Microservice – Data Sharing

Patterns and Anti-Patterns

Kim Horn

Version 1.3 **1 Jan 2023**

Agenda

- Core Principles
- Challenges of Sharing Data
- Ant-Patterns:
  - Data Base Pull
  - Batch Pull
  - Batch Pull CDC
- Patterns:
  - HTTP Pull
  - Data Pump
  - Data Pump – Data Lake
  - Data Pump - Asynchronous Event Pushing
  - Kafka Connect
- Data Mesh
- Solving the Interoperability problem
- Conclusion

The Data Sharing Problem

As Data is a resource, it should be used across the organization, for other business processes, reporting and big picture analytics, However:

- Centralised Data teams become a bottleneck as all analytics/reporting request go through them. All ‘raw’ data has to be available for them. Knowledge has to be duplicated.
- Data is moved and shipped around the organization, from one business unit to central team to another. The correct data gets lost, as lineage is too complex to manage.
- Teams take data – other teams need your data so they just take it, no idea of quality or lineage, with no contract. Integrations appear and grow, creating dependencies and quality issues.
- Impact on production system – other teams may run queries on your production data base to get data, reducing the performance for customers. ETL jobs get run at the wrong times.
- The data from one team is not interoperable with another, they do not know the format, meaning of domain terms, are not sure if its correct. Therefore may use it the wrong way.
- The cost to find and discover data, work out its quality, and meaning by other teams is massive.
- Microservices own their data and do not expose it for others. *This seems a fundamental issue for other non-microservice teams.*

Microservice Architecture Principles

Microservices are:

1. Modelled around a bounded context; the business domain, (Domain Driven Design)
2. Responsible for a single capability;
3. Individually deployable, based on a culture of automation and continuous delivery;
4. The owner of its data;
5. Consumer Driven, (Consumer Driven Contracts)
6. Not open for inspection; Encapsulates and hides all its detail (including data)
7. Easily observed;
8. Easily built, operated, managed and replaced by a small **autonomous** team;
9. A good citizen within their ecosystem;
10. Based on Decentralisation and Isolation of failure;

Microservice Principles Implications

Microservice architectures, as distributed systems, must follow certain principles, without these applied the architecture will not provide benefits, implications of principles:

- **Share Nothing Architecture** – Microservice shares nothing at all. Data bases are not shared. Code is not shared.
- **Database per Microservice** – Each Microservice owns its data, completely encapsulates its data base. The DBMS and schemas can change anytime with no external impact.
- **Keep Bounded Contexts Isolated** – do not share models between contexts (teams and applications). In the context of a DBMS the database and its schema are never exposed, not available for external usage.
- **Decoupling** - Services should be completely decoupled at run time and code time. Reduces impacts of change and provides team autonomy. Data Base code , schemas, and data are not shared.

Microservices = decoupled code, decoupled teams

- The main benefit of microservices is team autonomy, which provides the business with agility.
- As soon as you get coupled code, teams get coupled, agility is lost and microservices loose their benefit.
- You need to work to keep teams protected from external dependencies

Microservices are all about Business Agility based on Team Autonomy

Autonomy makes the team agile, and nimble; in control of their destiny. This in turn makes the business agile and able to react quickly to customer needs.

The benefits of a microservice architecture are not technical but social. When teams lose their independence, when the microservices, code or databases are shared by other teams then coupling occurs. Microservice then start to lose their value to the business and customers.

Bounded Context

A Bounded Context is an internally consistent system with carefully designed boundaries that mediate what can enter the system, and what can exit it.

A Bounded Context represents its cross-functional team as well.

Bounded Context helps enforce loose coupling *externally*, defining explicit points where:

- *external* data can *enter* (perhaps via consumer subscribed to a Kafka topic)
- *internal* data can *exit* (maybe via another Kafka topic, or via a well-design GET API, carefully crafted to hide any internal system details)

In Patterns, Principles and Practices of Domain-Driven Design, the importance of isolating bounded contexts is stated:

"To retain the integrity of your bounded contexts and ensure they are autonomous, the most widely used approach is a shared-nothing architecture where each bounded context has its own codebase, datastore and team of developers"

Data Sharing Creates Challenges – Can Break Principles.

Data Sharing is a difficult challenge for Microservice based architectures, due to:

- Data bases being hidden, they should not be shared.
- Reports may require data from many microservices.
- Often done by a separate team in the organisation, separate bounded contexts.

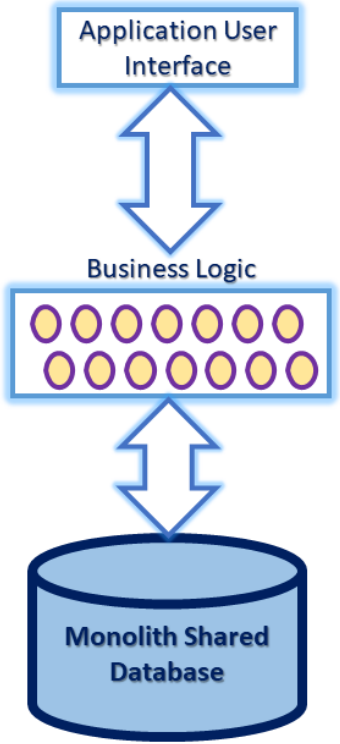
Sharing Creates tension between performance and microservice principles, can easily create coupling, breaking the isolation between separate bounded contexts:

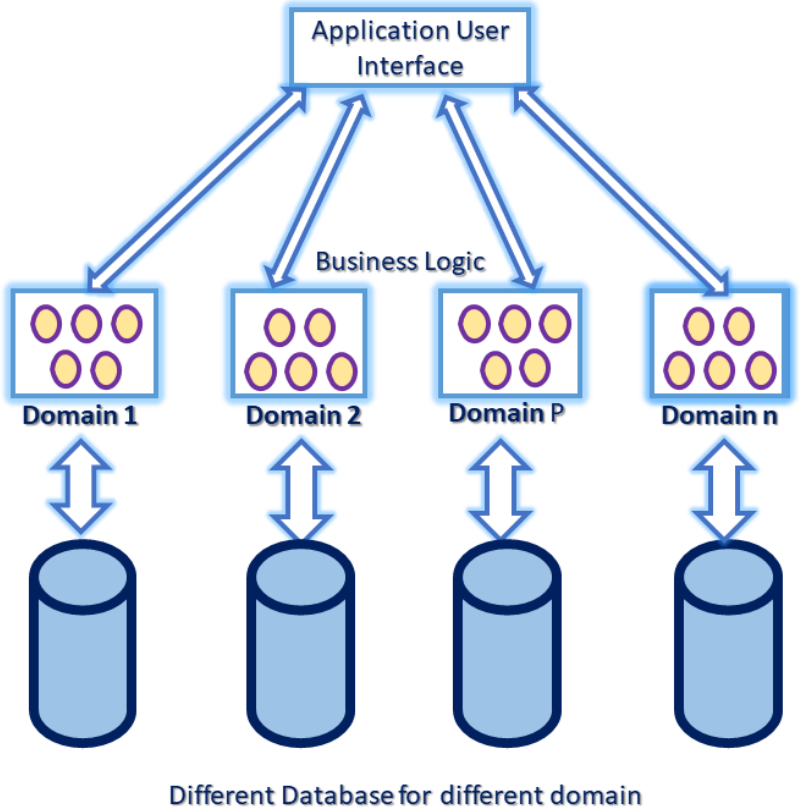
- Fastest performance is by other teams directly accessing the data in the data base. Breaks Principles: 1, 4 & 6
- Separate teams start using your services and code. Breaks Principles: 1, 2, 6 & 8

The following slides discuss a number of ant-patterns, patterns and solutions. The impact of each will be measured against these goals, with the first being the most important:

- **Bounded Context:** Poor, Medium, Good or very Good
  - Keeps domains (teams) isolated, improves agility, autonomy, maintainability. Separate contexts have their relationships (dependencies) made explicit (see Appendix):
- **Performance:** High, Medium or Low
- **Timeliness:** Poor, Medium, Good

Monolithic DBMS to DBMS per Domain



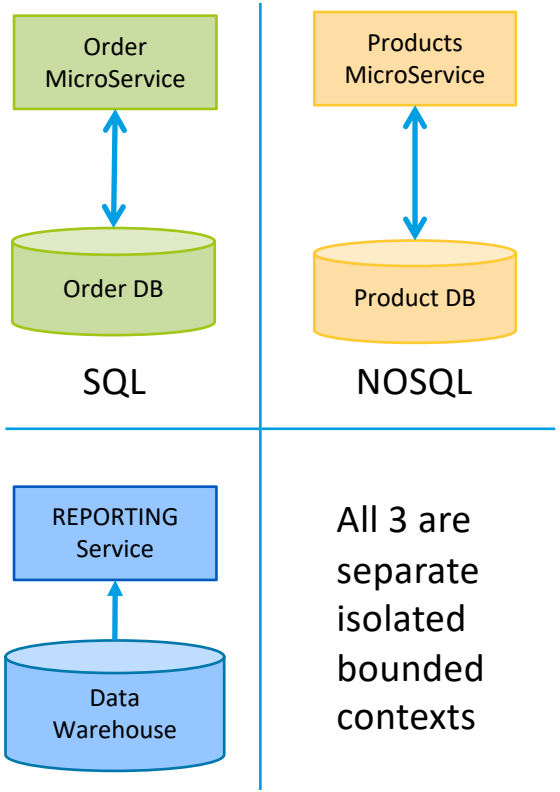


Different Database for different domain

Data Base Per Service – never share.

