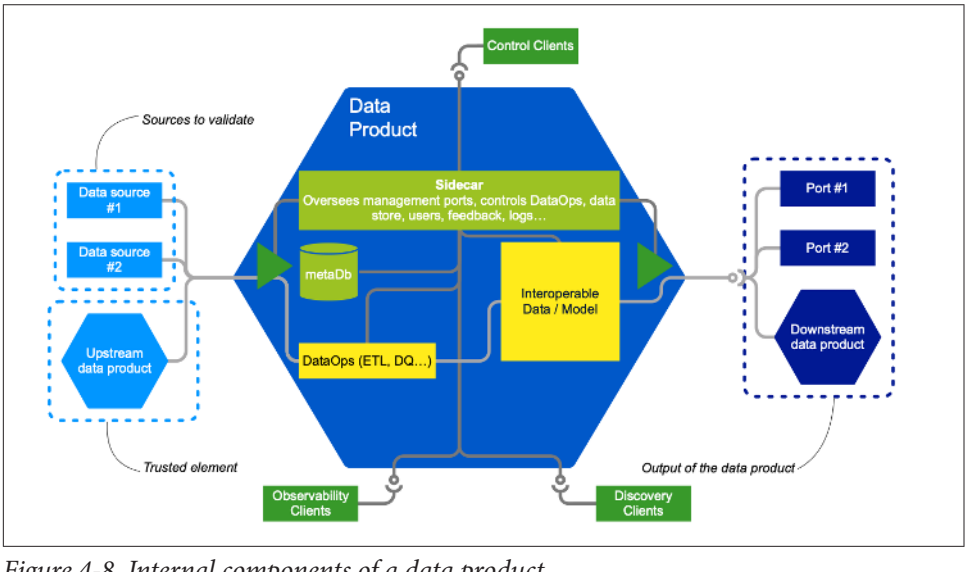

Figure 4-8. Internal components of a data product.

The work is divided in two major workstreams:

- The runtime, composed of the sidecar and metaDb.
- The data in the data product.

Data Product Runtime: Sidecar and Internal Database

Think of the data product runtime as the "engine room." It's where the machinery keeps humming, where APIs open doors to the outside world, and where stateful details are tucked neatly away in a small, embedded database. The sidecar acts like the product's brain and concierge: managing discovery, observability, and control, while the internal database keeps track of what's going on behind the scenes. Together, they make sure your autonomous data product doesn't just exist but actually runs.

Sidecar: The brain of your data product

The sidecar controls the operation of your autonomous data product. It orchestrates everything, knows about everything, and communicates with the external world at the metadata level.

For your first iteration, you will prioritize your development on the elements bringing the value your organization expects. In my scenario, I will design and implement the following:

- *Discovery:* Retrieving the contracts and product definitions
- *Observability:* Retrieving the last five user activities
- *Control:* Starting and stopping the pipeline

The categories (discovery, observability, and control) are used for sorting the operations logically. The APIs are working on resources.

Your first API endpoints are described in Table 4-4, alongside the Bitol-standardized payload.

Table 4-4. First resources to implement in your data product

Resources	End point	Description	Payload	Category
Products	products	Inquires about the product. Ideally can get prior versions for auditing purposes.	ODPS	Discovery
Contracts	contracts	Inquires about the contracts.	ODCS	Discovery
User activities	user-activities	Inquires about interactions/activities between users and the product. It can contain user subscription, access, and more.	OORS	Observability
Pipelines	pipelines	Controls the pipeline's activities and inquires about its status.	OOCS	Control

More endpoints are described in Appendix A.

metaDb: the data product state

The data product needs to store some of its information to maintain its state on provide information through its services.

Some of the most obvious information the data product stores are observability and data quality metrics, its users, its logs, and more. I would recommend a small, embedded database like H2, Derby, or Zen that do not require maintenance. Asides from logs, a relational database is a good match here.

The Data in Data Product

When it comes to data products, you don't always start with a blank canvas. Sometimes you're handed a Picasso-wannabee that looks more like finger paint. That's the essence of brownfield: taking existing data assets and shaping them into proper data products. On the flip side, greenfield is the dream scenario: start fresh, build shiny, and hope no one reminds you that constraints will crash your party sooner or later. Let's dive into each approach.

Brownfield: Transforming existing data assets into a data product

A common misconception I see very often is that data products are only available for new data projects (rare and it already starts badly: it's not a project) or migration projects (from frequently and still bad name) from existing data assets to new data assets that will be deemed as data products.

This thinking is wrong on so many levels. Why not transform your existing data assets into data products? One common objection I get is "I don't want my 4 million tables to become 4 million data products." That is a valid point, but:

- You do not have to migrate all your tables at the same time.
- Focus on your use cases—they probably have fewer than ten tables.
- Migrate use case by use case.

Think of it this way: someone already worked on designing/modeling the tables for those use cases. They may not be perfect, but they work. Do you need to redo this work?

This is also where you need to adopt more of a product mindset. Figure 4-9 illustrates two cycles of enhancement, producing data products with version vNext. Here is how it works:

1. The existing requirements, requests for improvement, and bugs are turned into a backlog.
2. This backlog is used for prioritization of vNext, based on the existing data asset.

3. Enter the development phase, which produces the data product.
4. During the development and production phase, new requirements, new bugs, postponed features, and tech debt are feeding the backlog.
5. Ready for the next iteration.

