Composable Reference Approach

Proposal for ODPS 4.0 model

opendataproducts.org/components

10th May 2025

By Jarkko Moilanen, Maintainer

Linux Foundation Project

A standard framework for creating, managing, and sharing data products using a programmatic approach.

THE LINUX FOUNDATION

Composable Reference Approach

Proposal for ODPS 4.0 model

opendataproducts.org/components

10th May 2025

Linux Foundation Project

A standard framework for creating, managing, and sharing data products using a programmatic approach.

THE LINUX FOUNDATION

Open Data Product Specification Core Team

Dr. Jarkko Moilanen
Igniter and maintainer

Manfred Sorg
Maintainer

DSc. Toni Luhti
Commercial operations

Tekla Wannas
Ecosystem & Marketing

Antti Poikola
Data Architecture Specialist

Zaher Abou Shakra
AI Engineer

Open Data Product Specification Timeline

2020-2021

Research and prototyping
300+ data products analyzed
Insights from 200+ articles

1.0

2/2022

SLA Object
7 Data Product Pricing Models

2.0

4/2023

Data Quality Object
DataOps

Part of Open Collective

2.1

8/2023

11 Data Product Pricing Models

3.0

5/2024

Data Quality as Code
SLA as Code
Defaulted to YAML

Joined The Linux Foundation

3.1

11/2024

Data Contract Support
Pricing Plans as Code

4.0

8/2025?

Component-based structure
AI Agents Support
MCP server

Limitations of 3.1 version

While I was defining monetizable data products for Data Product Monetization MasterClass with ODPS latest version, I made a few observations

LIMITATIONS

- Data Quality, SLAs, and Access point are defined once on Document level
- Pricing plans are separate object, but
  - In each plan it is likely that SLAs, Data Quality, and even access is different in plans
  - Even payment gateway is most likely different for humans oriented plans and AI agents focused.
- ODPS v3.1. is for single customer type with 1 access type (API, File, SQL), 1 DQ level, 1 SLA, 1 payment gateway

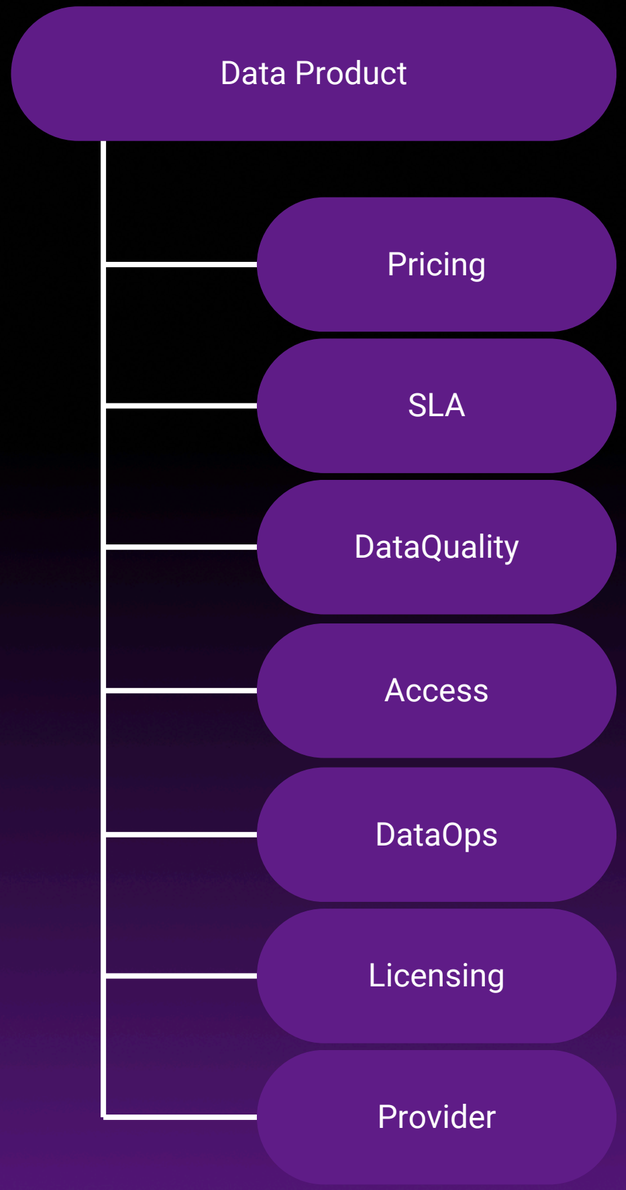
SOLUTION

- Data Quality, SLAs, Payment Gateway, and Access point defined per pricing plan
- Must be able to use also without pricing plans
  - Which is achieved if DQ, SLA, Payment Gateway, and Access is defined as reusable components.
- Somewhat same approach as in OpenAPI spec 3.0 forward