Benefits of a Database per Microservice:

- **Resiliency:** Changes to an individual database don't impact other services. Thus, there isn't a single point of failure in the application. Change is easily contained.
- I**ndividual data stores are easier to scale**. Moreover, the domain's data is **encapsulated** within the microservice. Therefore, it's easier to understand the service with its data as a whole. It's especially important for new members of a development team. It will take less time and effort for them to fully understand the area they're responsible for.
- **Polyglot persistence.** It means that we can use different database technologies for different microservices. So one service may use an SQL database and another noSQL. Choose the most efficient DBMS depending on the service requirements and functionality.
- Teams are autonomous, no one needs to use another's DB.



The 3 colors represent the 3 teams (bounded contexts) – thus what software each owns.

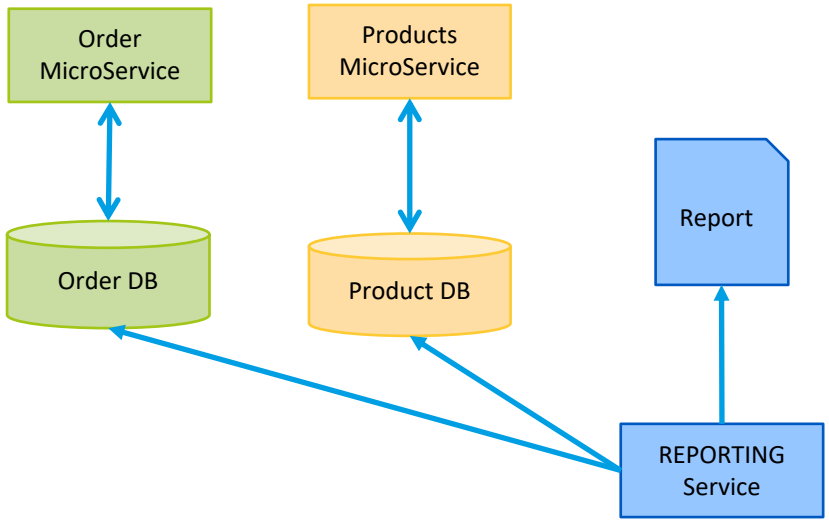
Anti-Patterns

Data Base Pull

Anti-Pattern

The consuming service reads the two data bases directly, and pulls the information:

- **Shared DB Style.** The Order and Product DBs no longer own their data
- **High Coupling** – Modification to any of the microservices data impacts the reporting service:
  - The data content can't change
  - The data meaning can't change
  - The data format can't change
  - The data tables/collections can't change
  - The DBMSs can't change



- **Bounded Context:** Poor - Highly coupled
- **Performance**: High
- **Timeliness:** Medium – Good

The reporting service will poll the DBs and the frequency will determine timeliness of data but these queries will impact the Microservices.

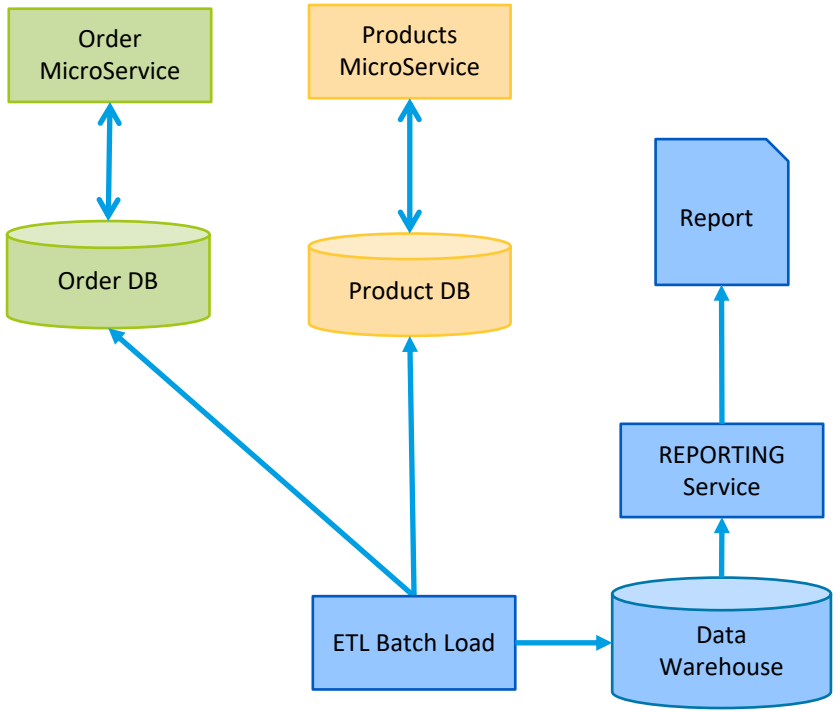
Batch Pull

Anti-Pattern

Pull the data directly, but reduce the impact on the Microservices and use a batch job, out of production times, to get all the data:

- **Shared DB Style.** The Order and Product DBs no longer own their data, shared with ETL Program
- **High Coupling** – Modification to any of the microservices data impacts the ETL Batch Load

- **Bounded Context:** Poor – Highly Coupled
- **Performance**: Medium
- **Timeliness:** Poor – data will not be recent.

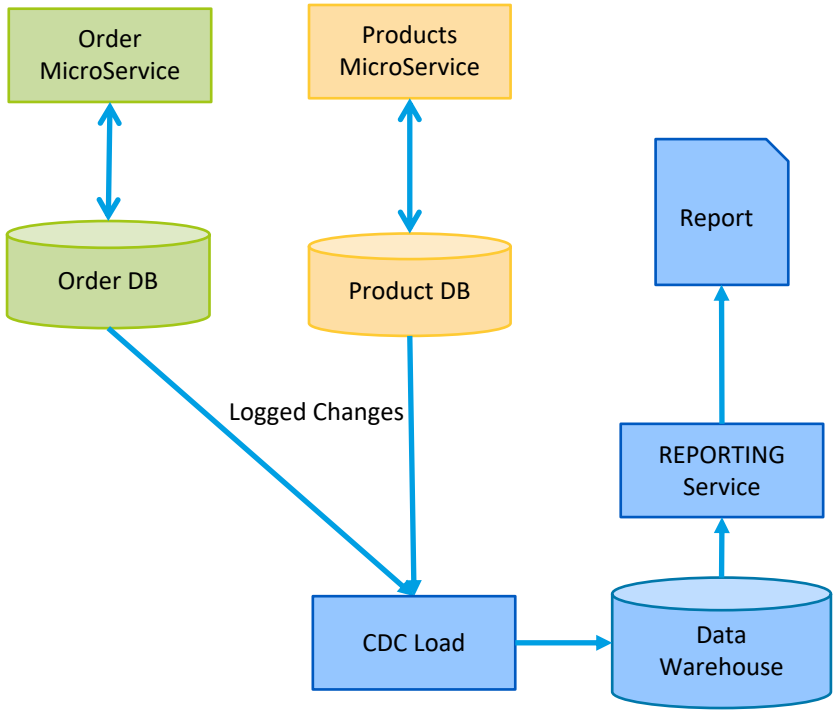


Batch Pull – CDC

Anti-Pattern

Pull the data directly, but use a CDC approach to get all the data.

- **Shared DB Style**. The Order and Product DBs no longer own their data
- **High Coupling** – Modification to any of the microservices data impacts the CDC Load. This time is not the microservice but the data base tables directly.
- Technology coupling of CDC program and the DB.
- Very Complex when the DBs are different technologies (SQL , NoSQL)



- **Bounded Context:** Poor - Breaks Isolation
- **Performance**: Medium
- **Timeliness:** Medium

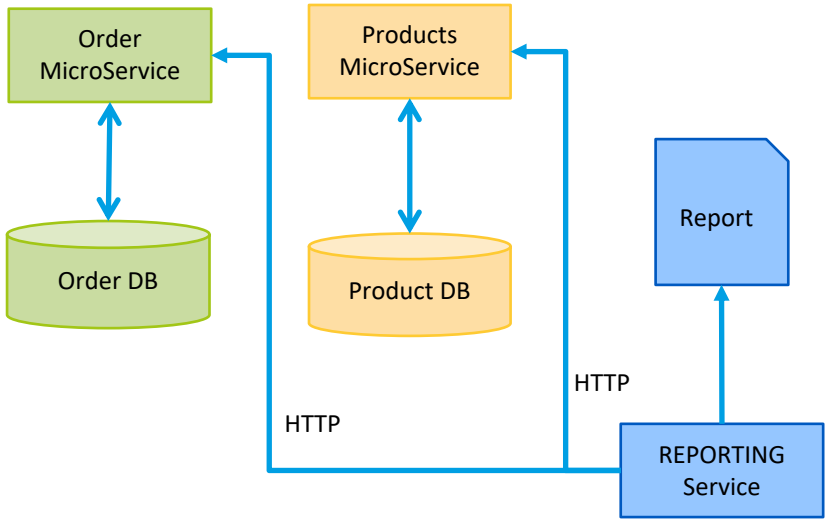
Similar pattern to shared Replica DBMS.