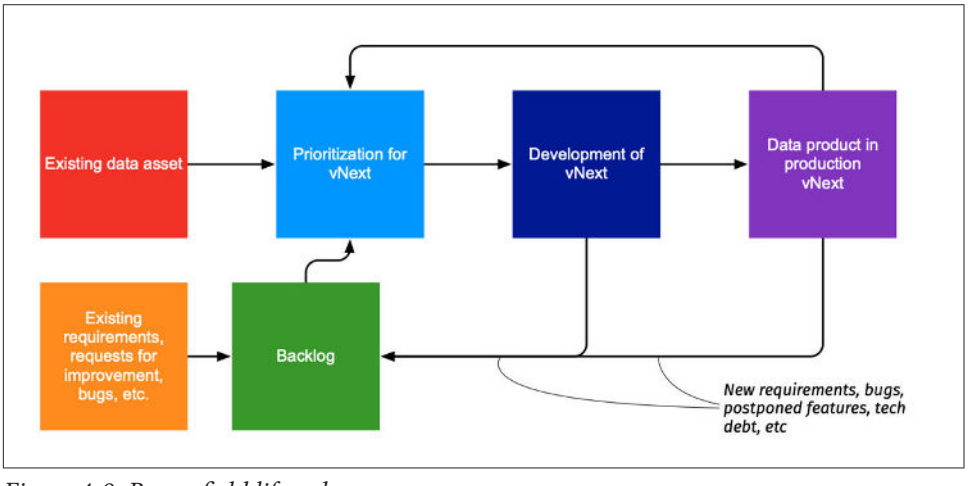

Figure 4-9. Brownfield lifecycle.

Note that your first iteration should be extremely fast, so you can deliver your existing asset as a data product as quickly as possible.

Greenfield: Building a data product from scratch

When you don't have any prior constraint, you can start building in a greenfield way. It's easier, or so you think, because you can enjoy more creative freedom, but let's be honest: an enterprise IT project without any constraint does not exist.

Figure 4-10 illustrates a greenfield approach to building a data product. It is oddly similar to a brownfield approach:

1. The existing requirements, requests for improvement, and bugs are turned into a backlog.
2. This backlog is used for prioritization of vNext.
3. Enter the development phase, which produces the data product. The development phase includes:
   a. *Modeling:* You will have to design the interoperable model.
   b. *Implementation:* Design the pipeline, define data quality rules, and set service levels.

4. During the development and production phase, new requirements, new bugs, postponed features, and tech debt are feeding the backlog.
5. Ready for the next iteration.

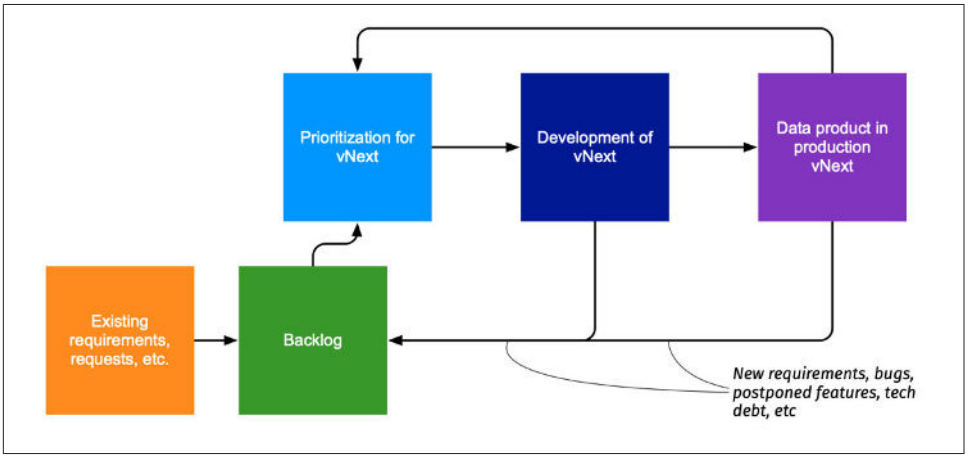

Figure 4-10. Greenfield lifecycle.

You may get a sense of ease in the greenfield approach, but you will have a longer development cycle.

Comparing brownfield and greenfield

Table 4-5 compares both brownfield and greenfield approaches.

Table 4-5. A summary of the green field vs. brown field approaches

Aspect	Greenfield	Brownfield
Definition	Building data products from scratch, no legacy constraints.	Creating data products on top of existing data assets, systems, infrastructure, and processes.
Freedom	Maximum creative freedom: pick stack, architecture, and standards.	Constrained by existing choices, but with working foundations to leverage.
Complexity	Low initial complexity, but you must design and implement all capabilities, delivering a MVP to validate your approach.	Some integration complexity, but you can incrementally enhance observability, quality, and security.
Speed	Fast to start, slower as you build governance interfaces and reliability layers.	Slower at first (integration, alignment), but accelerates by reusing what works.
Risk	Risk of reinventing the wheel or over-engineering.	Risk of technical inertia, but less "unknown" than starting from scratch.
Cost	High upfront investment for modeling, infrastructure, and processes.	Lower startup cost by reusing existing assets, with costs spread over improvements.
Examples	A startup launching a set of data products with ODCS/ODPS from day one.	An enterprise layering data product standards over its mature but imperfect CRM, ERP, and warehouses.