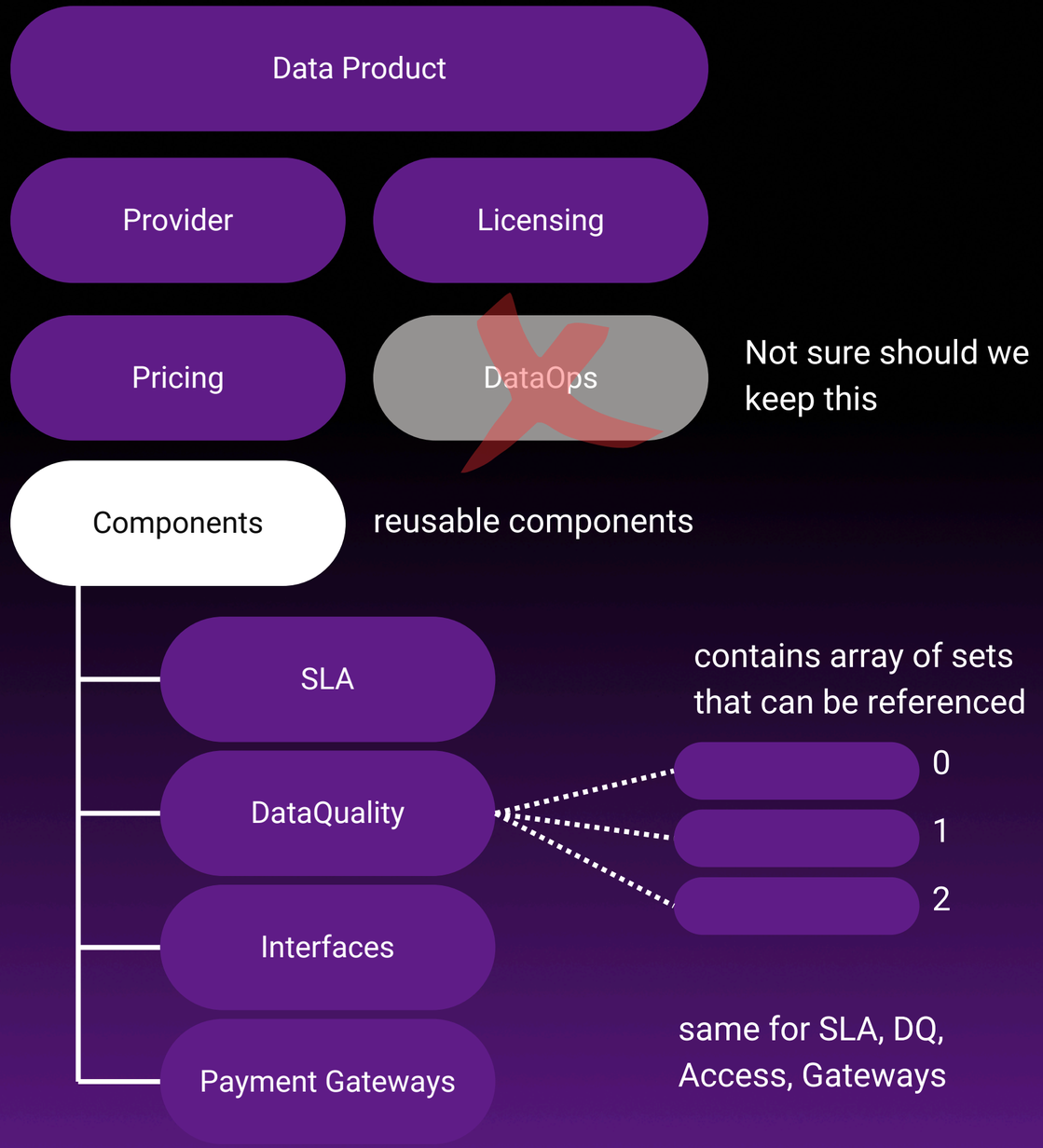
ODPS

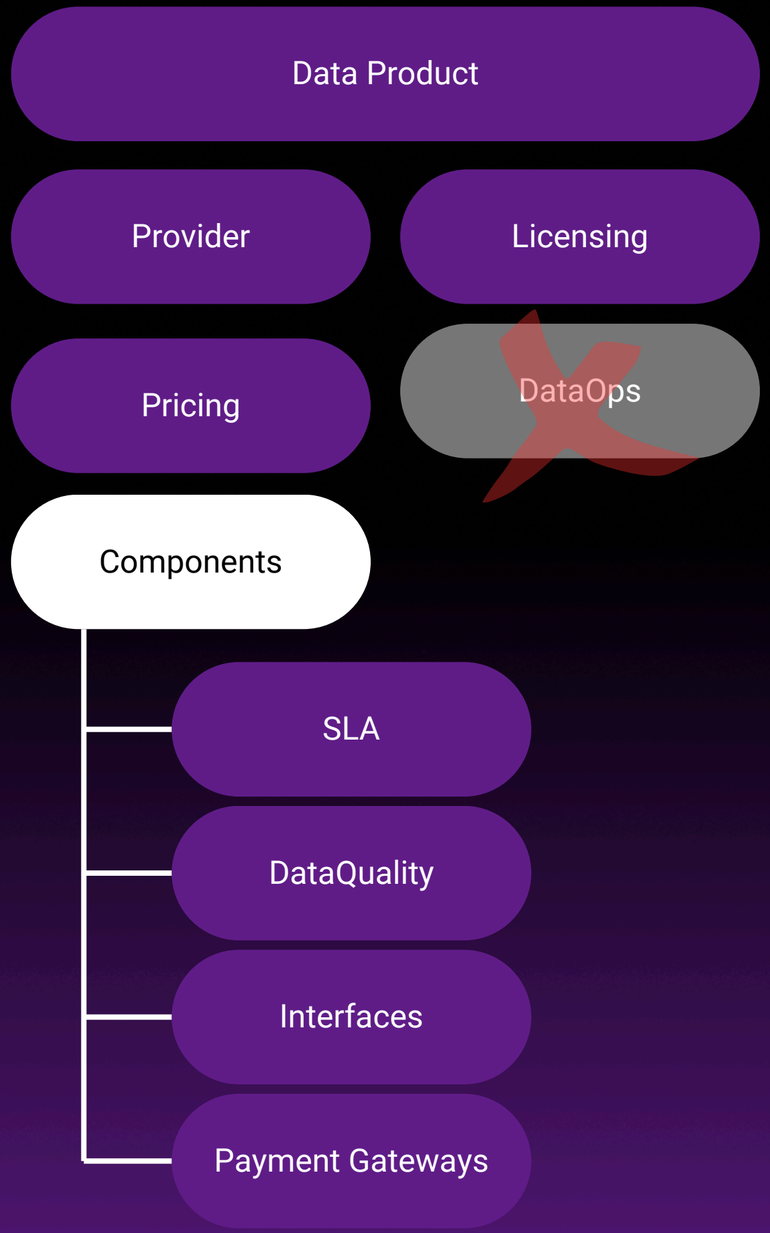
THE LINUX FOUNDATION

ODPS VERSION 3.1



ODPS VERSION 4.0 PROPOSAL





ENABLES 3 PATTERNS

1

Referenced Component Pattern

The data product defines operational metadata (like SLA, DQ, Interface, Payment Gateway) using $ref to a reusable object in the components section.