Patterns

HTTP Pull Pattern

Avoid decoupling by using the Microservice HTTP APIs, ask each microservice for its data.

- **No Shared DB**, Open Host Service Model
- **Low Coupling**
- However:
  - Slow
  - Data may be too big for HTTP
  - Impact microservice
- Service APIs may not be appropriate for reporting, which means creating 'artificial' APIs just for reporting.
- Complex retry/timeout logic when the synchronous API calls fail.



- **Bounded Context:** Good – Decoupled
- **Performance**: Low
- **Timeliness:** Medium to Good

The reporting service will poll the APIs and the frequency will determine timeliness of data but these queries will impact the Microservices.

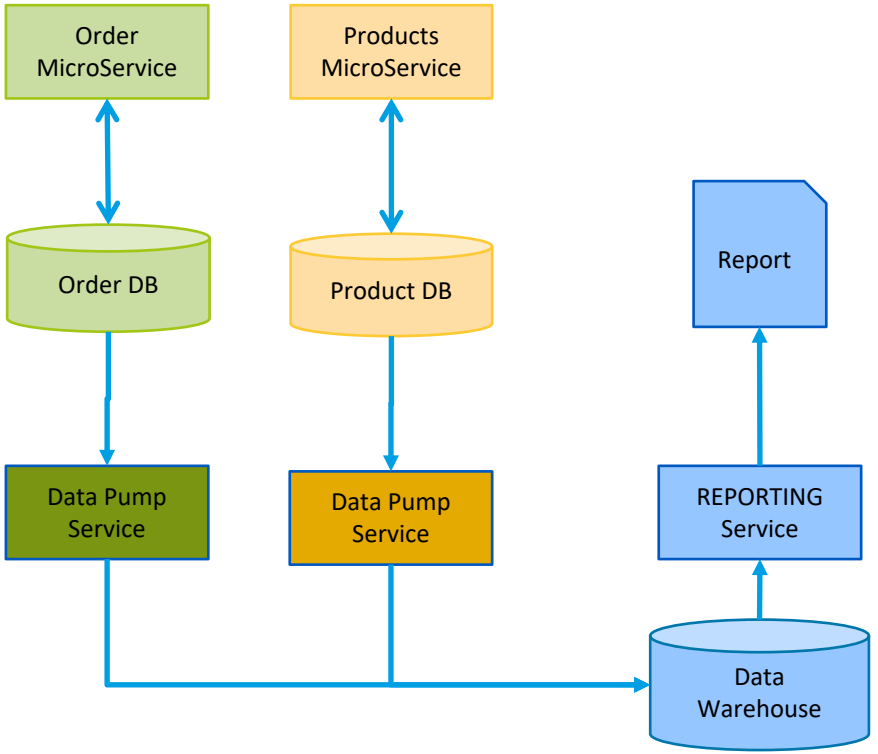
Data Pump

Pattern

Rather than pull the data, the Data Pump Pushes the data to the Data Warehouse:

- **No Shared DB.**
- Only required data or deltas pushed.
- Each Microservice team manages its Data Pump
- Now The Data pumps are coupled to the Data Warehouse but the microservices data (schema) are not.

- **Bounded Context:** Good
- **Performance**: Medium
- **Timeliness:** Low – Medium - Data Pumps runs out of peak only. Can be increased by using CDC as source for Pump.



Data Pump could use Debezium a product that reads a DB Transaction log

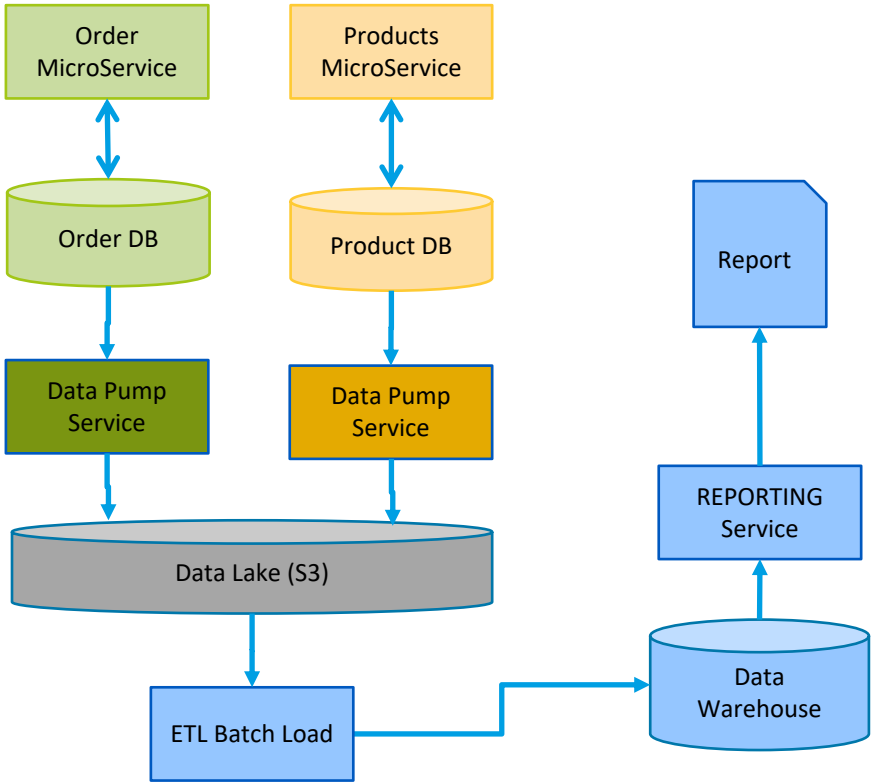
Data Pump to Data Lake (e.g. S3)

Pattern

Push using a Data Pump Pushing to a Data Lake:

- **No Shared DB.**
- Only required data or deltas pushed.
- Each Microservice team manages its Data Pump
- Consumers needs to know the Data Lake Table/File format (Contract). But have no knowledge of DB schema or Consumer.

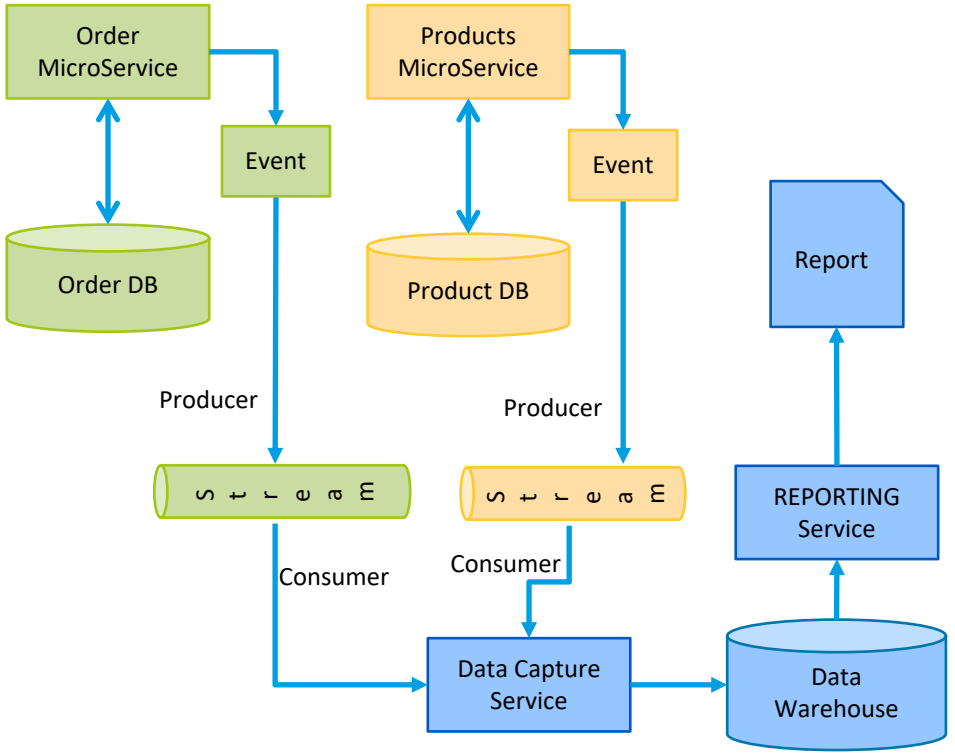
- **Bounded Context:** Very Good
- **Performance**: Medium
- **Timeliness:** Low – Medium - Data Pumps runs out of peak only. Can be increased by using CDC as source for Pump.