Summary

In this chapter, you learned the following:

- Personas first: Cindy (data engineer), Juliette (data consumer), and Beth (data product owner) frame the main roles and needs.
- Secondary personas matter too: Domain data owners, sysadmins, architects, governance leads, and exec sponsors all shape success.
- RACI clarifies responsibilities: one "A" per row keeps accountability clean.
- Experience planes = blueprint: Infrastructure (foundation), data product (independent units), and mesh (connective tissue).
- Four workstreams = construction crews: Platform, runtime, infrastructure, and consumer experience (CX).
- Capabilities vs. features: Think big buckets (capabilities) broken down into concrete functions (features).
- Runtime essentials: Sidecar + metaDb keep the data product alive, handling discovery, observability, and control.
- APIs are the product's voice: /products, /contracts, /user-activities, and /pipelines are your MVP endpoints.
- Brownfield vs. greenfield: You can wrap existing assets into data products (brownfield) or start from scratch (greenfield)—both work, trade-offs differ.
- Standards keep you sane: ODCS + ODPS describe what a data product is, while OOCS + OORS (and OTel) define how it behaves.

CHAPTER 5
Architecting Data Products

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at *ccollins@oreilly.com*.

Data products do not magically become "products" because we put the word in a slide deck. They become products because they are designed, architected, and operated as such. That means making deliberate architectural choices: where logic lives, how responsibility is split, how trust is enforced, and how the system behaves when (not if) something goes wrong.

This chapter is about those choices. Not vendor diagrams, not reference architectures carved in marble, and definitely not "just add governance." Instead, you will look at practical architectural patterns that make data products active: self-aware, observable, enforceable, and evolvable. Architecture here is not about boxes and arrows for their own sake; it is about enabling conversation between systems, teams, and eventually machines.

To keep ourselves honest (and sane), we will rely on a lightweight architectural lens, progressively zooming in only when it adds value. The goal is simple: help you design data products that can survive change, scale without chaos, and earn trust repeatedly, without turning your architecture diagrams into modern art. I will loosely align with

the first three levels of the C4 model for architecture and *The C4 Model* by Simon Brown (O'Reilly, 2026).

Now that expectations are set, let's talk architecture.

The C4 model is a pragmatic way to describe software architecture without turning every diagram into abstract art or a PhD thesis. Faced with the rich buffet of methodologies out there (yes, I had to pick one), I picked C4 because it's simple, hierarchical, and actually usable by humans: start with the big picture (context), zoom into containers, then components, and only go low-level when it truly matters. Think of it as Google Maps for software architecture: country view first, city view later, then street view, and no one forces you to draw every brick unless you really want (or need) to.

The four views of the C4 Model are as follows:

Context
: The big picture including users, systems, and how your system fits into the world.

Containers
: The major building blocks such as applications, services, databases, and their responsibilities.

Components
: What's inside a container as the key components and how they interact.

Code
: The implementation details like classes, interfaces, and source code (used sparingly, for obvious sanity reasons).

Start wide, zoom in, but stop before everyone regrets it.

Why Are Data Products Active?

Let's look at it the other way. What would be a passive or static data product? It would provide data, some level of value, eventually a document describing the provided data, ideally as an ODCS data contract. However, you wouldn't be able to interact with it in a similar way as this:

> You: How are you, Ms. Customer Data Product?
>
> Ms. Customer Data Product: Oh! I am glad you asked. I have not been updated for a few days now; it must be the Salesforce pipeline again.
>
> You (feeling concerned): I am sorry to hear that. Do you have any additional clues?
>
> Ms. Customer Data Product (ecstatic, but a little embarrassed at the idea of sharing her internal affairs): Version 2 of the raw output port is still working, however its version 1 seems to not be working because of the associated input data contract. It seems to be due to an upstream schema change.

Of course, it would have been nice if Ms. Customer Data Product had told you that preemptively, but have you subscribed to her updates? If not, this one is on you. This dialog looks like the sequence diagram in Figure 5-1.

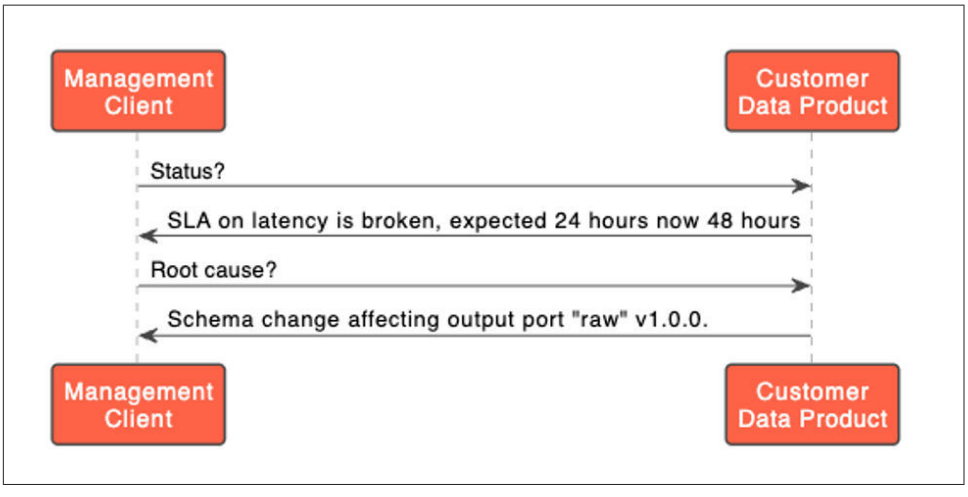

Figure 5-1. Discussion with a data product

Try asking a CSV how it's doing. It won't answer. Not because it's rude, but because it's blind, mute, and unaware of its own existence. An *active data product*, on the other hand, knows its state, its dependencies, and its problems. It should tell you how it's doing before your consuming dashboard or application explodes. That's not a luxury; that's how trust is built.

Table 5-1 summarizes the main differences between a static file (CSV, Parquet, etc.) and a data product.

