Scalable Policy-as-Code Decision Points for Data Products

Matteo Brambilla, Pierluigi Plebani
Dipartimento di Elettronica Informazione e Bioingegneria
Politecnico di Milano
Milan, Italy
matteo15.brambilla@mail.polimi.it, pierluigi.plebani@polimi.it

***Abstract*—Data products are emerging as architectural elements of data mesh with the aim of improving data management within organizations. Among the main aspects in the design of a data product is the need to define the policies that regulate its access. These policies concern not only security aspects but, more generally, compliance with regulations (e.g., GDPR, HIPAA) or organizational regulations.**

Taking advantage of similarities with microservice-based solutions, policy-as-code is also used in the data mesh to regulate access to data products. Following the proposed models for the service mesh which regulates a microservice-based environment, the performance of the Policy Decision Point (PDP) becomes crucial to ensure efficient access to data managed by data products. Since this may require a replication of PDPs, in order to achieve true scalability, each of them will have to take into account only the policies associated with the data products under their responsibilities.

This work provides a solution to automate the replication of PDPs and the distribution of policies to be controlled according to an approach that balances policy consistency and system scalability. The work has been validated through an extension of the Kubernetes orchestrator that, based on the information collected from the monitoring system, it is able to define which is the right number of PDPs and which data product, and related policies, should be assigned to each of them. Notably, the policy management is based on a Open Policy Agent (OPA) implementation.

***Index Terms*—policy-as-code, scalability, data mesh, authorization, policy, OPA, Kubernetes**

I. Introduction

In recent times, data mesh has emerged as a prominent business practice for enhanced data management. A fundamental aspect is the data product, an architectural element providing access to data under the data owner's authority [1].

Consequently, the data owner must clearly communicate access policies for their data products. These policies vary significantly and cover traditional security, resource control, and compliance with general (e.g., GDPR), sectoral (e.g., HIPAA), or company-specific regulations.

Given the similarity between data products and microservices [2], policy-as-code is adopted for data mesh implementation [3], akin to service mesh. This involves Policy Decision Points (PDPs) evaluating access requests and Policy Enforcement Points (PEPs) acting on PDP decisions to allow or block requests [4].

While research has extensively studied access policy definition and enforcement, data mesh's decentralized nature raises concerns about PDP decentralization's impact on policy coherence and scalability [5]. A centralized PDP simplifies coherence analysis, whereas a fully decentralized PDP enhances scalability by avoiding bottlenecks.

This work aims to develop an approach for an optimal balance between policy coherence and scalability for accessing data products in a cloud environment. It decouples policy definition from infrastructure management. Policy definers can assess consistency as if with a single centralized PDP. However, policy deployment and assignment to appropriate PDPs are managed by a component that functionally distributes them, ensuring data products are correctly equipped. The approach was validated using Open Policy Agent (OPA) [1] on Kubernetes clusters extended with the defined policy distribution logic, demonstrating positive performance effects.

The rest of the paper is organized as follows. Section II discusses related work on policy-as-code for data products. Section III provides and overview of the approach while Section IV details the proposed architectural elements. The implementation details and validation results are discussed in Section VI. Section VII concludes the work, outlining future directions.

II. Related Work

A. Data product

The data product concept, while not new, gained prominence with the data mesh [1], a decentralized socio-technical approach for efficient data management, access, and sharing. Introduced by Dehghani [6], data mesh distributes data management to domain-specific teams overseeing the entire data lifecycle. In this paradigm, a data product is a self-contained, reusable unit integrating data (content and metadata) and code for access, transformation, or processing.

The data product concept is also emerging as a relevant concept in the design of EU data spaces [7], defined as a provider-packaged data collection including metadata, licenses, usage terms, and governance elements [8].

Technically, data product architectures share analogies with microservices [3]. Data product management practices (e.g., domain importance, clear artifact responsibility, composability) mirror those of microservices, which can also serve as

[1] https://www.openpolicyagent.org

a foundational implementation element. This work assumes authorization mechanisms for microservices are applicable to data products due to this close relationship.

B. Policy-as-code

Similar to service mesh, adopting policy-as-code is beneficial for data mesh [9] to implement authorization mechanisms.

For data products, such mechanisms must enforce policies between data providers and users, covering aspects like security (access rights), performance (usage limits), and storage (data location) [10].

Policy-as-Code expresses policies as interpretable code. It has been used for security policy management [11]. Concurrently, [5] notes the need to balance scalability (requiring decentralized policy management) with inter-policy coherence. Focusing on scalability, [12] proposes scaling by replicating PEPs, while [13] also decentralizes PDPs using caching.

III. Overview of the approach

Policy-based authorization for data products, often implemented as microservices, relies on the synergy between a Policy Decision Point (PDP) and a Policy Enforcement Point (PEP). Consumer requests are typically routed via a gateway to the PEP, which consults the PDP. The PDP, using an Attribute-Based Access Control (ABAC) model [14], decides based on its policies, and the PEP enforces this decision.