Data Pump - Asynchronous Event Pushing Pattern

A better solution to the above is Asynchronous Event Pushing. Each Microservice sends (produces) important Events to a Stream/Queue.

- Only required data pushed.
- **Highly Decoupled.** The Producers do not know about the Consumers.
- The Event data can be maintained despite changes to the Microservice Data, DBMS, or internal models.
- Consumers need only know the Event content format (Contract). But have no knowledge of DB schema or Consumer.
- **Bounded Context:** Very Good
- **Performance**: High
- **Timeliness:** High

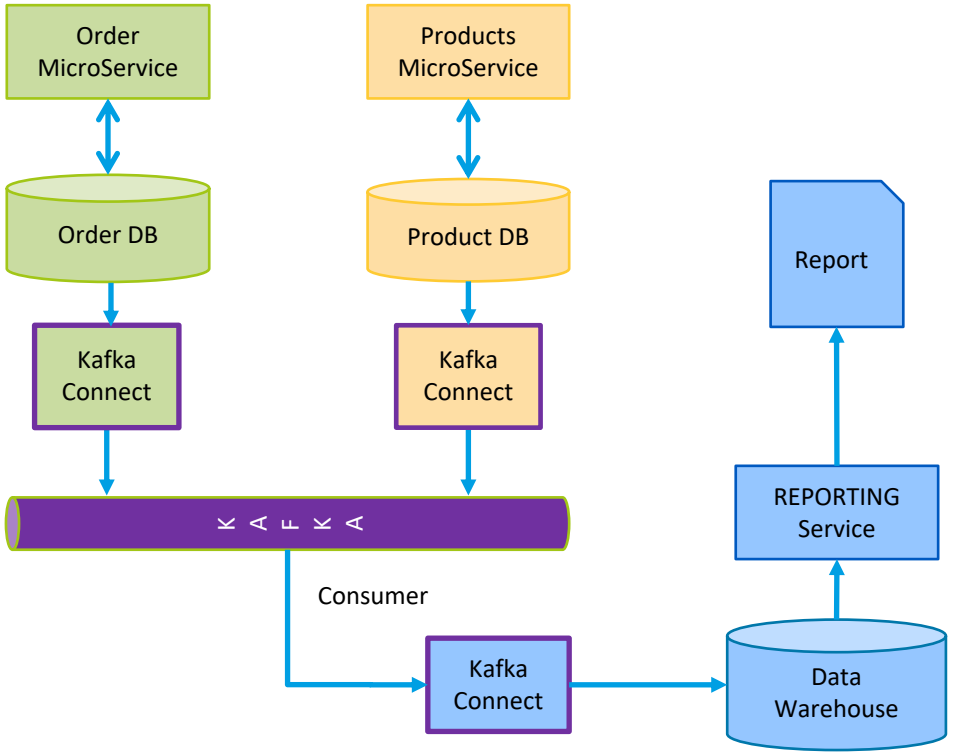


Kafka Connect - Asynchronous DB Change Pushing Solution*

Kafka Connect takes DB changes and publishes Messages to a Kakfa Topic, then pushes into Warehouse via Kafka Connect:

- A proprietary solution
- **Highly Decoupled**. The Producers do not know about the Consumers.
- Consumers need only know the Message content format (Contract). But have no knowledge of DB schema or Consumer.

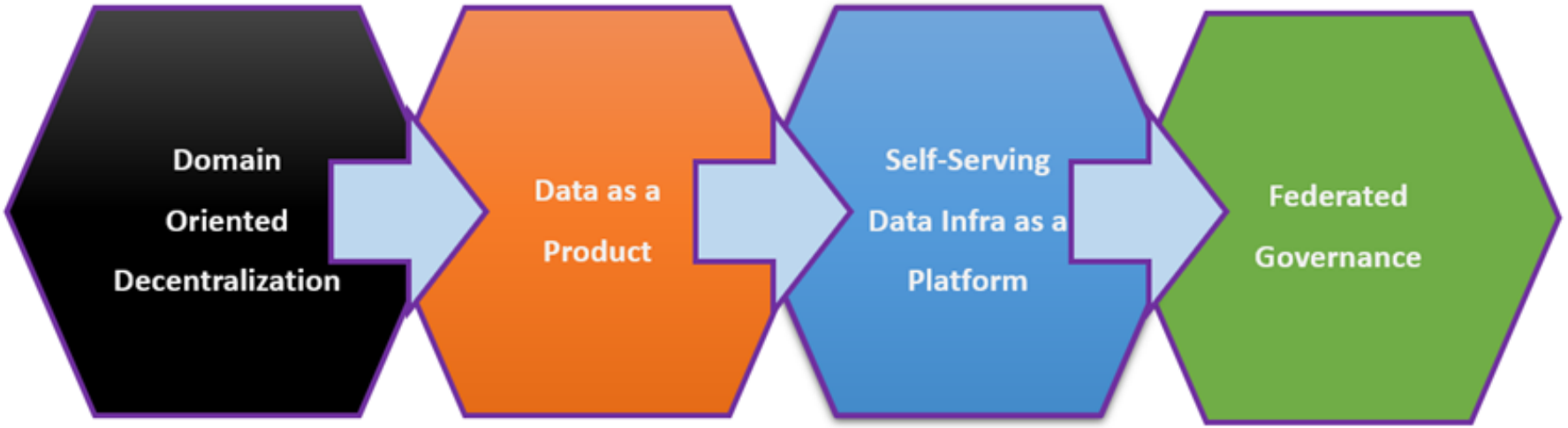
- **Bounded Context:** Very Good
  Performance: High
- **Timeliness:** Good



* Called 'Solution' as it's a product specific Pattern

Data Mesh Architecture

Data Mesh Principles



Principles

1. **Domain Oriented Approach**: The mass of data is broken down into domain-specific streams, which can be transformed to create a joint aggregate view of the business domain. These data streams are owned by independent teams or users, who are, in turn, attached to business experts who analyse the data for insights.
2. **Data as a Product**: Teams see data as their product and is responsible for processing the data to be used as a ready-to-use unit.
3. **Self-serve data infrastructure as a platform**: An automated platform allows the product team to control the lifecycle of data from the hands of a developer at the source to connect the data product and then run the semantic queries at mesh level. The platform provides storage, pipeline, data catalogue, and access control to the domains. It reduces any chance of duplication in effort and hence increases the efficiency of data treatment.
4. **Federated Governance**: The implementation of data mesh architecture is defined by federated governance with standards and interoperability as primary architectural guidelines. It emphasizes that each domain should be discoverable, addressable, self-describing, secure, trustworthy, and interoperable.

Why is this Important?

The data mesh architecture deals with the data more reliably and processes that information in real-time. Additionally, it can overcome multiple challenges of monolith data architecture.
Benefits:

1. It brings ownership and hence the accountability and responsibility of sharing that data as a product.
2. It improves quality as data is treated at the source and is good to be used as an independent unit.
3. With the data cleaned at the source, the responsibilities are equally shared and hence support organizational scaling.
4. It increases efficiency with the incorporation of data infrastructure as a platform.
5. It can handle a vast set of data without any difficulty, as they are domain-wise segregated for their respective domain experts to work on them.
6. The data can now be directly incorporated into data analytics tools, making it more feasible in operational values.

How is it different from monolith architecture?

Data is at the core of every organisational strategy. The data architecture is very critical to the efficiency, scalability, and usability of the data. And hence architecture in the core defines the insights of the data.

Over the past years, organizations have excelled with the various scaled forms of data architectures like monolith data architecture (data lake approach). But with expansion across domains, the centralised data platform architectures like monolith have significant challenges to deliver data with the speed and flexibility of the scaling organizations need.