Table 5-1. Comparison between a static data asset (like a CSV file) and a data product

Dimension	Static File	Data Product
Nature	Passive artifact, frozen in time	Living artefact, aware of its own state
Freshness	Unknown unless someone checks	Continuously tracked and exposed
Schema management	Implicit, tribal knowledge	Explicit, versioned, contract-driven
Change handling	Breaking changes discovered by consumers	Breaking changes detected and flagged early
Data contracts	Documented (maybe), not enforced	Executable and enforced at runtime
Observability	Pipeline-level logs, if any	Product-level metrics (SLA, quality, usage)
Failure detection	After consumers complain	Proactive, often before impact
Lineage	Reconstructed during incidents	Live, continuously maintained
Interaction model	Download → transform → hope	Query, subscribe, interact
Feedback loop	None	Built-in (usage, quality, incidents), by both users and systems

Dimension	Static File	Data Product
Ownership	Vague, informal	Explicit owner with accountability
Governance	Manual reviews, spreadsheets, PDFs	Automated, policy-driven, continuous
Reusability	Low, every team reinvents logic	High, standardized and discoverable
AI readiness	Context-free, risky input	Trusted, contextual, AI-friendly
Time to trust	Long and fragile	Short and repeatable
Failure mode	Silent corruption	Noisy, explainable, actionable
Organizational impact	Chaos scales linearly	Value scales exponentially

Architecting for the Active State

In Chapter 1, I mentioned that data products are logical constructs. A data product is always true, unlike Schrödinger's Cat. However, data products can also be implemented, and this is where they bring more value. It has an impact on the architecture.

A logical data product is simply an entry in a catalog (or marketplace). Its value resides in its documentation and, depending on the marketplace's features, its ability to collect and share active metadata (see Chapter 8 for a deep dive into metadata).

By being active, a data product enables dialogs (information exchange) with clients. Those clients could be consuming metadata for discovery or monitoring, controlling the data product, or consuming data via the output port. The data product can be a data consumer itself. Figure 5-2 reminds us of this architecture.

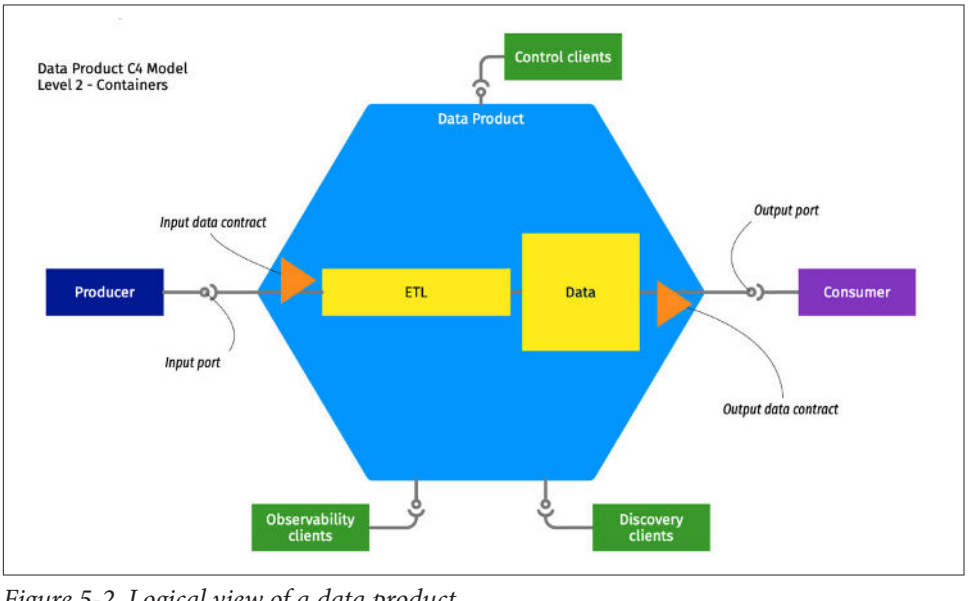

Figure 5-2. Logical view of a data product

When you get to the next level of architecture, there are at least three ways of implementing them:

Centralized
: The data product remains a logical construct in a catalog.

Fully autonomous
: The data product owns everything and is delivered as a container (Docker or other) that can sit next to the data, avoiding data movements.

Hybrid
: A little bit of both.

The following sections group the observability, discovery, and control clients under the generic *management clients* term since they interact in a similar way with the data product.

Centralized Implementation

A centralized implementation is the most popular implementation and most likely the easiest one (Figure 5-3).

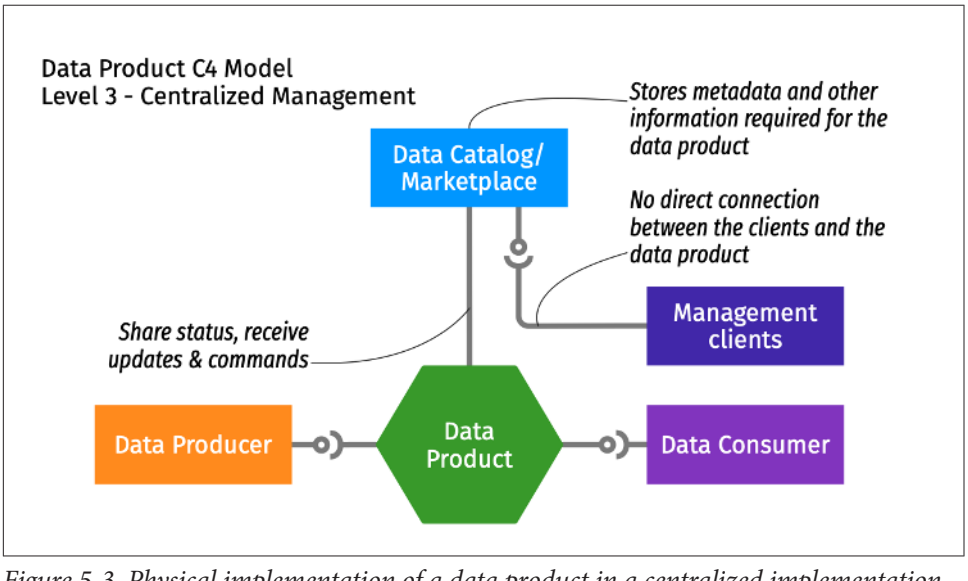

Figure 5-3. Physical implementation of a data product in a centralized implementation (C4 level 3)

Management clients do not directly contact the data products, but contact their "handler."