2

Inline Definition Pattern

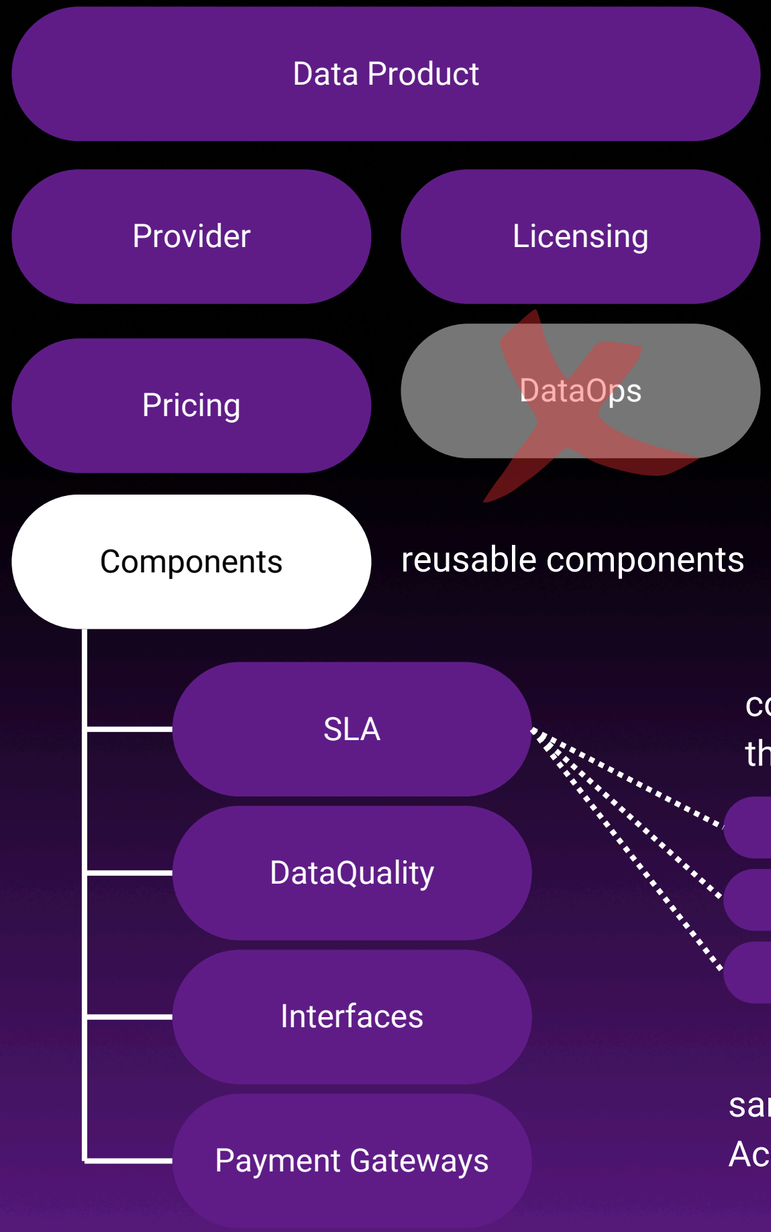
Operational metadata is defined directly and fully within the data product spec, not referenced from components. Similar to v 3.1.

3

Pricing-Driven Referencing Pattern

Each pricingPlan in the data product specification drives the selection of operational components (SLA, DQ, Access, Payment Gateway) via $ref links to central components.

ODPS VERSION 4.0 PROPOSAL



Not sure should we keep this

reusable components

contains array of sets that can be referenced

0
1
2

same for SLA, DQ, Access, Gateways

1

Referenced Component Pattern

The data product defines operational metadata (like SLA, DQ, Interface, Payment Gateway) using $ref to a reusable object in the components section.