Moving to the data mesh architecture from a monolith requires a mindset shift:

- Centralized ownership to decentralized ownership
- Pipelines as a priority to domain data as a first-class concern
- Data as a by-product to data as a product
- A siloed data engineering team to cross-functional domain-data teams
- A centralized data lake/warehouse to an ecosystem of data products

Data Mesh and Microservices Align

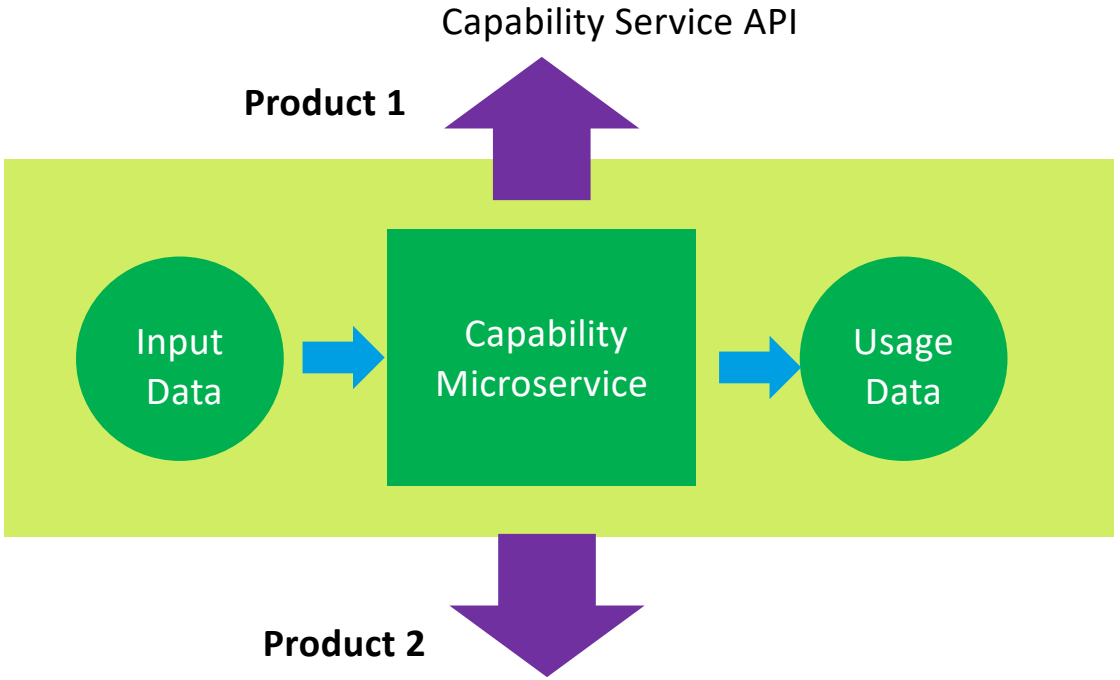
Microservice Architecture and Data Mesh Architecture align to

Solve the Data Sharing Problem

A Microservice may have 2 Outputs: Service and Data

Consider a Microservice team:

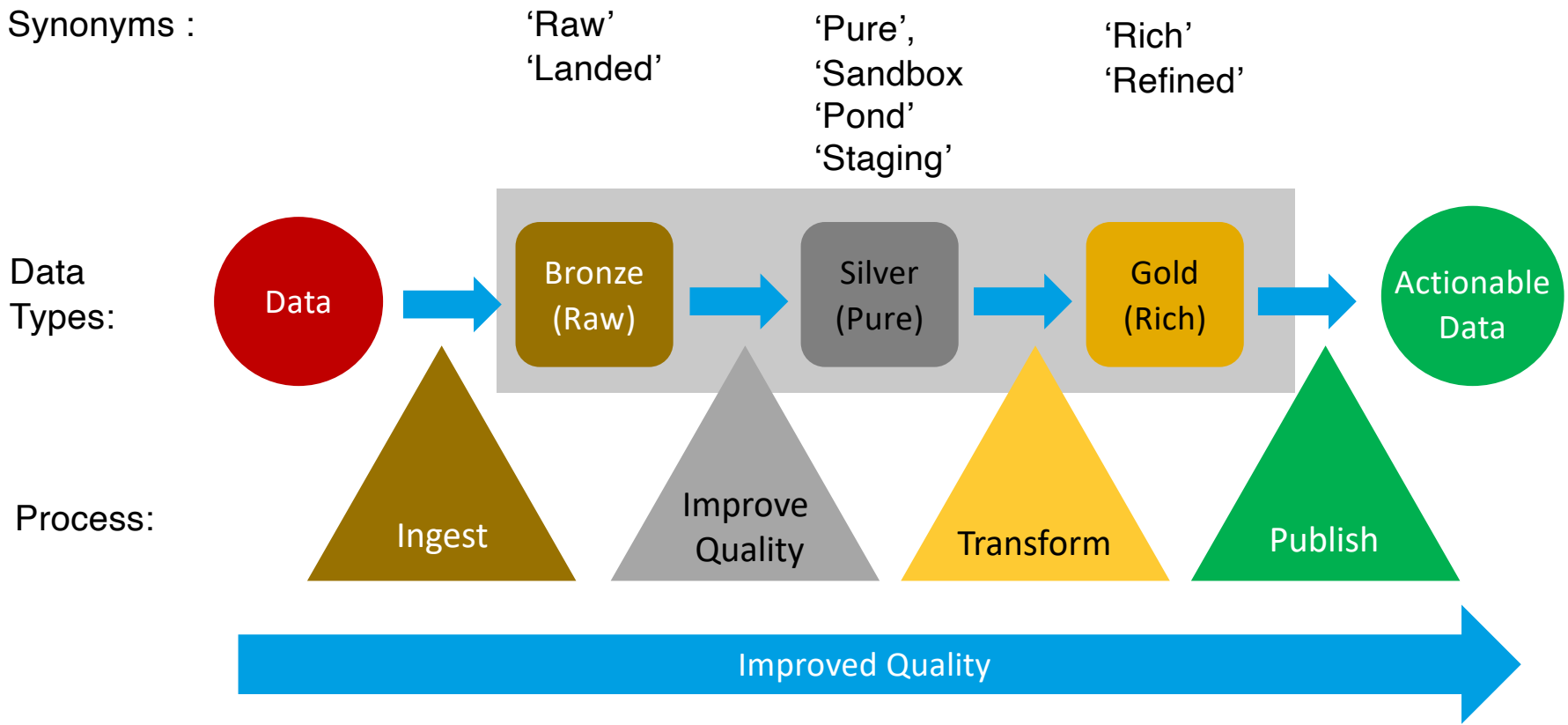
- This team knows its data; they are the organization's experts for their domain
- They processes the raw data from source.
- They design the Microservices to use that Data.
- They know the usage transactional data produced.
- They know the quality, meaning, and domain relevance of their data.



Data Outputs:

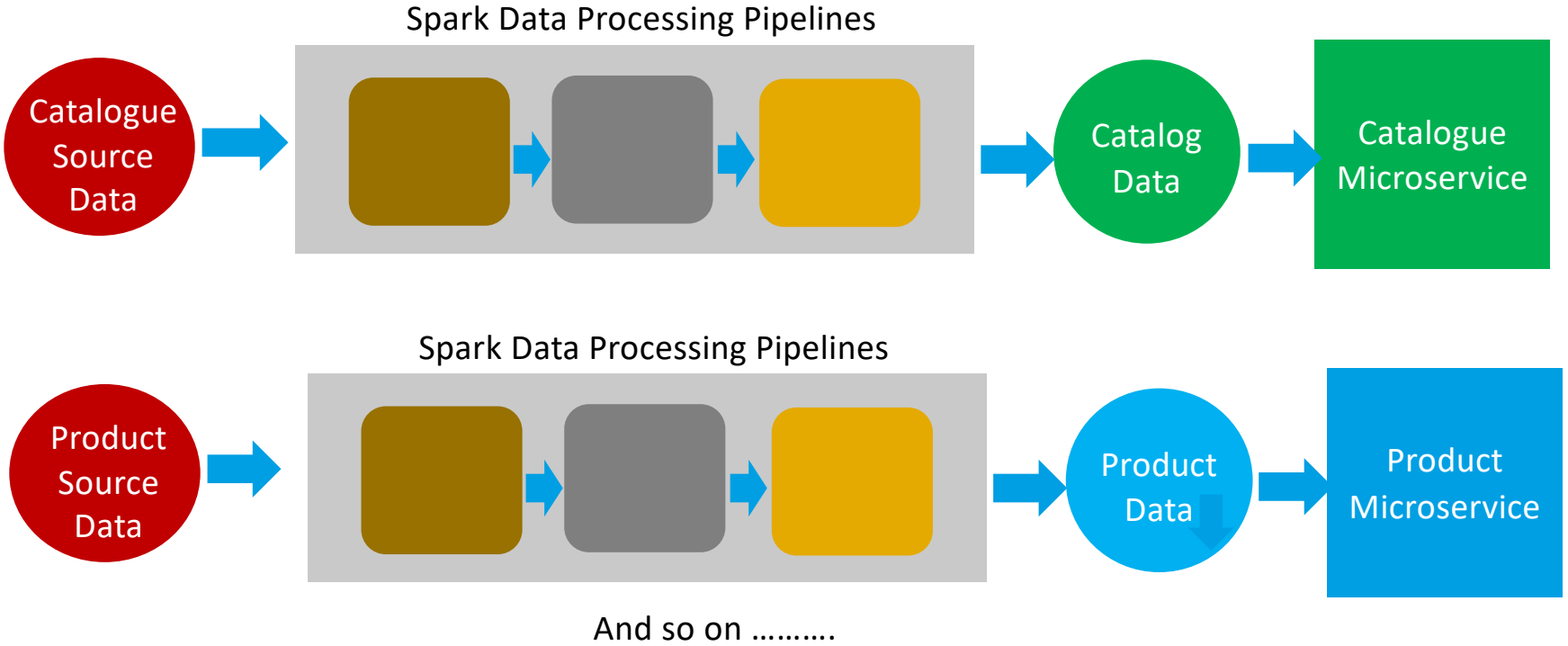
- Input Capability Data
- Transactional Usage