In a variation of this design, the management clients' request can have the impression of connecting to the data product, but the request is rerouted to the marketplace via a router (like nginx, for example). Figure 5-4 illustrates this mechanism.

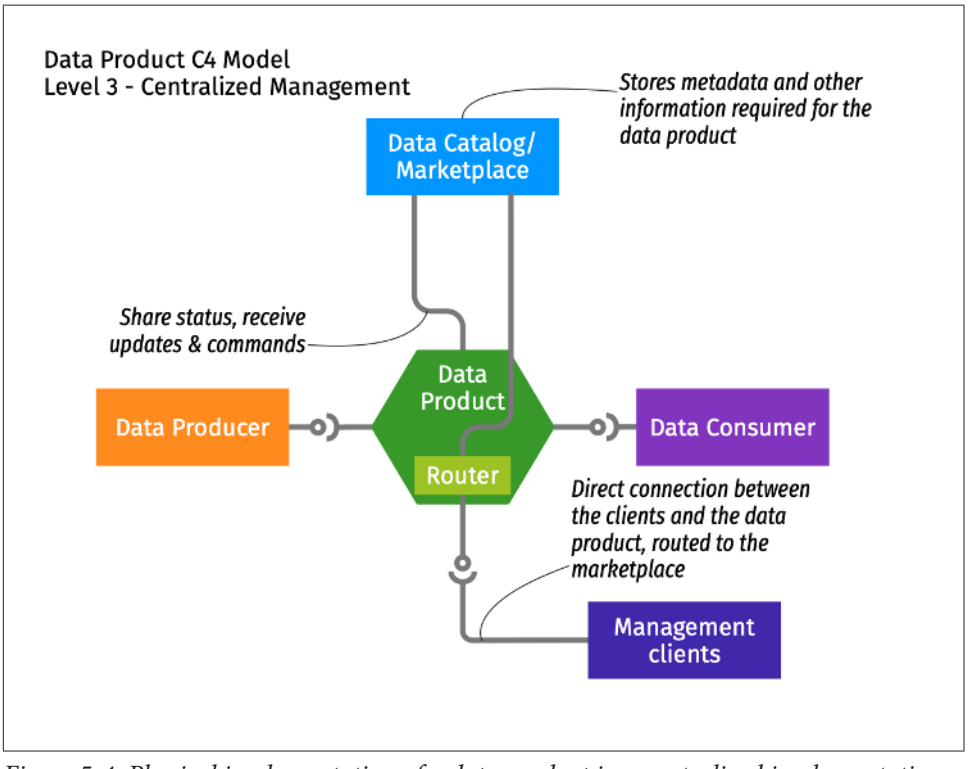

Figure 5-4. Physical implementation of a data product in a centralized implementation with direct connection to the data product (C4 level 3)

In this scenario, the marketplace can also be extended to the mesh experience plane (see the various experiences planes in Chapter 4).

Autonomous Data Product

The autonomous data product is a standalone approach to deploying data products. The autonomous data product can work without a marketplace or any central element reducing risks around single point of failure (SPOF).

Autonomous data products are most likely the natural evolution of data products: imagine Docker (or another) container carrying the necessary features and floating in your infrastructure, close to your data, disabling the need for heavy data movement (Figure 5-5).

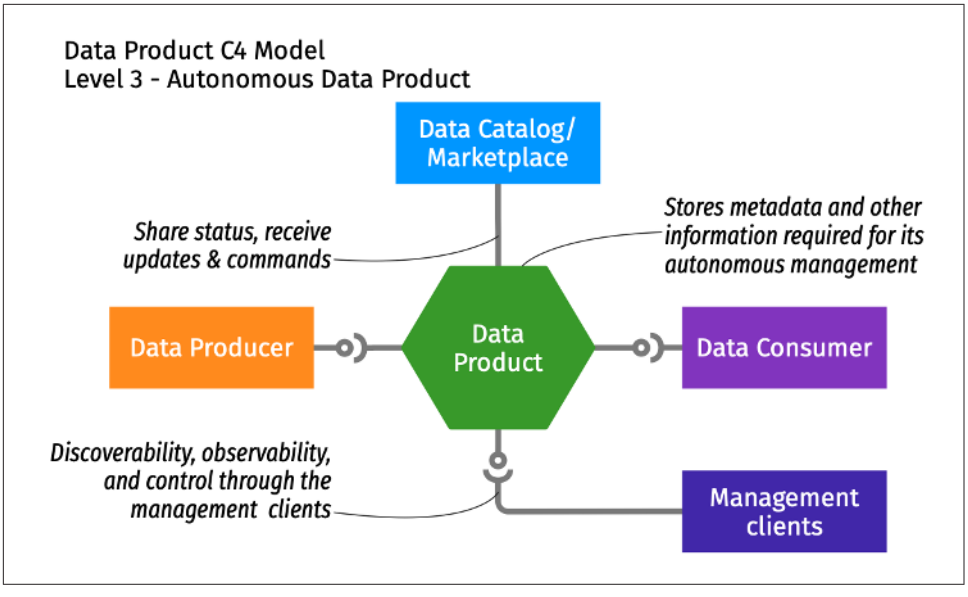

Figure 5-5. Physical implementation of an autonomous data product (C4 level 3)

In this situation, the management clients directly interact with the data product. The data product is in charge of saving its internal status, configuration, and active metadata in a local database. I often call this database *metaDb*. (In some of my previous articles and books, I used *QuantumDb,* as I was focusing on data *quanta* (singular data *quantum*) instead of data *products*. I will not use that terminology here.)