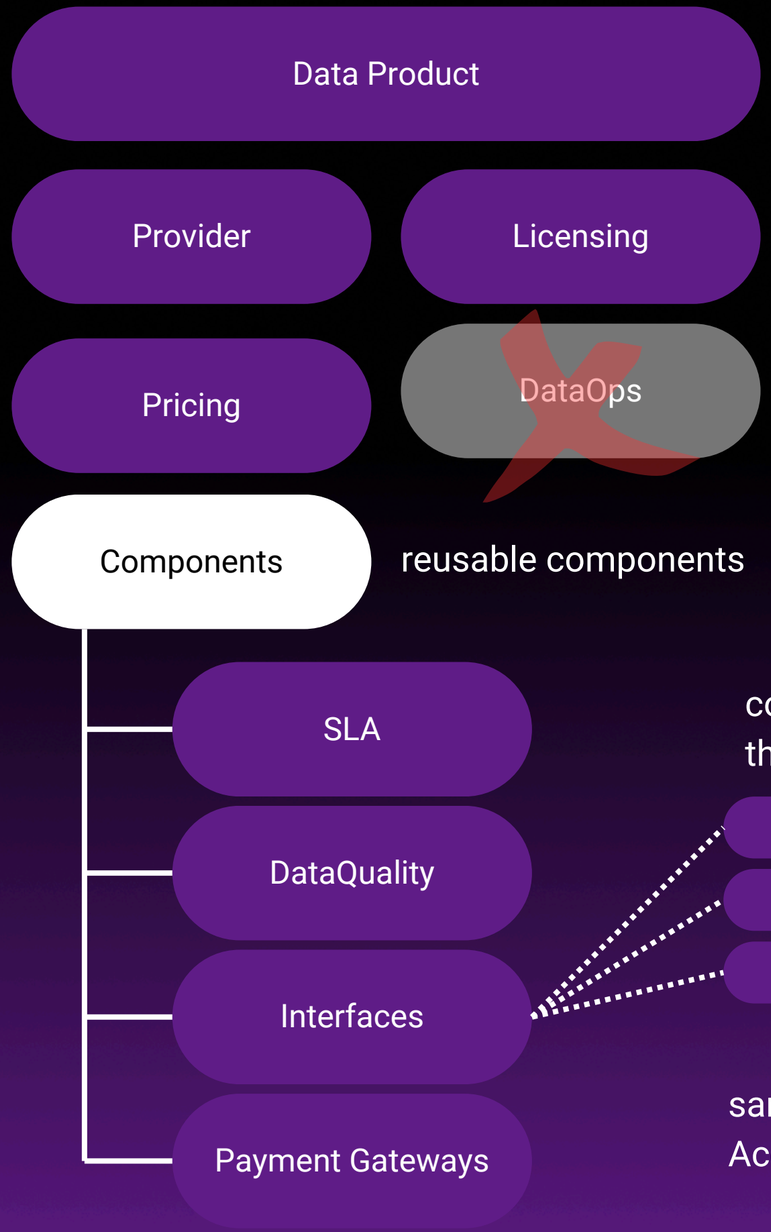
```yaml
schema: https://opendataproducts.org/v3.0rc/schema/odps.yaml
version: 3.0
product:
  en:
    name: Pets of the year
    productID: 123456are
    visibility: private
    status: draft
    type: derived data

  serviceLevel:
    $ref: "#/components/slaSets/1"

  access:
    $ref: "#/components/interfaces/0"

  dataQuality:
    $ref: "#/components/dataQualities/0"

  paymentGateway:
    $ref: "#/components/paymentGateways/1"
```

become just wrapper shells with $ref pointers.

Number is position in aray. In here 1st in interfaces component

perhaps can be references without array pos as well $ref: "#/Components/interfaces