Multi-Hop - Data Pipeline - Data Quality Types



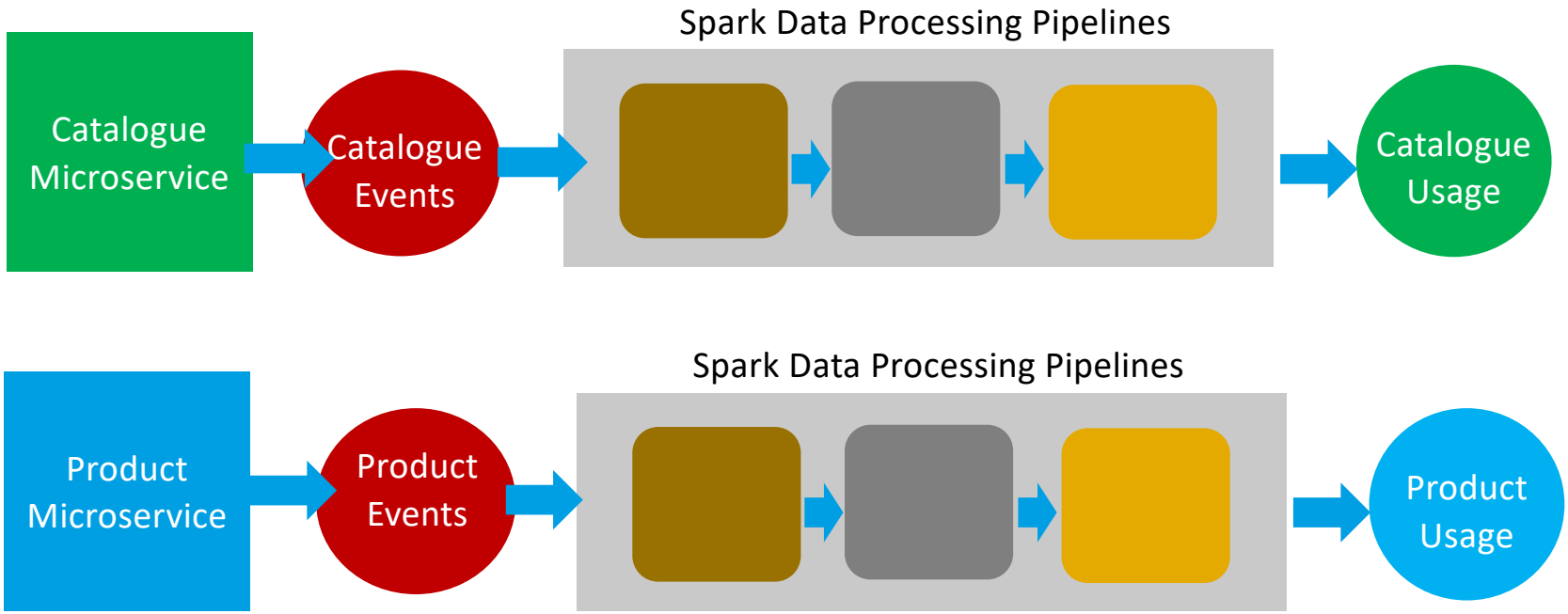
Capability Data – Source to Microservice Pipelines

Consider a eCommerce product catalogue system. Capability teams process the source data for each microservices



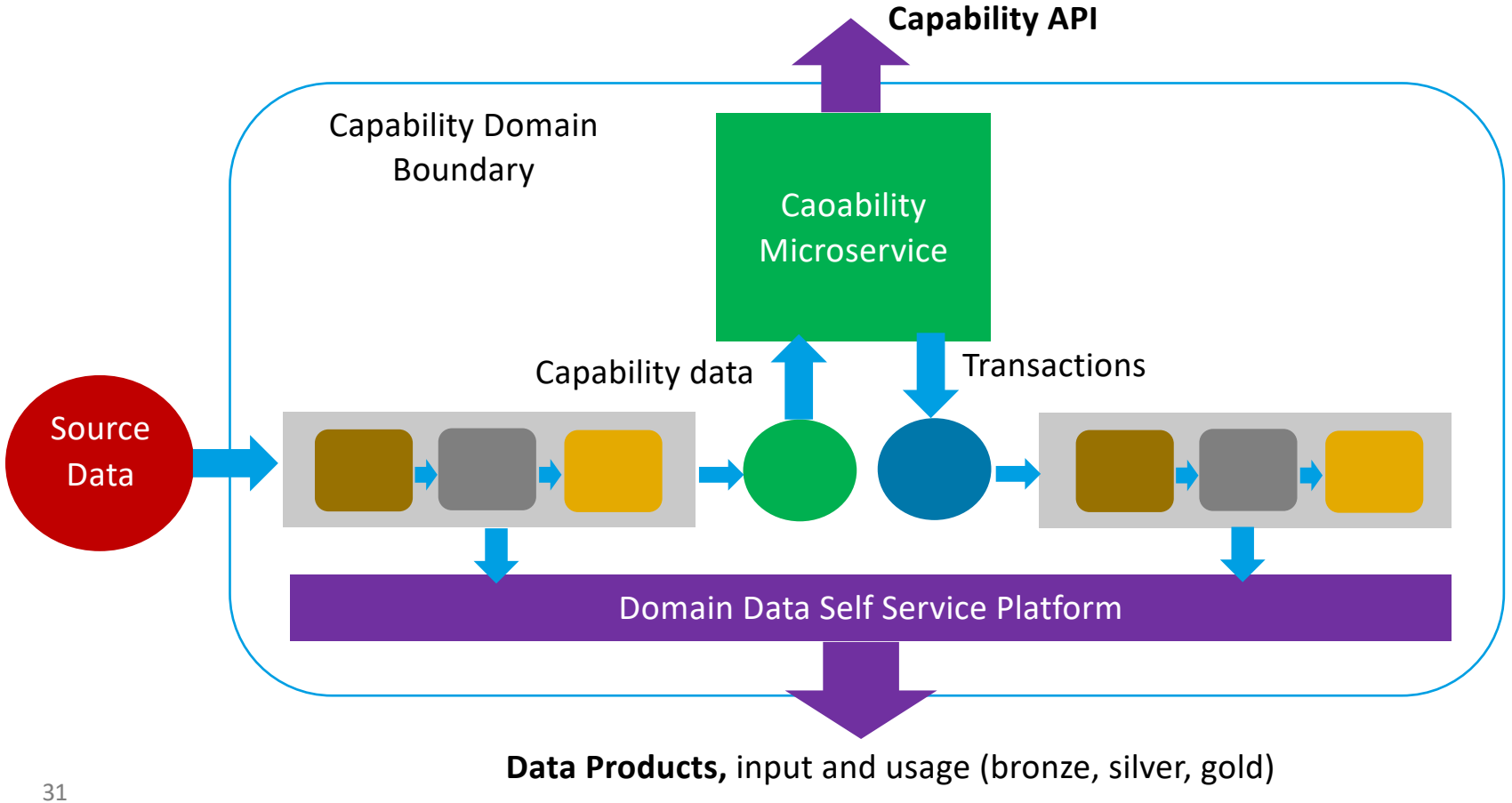
Transactional Data – Microservice Events - Usage Pipelines

The capability teams process the events produced by the microservice transactions



And so on …….