As the data product is completely autonomous, metaDb should keep track of the following:

Observability information
: The data product keeps track of the health of its internal components, like the onboarding ETL, input and output data contracts validation, internal storage, and so on.

Access information
: Who, when, how are users interacting with the data products.

Admin tracking
: Data products admin tasks like starting, stopping, changing a pipeline, and so forth should be logged as well.

Feedback loop
: The data product collects its own feedback from users. The system feedback loop is done via observability.

The database can take any shape you want: relational, log-based, blockchain, hybrid, or more; what matters is the API. Note that based on the complexity of the data product you are building, the implementation of metaDb and the associated code (see the section "Capitalizing on the Sidecar" later in this chapter) should not be affected.

Shared Resources

In some situation, you may be required to share resources to follow global architecture patterns or other policies. Figure 5-6 shows a simplified design of the architecture that my team implemented at PayPal. To avoid the proliferation of the metaDb database instances (one per data product), the central database team requested that the team use a single instance. It required new cycles of architecture and implementation that could have been prevented.

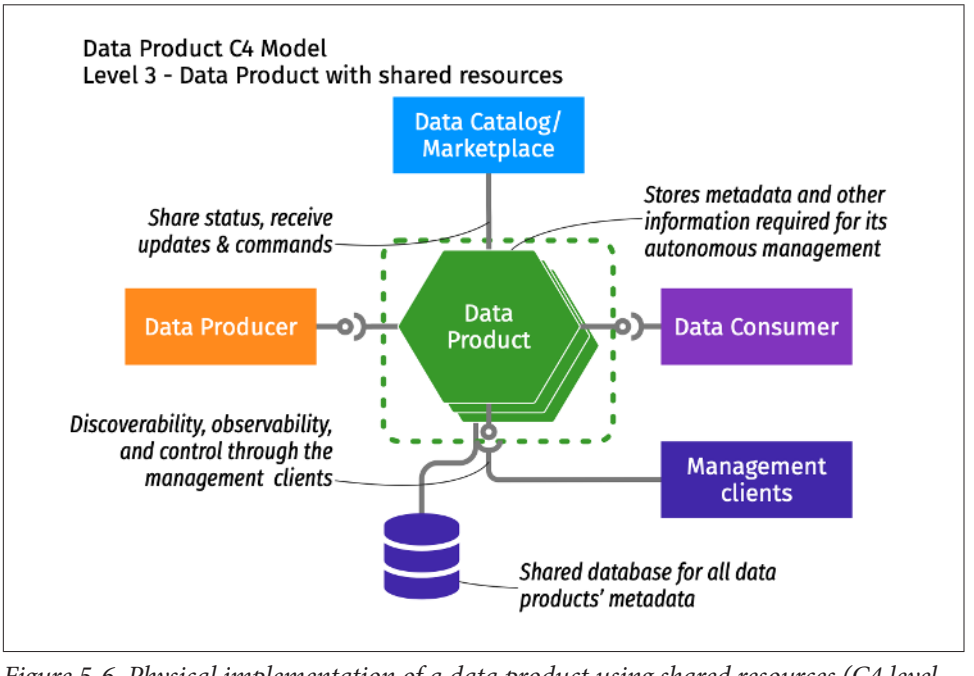

Figure 5-6. Physical implementation of a data product using shared resources (C4 level 3)

This situation is less than ideal but may be a requirement. I'll advise you to have a good discussion with your enterprise architects and the teams in charge of the services you plan to use.

Capitalizing on the Sidecar

I covered the internal metadata storage of the data product in metaDb. However, there is a layer of software that needs to be implemented as a sidecar.

Let's talk about what a sidecar brings and which services it adds to data products.

Comparing a Library and a Sidecar

In most software applications, you would use a *library* to standardize some of the features and behaviors that will be used in common between apps. Examples of libraries include Apache Commons Lang, which enhances Java's core functionalities, NumPy, numerical computing library for Python, and Jackson, a JSON parsing for Java. A library is not a service. It doesn't negotiate. It just executes, and takes you down when things go south.

On the other hand, within the Kubernetes ecosystem, Istio uses *sidecars* (Envoy) that enable networking function for each pod it is attached to. A sidecar is how platforms add behavior without touching your code.

Table 5-2 summarizes the difference between the two concepts.

Table 5-2. Comparison between a library and a sidecar

Dimension	Library	Sidecar
Where it runs	Inside the application process	Next to the application (separate process/container)
How it's used	Imported and linked at build time	Accessed over IPC, HTTP, or gRPC
Deployment	Rebuilt and redeployed with the app	Deployed and updated independently
Coupling	Tight coupling	Loose coupling
Language dependency	Yes (language-specific)	No (language-agnostic)
Upgrade impact	Requires app rebuild (or at least retesting with dynamic libraries)	Can be upgraded without touching the app
Failure isolation	None (shared fate)	Partial (process-level isolation)
Best suited for	Business logic, utilities, SDKs	Cross-cutting concerns (security, observability, policy)
Typical examples	JSON parser, Object-Relational Mapping (ORM), ML SDK	Service proxy, auth agent, metrics collector
Operational complexity	Low	Medium (you operate another thing)
Risk	Strong dependency with the applications, needs high level of regression testing	Looser coupling, could be used in a malicious way
Mental model	"Call a function"	"Delegate to a companion service"

What's in the Sidecar?