ODPS VERSION 4.0 PROPOSAL



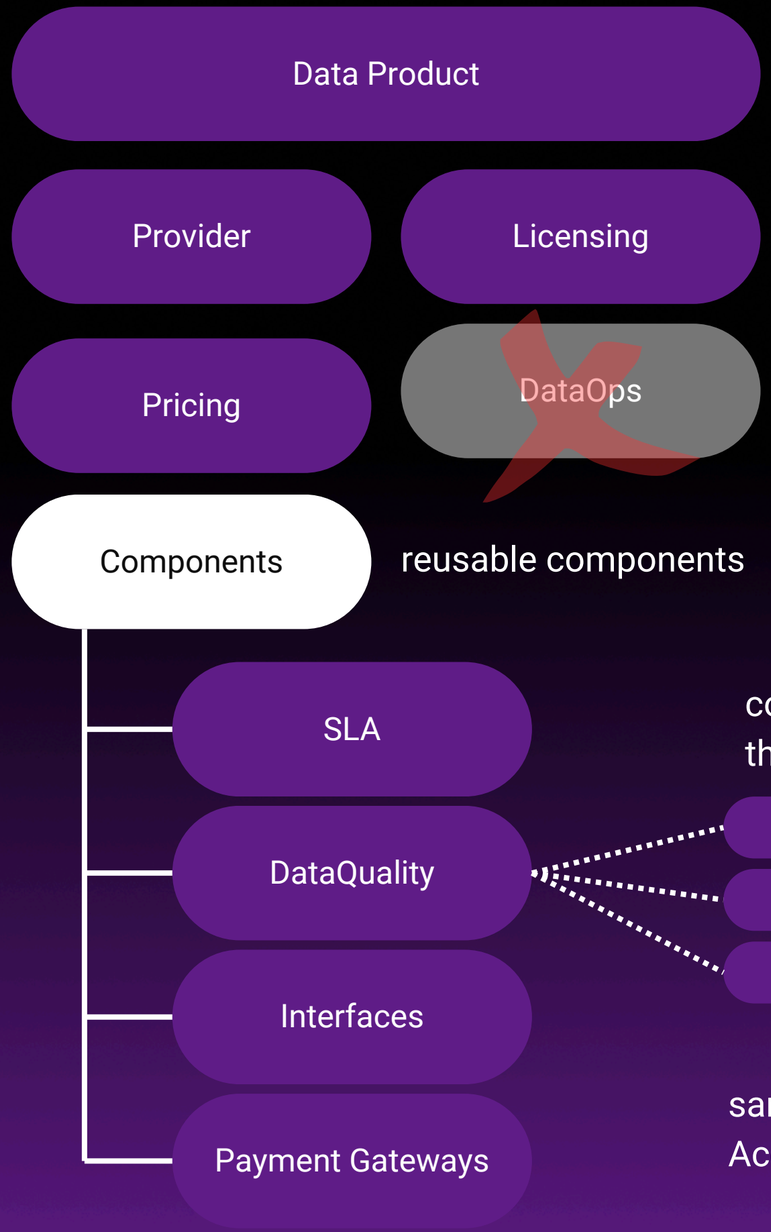
2 Inline Definition Pattern

Operational metadata is defined directly and fully within the data product spec, not referenced from components.

```yaml
schema: https://opendataproducts.org/v3.0rc/schema/odps
version: 3.0
product:
  en:
    name: Pets of the year
    productID: 123456are
    visibility: private
    status: draft
    type: derived data

  serviceLevel:
    slas:
      - dimension: uptime
        objective: 99
        unit: percent

  access:
    interfaces:
    - name: API
      description:
      - en: REST API for real-time event data access.
      authenticationMethod: OAuth
      specification: OAS 3.0
      format: REST
      specsURL: https://urbanpulse.ai/urbanpulse.json
      documentationURL: https://urbanpulse.ai/docs
```

CAN BE SIMILAR FLAT STRUCTURE LIKE IN V 3.1.

ODPS VERSION 4.0 PROPOSAL



Not sure should we keep this

reusable components

contains array of sets that can be referenced

0
1
2

same for SLA, DQ, Access, Gateways

3

Pricing-Driven Referencing Pattern

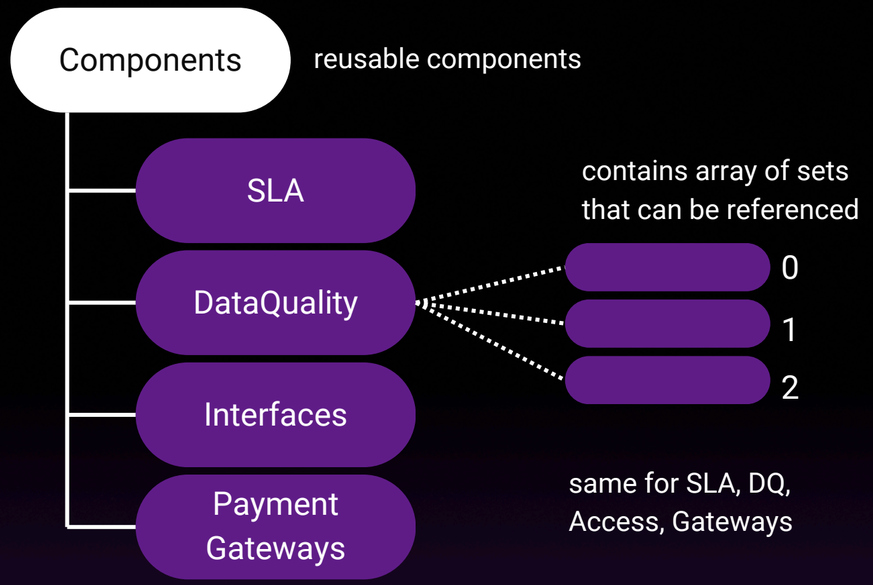
Each pricingPlan in the data product specification drives the selection of operational components (SLA, DQ, Access, Payment Gateway) via $ref links to central components.

```yaml
schema: https://opendataproducts.org/v3.0rc/schema/odps.
version: 3.0
product:
  en:
    name: Pets of the year
    productID: 123456are
    visibility: private
    status: draft
    type: derived data

  pricingPlans:
    en:
      - name: Basic Access
        price: 0.00
        unit: recurring
        billingDuration: month
        offering:
          - 10,000 API calls/month
        serviceLevel:
          $ref: "#/components/slaSets/0"
        dataQuality:
          $ref: "#/components/dataQualities/0"
        interface:
          $ref: "#/components/interfaces/0"
        paymentGateway:
          $ref: "#/components/paymentGateways/0"
```