Self-Service Platform – Other teams just get the data they want



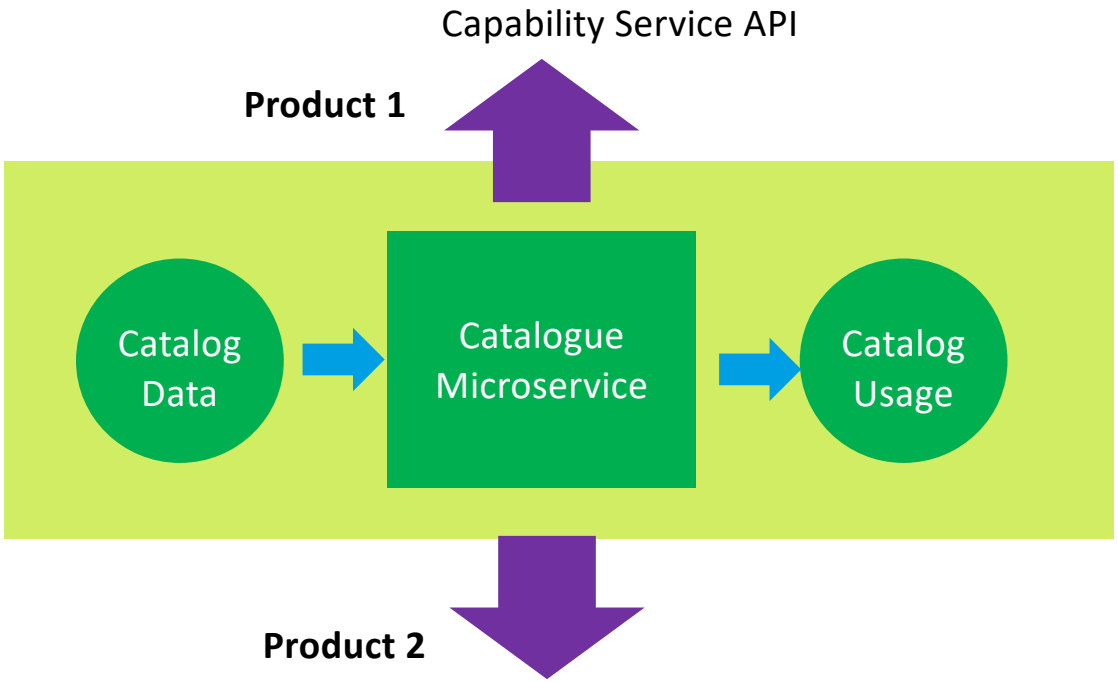
Copy

Consider two Product Types: Domain Capability Services & Data

Consider 1 Microservice,
e.g. Catalog

Consider the team:

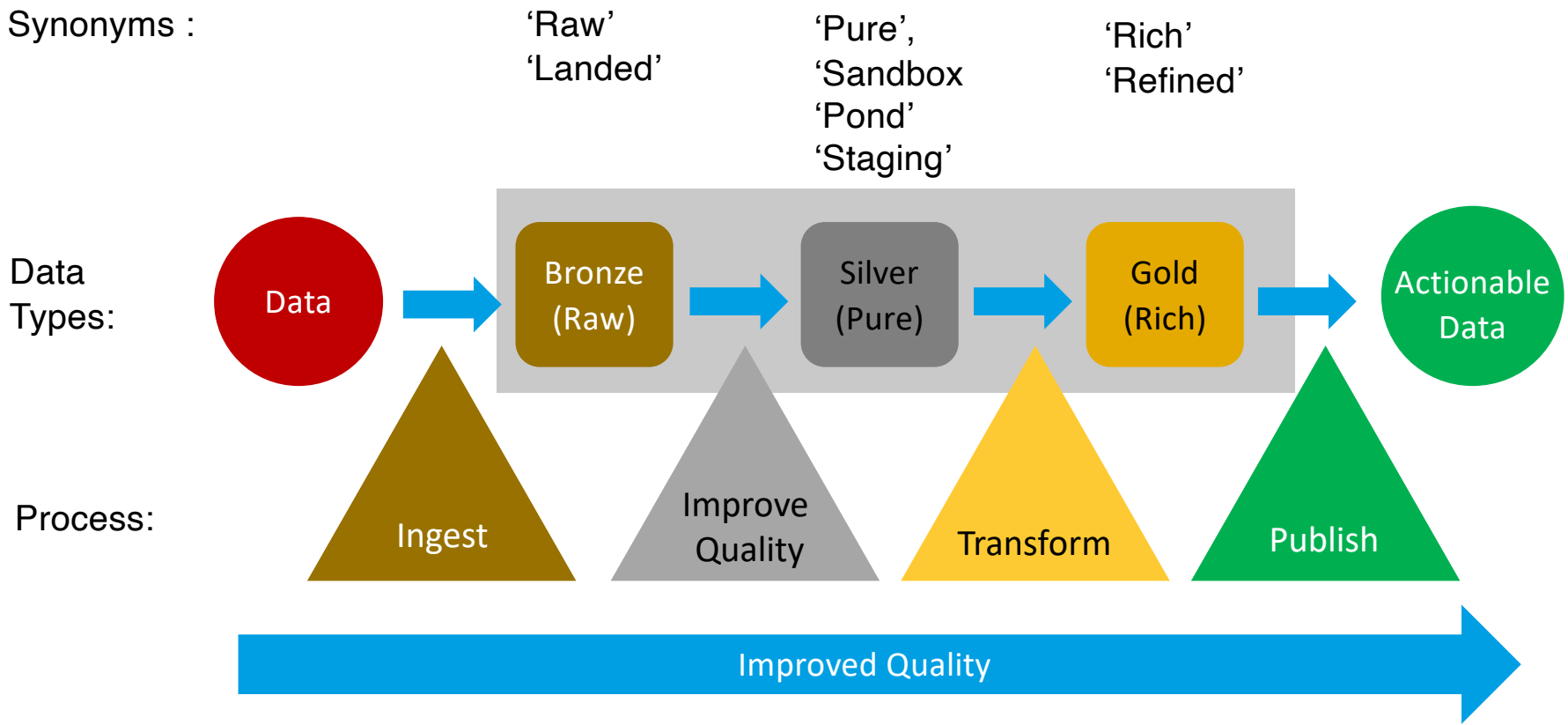
- This team knows its data; they are the organisations experts for their domain
- They processes the raw. data from source.
- They designs the Microservices to use that Data.
- They now the usage transactional data produced.
- They know the quality, meaning, and domain relevance of their data.



Catalog Product Data:

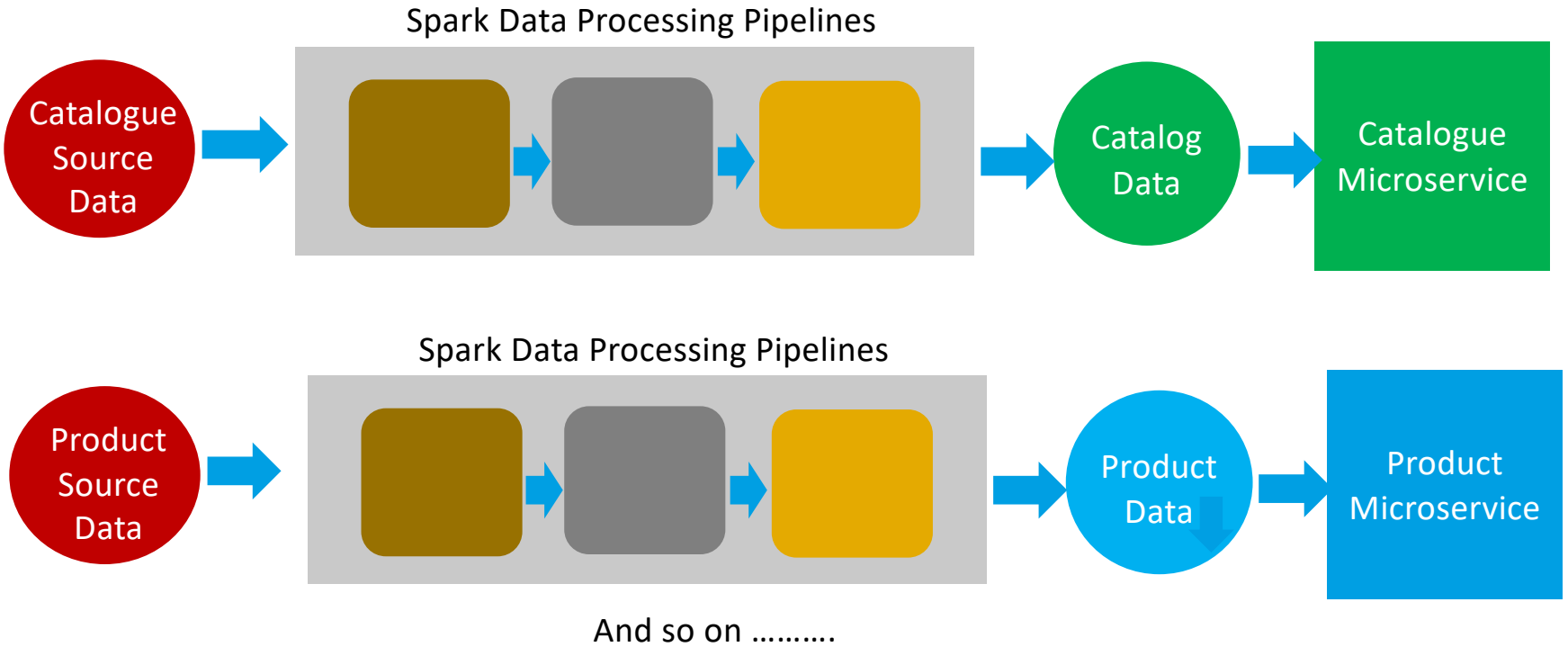
- Capability Data
- Transactional Usage

Multi-Hop - Data Pipeline - Data Quality Types



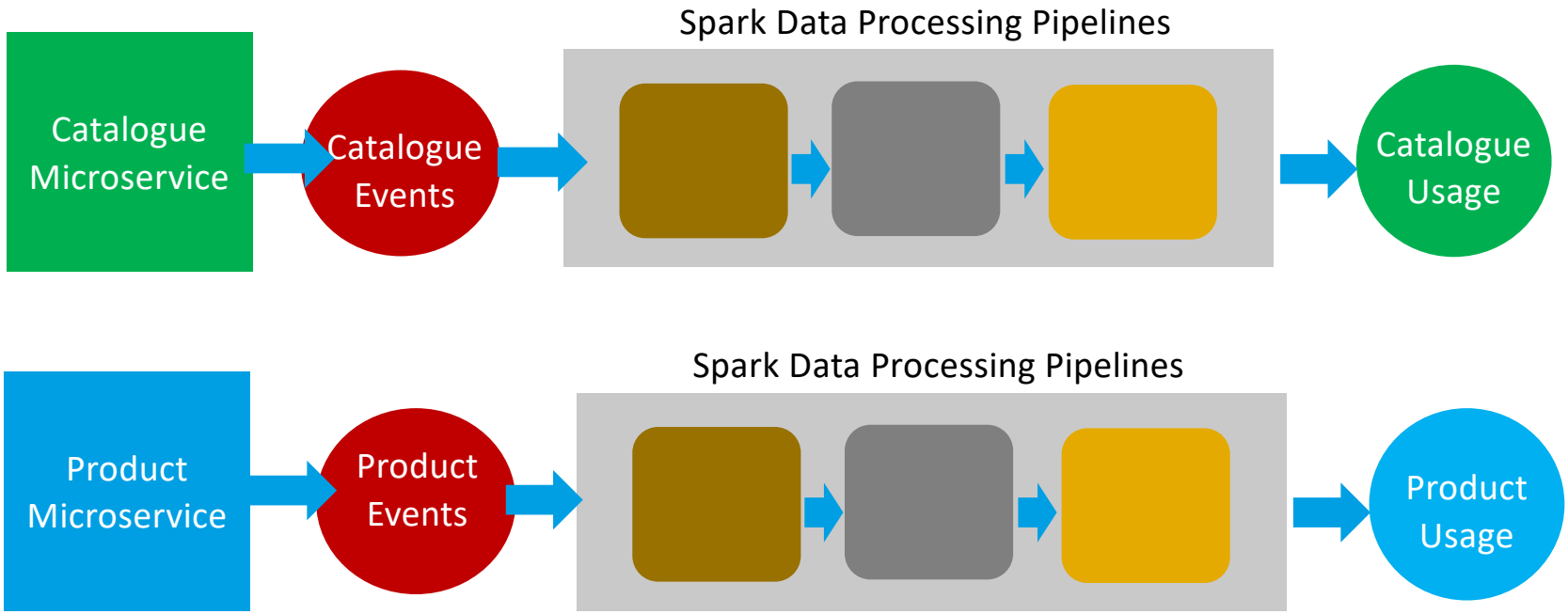
Capability Data – Source to Microservice Pipelines

The capability teams process the source data for the microservice



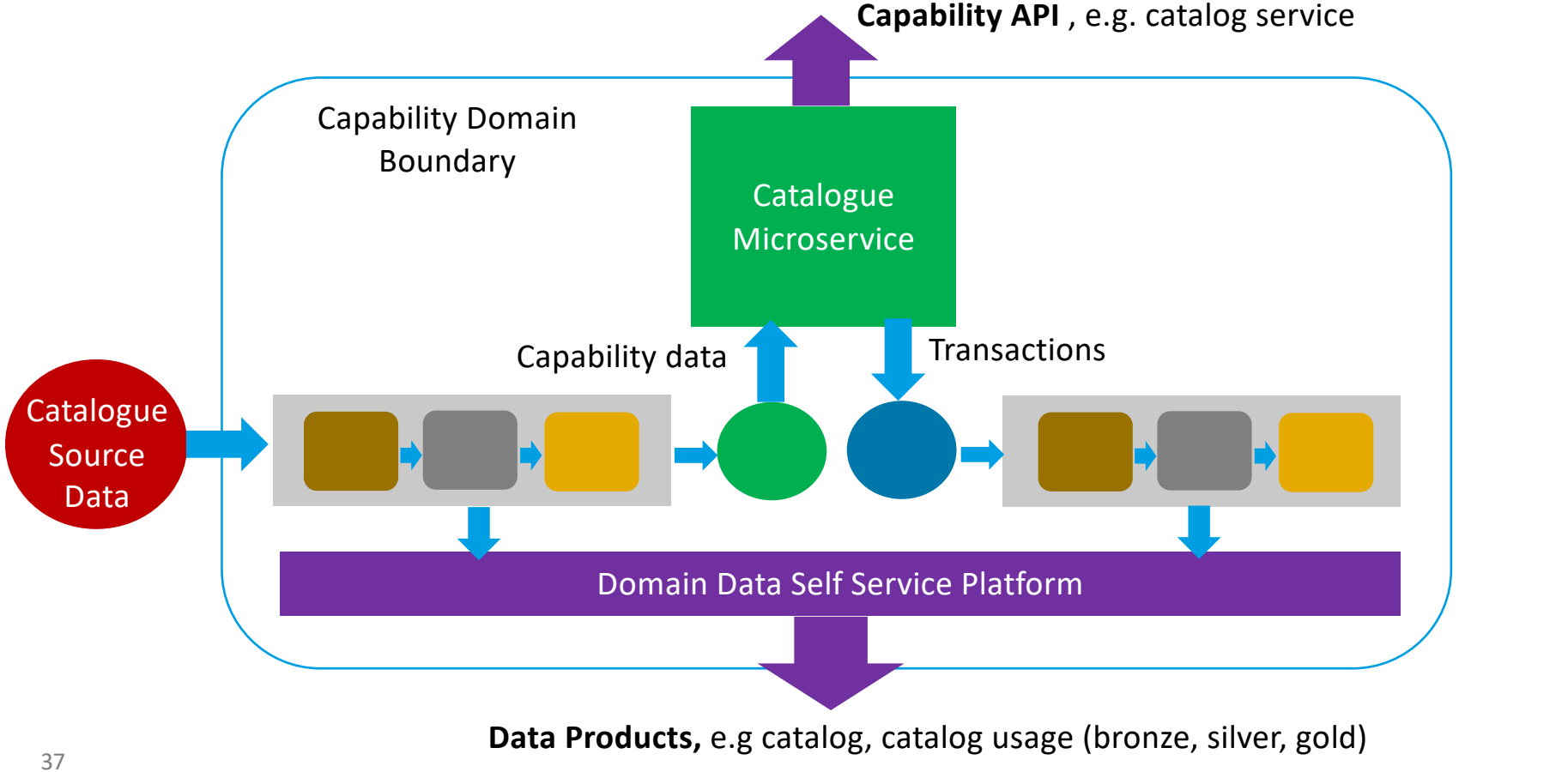
Transactional Data – Microservice Events - Usage Pipelines

The capability teams process the events produced by the microservice transactions



And so on …….

Self-Service Platform – Other teams just get the data they want



Conclusion

The **AntiPatterns** “Reach In - Pull” specific data from Microservice DBs. They bypass the Service itself. They are highly coupled as consumers need to know the schema and database details in depth. Any small change to the DBs will impact the consumers.

The **Patterns** use the ‘Open Host Service’ model, where providers expose a decoupled approach for consumers. . The HTTP Pull Pattern is best when performance is not critical. The Data Pump approaches allow the microservice teams to manage their domain, and the data base can change independently of the consumers. The Event-Based and Data-Lake Push models provide timeliness of data.

The type of relationships between teams (bounded contexts) has to be agreed and made explicit. Parties that work closely together can form partnerships, whereas others may need to protect their domains. Microservice Architectures aim to keep boundaries clear and protected.

Impacting the principles of microservices means they start lose their value. The worst impact is by exposing their internals. Data Mesh aligns with microservices to solves interoperability issues, where Data becomes a Product, and is shared by self service platforms.

Appendix
Relationships

Relationships between Bounded Contexts.

Different teams can have different relationships, including:

- Coupled:
  - Partnership – negotiation and planning are done together, both succeed or fail together
  - Customer/Supplier – one drives the other but they work together
  - Conformist – do whatever other demands
- Decoupled:
  - ACL - Anti Corruption Layer – consumers protects their domain
  - Shared Kernel – both share a subset of the model (partially coupled)
  - Open Host – supplier protects their domain, expose an API/Contract for others to use, encapsulation.
  - Published Language – expose a well known common language (API/Contract)
  - Separate Ways – the two teams just do not integrate.