The preceding section discussed the importance of the sidecar. Let's dive into the feature and the components that are in the sidecar. Table 5-3 lists the services offered by the sidecar as well as why the service is needed.

Table 5-3. Features of a data product sidecar

Sidecar service	What it does	Why it does it
Contract enforcement	Validates data against the Open Data Contract Standard	Breaks early, not downstream
Version management	Manages schema and semantic versioning	Enables safe evolution
Policy and access control	Enforces who can read/write/use	Avoids a central gatekeeper
Quality checks	Ensures data quality	Ensures trust without babysitting
Observability	SLAs, metrics, lineage signals	Operates, not guess
Service level enforcer	Some SLAs, like retention period	Ensures compliance
Lifecycle automation	Publish, deprecate, retire	Enables product thinking, not dataset
Discovery and metadata sync	Pushes metadata to source control, catalog, marketplace, or mesh experience plane	Eases discovery as it is findable by design
Cost and usage tracking	Measures consumers and load	Enables chargeback & ROI
Eventing & alerts	Emits contract breaches & changes	Machines talk to machines (and sometimes to humans)
User feedback loop	Collects usage but primarily feedback	Enriches the knowledge of the data product owner
Management services	Supports the control, observability, and dictionary services	Allows the communication with the external world
Scheduler	Schedules the various tasks that need to be automated, like testing data quality, measuring SLAs, etc.	Enables scheduled automation
AI-readiness signals	Confidence, bias, suitability	Enables safe reuse by AI agents

When you map that to the architecture diagram, as illustrated by Figure 5-7, you see that the moving parts between different data products are the following:

- The transformation itself.
- The data exposed through the output ports.
- The data contracts linked to the output ports.
- The optional data contracts linked to the input ports (or input data contracts).

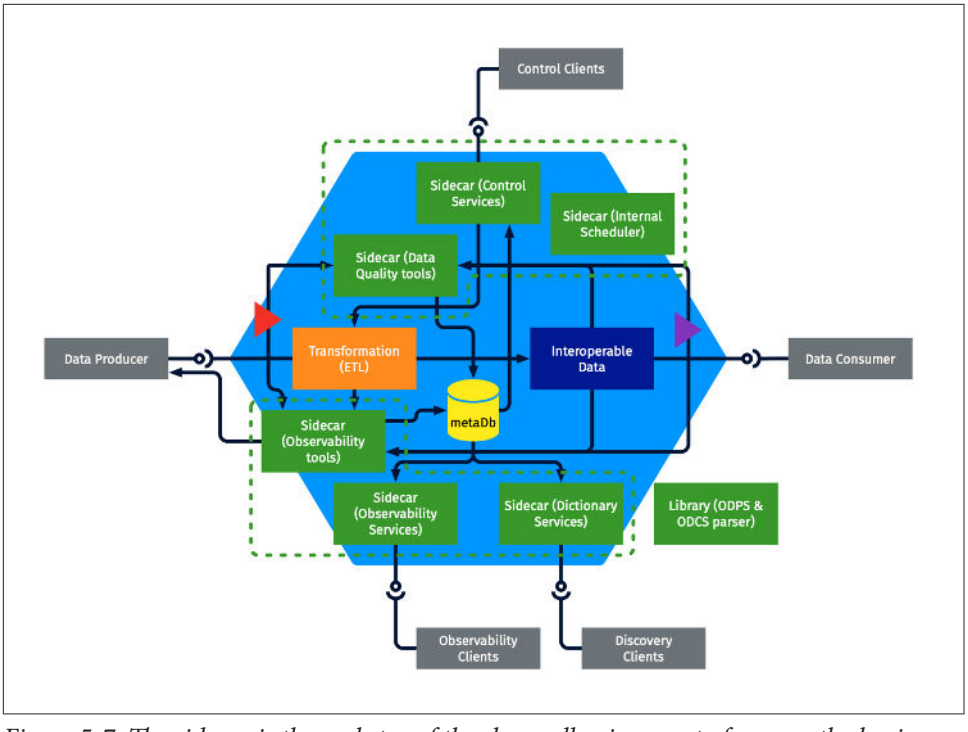

Figure 5-7. The sidecar is the real star of the show, allowing you to focus on the business transformation

Modern software deployment is no longer a ceremonial release night with pizza and crossed fingers (avoiding Fridays like the plague). CI/CD pipelines turn code changes into a boring, repeatable conveyor belt: commit, test, build, deploy, repeat. Every change is automatically validated, packaged, and pushed forward, which means small updates ship often and failures are caught early.

Containers are the secret sauce that makes this work at scale. By packaging an app with its runtime and dependencies, containers ensure that "it works on my machine" finally works everywhere else too. Tools like Docker and orchestrators such as Kubernetes let teams deploy the same artifact from laptop to production with confidence, roll out updates gradually, and roll back just as fast when something smells funny. The result is faster innovation, safer releases, and engineers who sleep through the night (or play video games, but that's out of the topic of this book).

Autonomous data products follow this philosophy, and the sidecar helps. Of course, the sidecar will evolve: you can imagine a first version using Spring Boot to support REST-based API and evolve to support both REST and message queuing via Kafka or RabbitMQ.