Each plan can have different but reusable SLA, DQ, Interfaces, and payment gateways

Number is position in aray

ODPS COMPONENTS EXAMPLE DRAFTS




```yaml
components:
  dataQualities:
    - dqName: Basic
      description:
        - en: Minimum acceptable quality for open data publishing
      metrics:
        - dimension: accuracy
          objective: 98
          unit: percentage
        - dimension: completeness
          objective: 90
          unit: percentage
```

$ref: "#/Components/DataQualities/0

```yaml
Components:
  interfaces:
    - name: API
      description:
      - en: REST API for real-time event data access.
      authenticationMethod: OAuth
      specification: OAS 3.0
      format: REST
      specsURL: https://urbanpulse.ai/urbanpulse.json
      documentationURL: https://urbanpulse.ai/docs

    - name: Agent
      description:
      - en: MCP interface for structured data access and agent interaction.
      authenticationMethod: Token
      specification: MCP 2025-03-26
      format: MCP
      specsURL: https://urbanpulse.ai/llms.txt
      documentationURL: https://urbanpulse.ai/llms-full.txt
```

$ref: "#/Components/Interfaces/0

$ref: "#/Components/Interfaces/1

```yaml
components:
  paymentGateways:
    - name: Agent
      type: Axio
      version: 1.0
      reference: https://www.x402.org/
      spec: |
        paymentMiddleware("0xYourAddress", {"/your-endpoint": "$0.01"});

    - name: API
      type: Stripe
      version: 1.0
      reference: https://docs.stripe.com/
      spec: |
        link or code to ignite purchase
```

$ref: "#/Components/PaymentGateways/0

$ref: "#/Components/PaymentGateways/1

3 patterns overview

Pattern Name	Defined In	Reuse Support	Plan-Aware	Typical Use
Referenced Component	Document-level via `$ref`	✅ High	❌ No	Centralized control, reusable objects
Inline Definition	Document-level inline	❌ None	❌ No	Custom logic, flexibility
Pricing-Driven Referencing	Per-pricingPlan via `$ref`	✅ High	✅ Yes	Tiered service delivery per plan

Does that solve the problem? Yes it does

Pros and Cons

Aspect	✅ Component-Driven Model	❌ Inline-Only Model
Reusability	✅ Define once, reuse across products or plans	❌ Every product defines everything anew
Pricing Plan Support	✅ Enables per-plan SLAs, DQ, interfaces, etc.	❌ Cannot differentiate services per plan
Scalability	✅ Clean scaling for catalogs with 10s–1000s of products	❌ Becomes redundant and hard to maintain
Governance	✅ Easier to audit, version, and track shared definitions	❌ No centralized visibility or control
Automation-readiness	✅ Perfect for code generation, validation tools	❌ Parsing logic gets duplicated
Modularity	✅ Clear separation of concerns via components	❌ Everything is mixed together
Cognitive Load (early)	❌ Higher for new or small teams	✅ Easy to grasp and prototype quickly
Tooling Complexity	❌ Requires `$ref` resolution, fallback logic	✅ Minimal logic needed for basic tooling
Readability (raw YAML)	❌ Harder to read directly (details hidden)	✅ All key info is visible in one file

WHY IT'S STILL WORTH IT

Despite these drawbacks, the component-driven model offers clear long-term benefits:

- Modularization
- Reuse
- Clear separation of concerns
- Pricing-driven service modeling
- Easier automation and policy enforcement

Needs a few guide videos. Crucial is to communicate the patterns, that opens the logic of proposed model

We’re moving from static definitions to dynamic, reusable components — built to scale, govern, and monetize data products.

This new component-driven model isn't just a technical improvement

IT'S A STRATEGIC FOUNDATION

opendataproducts.org