As the sidecar is part of your container deployment, the feature set and the support for more external tools and technologies will grow while keeping maintenance costs at a reasonable level.

Minimum Viable Sidecar

You know it: Rome, days, building time, and so on. Build something quickly so you can demonstrate the concept.

Defining a Sidecar MVP accelerates adoption by giving teams a clear, minimal, and non-negotiable starting point: contract enforcement, basic observability, metadata sync with a catalog (or other documentation platform), and a baseline of security. It reduces cognitive load, establishes trust from day one (no "we'll secure it later"), and keeps platform teams sane by avoiding sidecar sprawl. Most importantly, the minimal viable sidecar will let data products ship value early so you can gather feedback, show value, and solidify your project.

Security as a First-Class Citizen

Security is not a bolt-on, nor a Chapter-11 afterthought. It starts inside the data product, with the sidecar. The foundation is the data contract, which explicitly defines ownership, roles, and usage expectations. From that, Role-Based Access Control (RBAC) naturally emerges: who can read, write, operate, or administer the data product is not inferred from tribal knowledge, but is defined in the contract and enforced at runtime. The sidecar implements this contract-driven security by authenticating callers, authorizing actions, protecting secrets, and emitting audit logs.

Security should not be a showstopper for good actors. That's why the data contract specifies the role, its abilities, and the chain of command. Example 5-1 illustrates how you can access this product's output port: you need to have the `customer360_user` role to access in `read` only, then approval for your reporting manager, and finally by the Mandalorian, because this is the way.

Example 5-1. Contract abstract for accessing the specific output port of a data product

```
roles:
  - role: customer360_user
    access: read
    firstLevelApprovers: Reporting Manager
    secondLevelApprovers: mandalorian
```

In short:

- Support for RBAC directly at the data contract level.
- Logging of all activities at the product level.

- Optionally, logging of all SQL queries when using relational databases (or other queries).

Contracts define trust, the sidecar enforces it (Chapter 11 goes deep on the *how*). But keep in mind a rule of thumb: if security is not enforced by the sidecar, it's not security; it's hope.

Avoiding Distractions of Edge Cases

As you are building data products, there is one dangerous rabbit hole: focusing on the edge cases and losing track of the value. Before turning the architecture page, let's think about a few things that will help you and your users in adopting data products:

Outcome, not internals
: A typical example comes from my first implementation of a data mesh while at PayPal. The team wanted to better understand the behavior of the pipelines feeding the interoperable data. There were two approaches to it: either jump into the pipeline and make sure we had the circuit breaker code in place or focus on the outcome of the pipeline. Diving inside the pipeline would require understanding the code that may have been written a long time ago, which no one really understood, eventually finding tools to analyze or recompile code that was not even in source control. Focus on the outcome: on what the pipeline should deliver, not on its internals. Describe those outcomes in an ODCS data contract as data quality rules or SLAs.

Mainstream technologies first
: If you need to monitor some system activities as part of your data products, focus on technologies you understand before you jump on integrating Perl-based pipeline, make sure your PySpark-based pipeline are well integrated.

Consistent experience
: Be respectful of your end-users. They are not engineers, they are business people, almost "normal" people. Thanks to Amazon and Google, everybody expects to see a human feedback rated on a scale of 5 (like the 5 stars you will rate this book on Amazon and Goodreads). In this scenario, why would the user accept a system rating (the return of observability) based on 100? Let's be consistent and keep a rating of five everywhere.

Brown field first
: Having a green field approach for your first data product will slow you down. The team will spend an enormous amount of time finding and validating the sources, building a new pipeline to a hydrate a datastore that has not been modeled yet. By shifting to a product mindset, you find a more byte-sized starter product and focus on the value the data product will deliver based on existing

data. It allows you to get a fail-fast approach that will allow you to pivot more quickly if things go south.

Adopt modern software engineering methodologies
: You may want to experiment with some specific features in your data product, like turning it as an agent or an MCP server. Feature flagging can help you turn on and off some features for some specific users. *Feature flagging* is the art of shipping code to production with a big red *not yet* switch, so you can turn features on, off, or oops-back-on without redeploying.

Summary

Here's what you learned in this chapter, in a practical, (highly) opinionated, and allergic to fluff way:

- Data products are not files. Files are passive. Data products are active, self-aware systems.
- An active data product knows its state, freshness, dependencies, quality, and failures and should tell you before downstream explodes.
- Architecture matters because activity requires execution, not just documentation.
- Static assets scale chaos; data products scale trust.
- Data products can be implemented in three ways: centralized (easy, popular, SPOF-friendly), autonomous (self-contained, resilient, future-proof), or hybrid (a political compromise).
- Autonomous data products own their behavior, state, and metadata, typically via an internal database I call metaDb.
- Sharing infrastructure (like a global metaDb) is sometimes required, but usually comes with architectural tax.
- The sidecar is the real hero: it adds behavior without polluting business logic, and standardizes enforcement, observability, policy, and lifecycle.
- Libraries execute; sidecars negotiate.
- Sidecars enable the following: contract enforcement (ODCS), versioning and safe evolution, observability and SLAs, access control and RBAC, metadata sync and discovery, monitor cost, usage, feedback, and AI-readiness signals.
- Start with a minimum viable sidecar to avoid platform sprawl and analysis paralysis.
- Security is contract-driven and enforced at runtime, or it doesn't exist.
- Focus on outcomes, not pipeline internals.
- Favor mainstream tech, consistent user experience, and brownfield wins.

- Avoid edge-case rabbit holes; value first, elegance later.
- Don't be afraid of creating too many data products.
- If your data product can't explain itself, it's not a product, it's a dataset in denial.