A. Centralized PDP

A centralized strategy (Figure 1a) routes all requests through a single gateway (acting as the sole PEP) to one PDP. This simplifies management and resource use but can become a scalability bottleneck. Experiments with performant systems like Open Policy Agent (OPA), particularly with "early exit" and "arrays" policies, demonstrate[2] a linear increase in evaluation time and memory usage as the number of policies grows (Figure 2).

This is because diverse conditions in policies hinder effective indexing, forcing the engine to scan most of them. This highlights the criticality of minimal policy bundles at each evaluation point. Furthermore, policies involving data manipulation (e.g., database lookups, benchmarked using data arrays) also show linear performance degradation (Figure 3). This suggests the need to partition data and distribute policies, for instance, on a per-tenant basis, underscoring the benefits of an automated system for custom policy distribution.

B. Decentralized PDP

To address scalability, a decentralized "sidecar" model (Figure 1b) deploys a PDP with a limited, service-specific policy set alongside each microservice. This minimizes latency. However, it can lead to inefficient resource use and increased complexity in managing distributed services, especially when different data products share policies, requiring meticulous alignment across PDPs. The choice between centralized and decentralized approaches is typically a design-time decision. Unforeseen growth in data products can strain either system, increasing the number of authorization instances and computational resources, even with the more scalable sidecar model. Accurately predicting request volumes and policy loads at design time is difficult. Thus, a separation between policy definition and infrastructure management is crucial, making policy distribution transparent to those defining the rules.

[2]https://github.com/bramba2000/opa-benchmark/blob/main/Results.md

C. Hybrid approach

This work proposes a hybrid approach (Figure 1c) to balance the trade-offs of centralized and decentralized models, aiming for optimal performance with minimal complex orchestration or human intervention. The system dynamically deploys an adequate number of PDPs based on the data products, their associated policies, and the current workload. Each PDP is assigned a subset of data products and configured only with the policies relevant to them. This approach reduces resource consumption compared to a fully decentralized solution and limits the scope of policies evaluated by each PDP, enhancing overall efficiency.

IV. Proposed architecture

The proposed architecture to achieve the aforementioned goal consists of the following three main components organized as reported in Figure 4:

- Policy registry: it contains the set of all the policies defined for the managed data products.
- Dependency manager: it holds a map of the relationship between data products and policies useful to understand how the policies can be distributed among the PDPs.
- Scaling controller: it is responsible to identify the right configuration of PDP and to enact the scalability strategy.

A. Policy Registry

The *policy registry* serves as the central repository for tracking a policy reference. It may contain the actual code for a policy (in the case of OPA, the policy is defined using Rego), or it may provide a reference to where to obtain it; examples of such references are URLs to Git repositories, storage buckets, or any other references to a downloadable text document. Choosing the supported formats and obtaining the policy from the provided reference is an implementation detail.

The primary functionality of the policy registry is to store the information required for a policy. It can check for policy duplication by looking at published policies and finding instances where they have been copied by accident. This can happen if a repository has policies with different file names that have the same result or if a composite policy is pushed when the basic one is already there. Duplication detection depends on the policy language chosen, availability of tools, and other solution-specific languages.

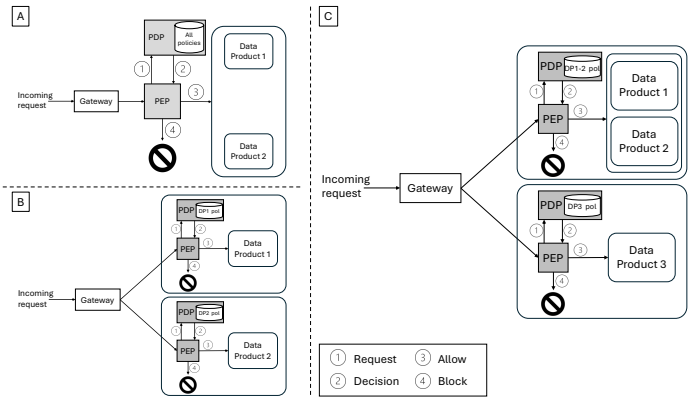

Fig. 1. PDP-PEP deployments: A - centralized; B - decentralized; C - hybrid

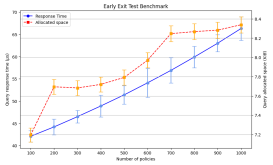

Fig. 2. Early exit test case: response time and allocated space for query

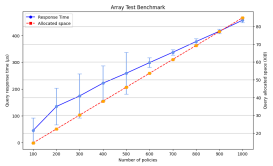

Fig. 3. Array test case: response time and allocated space for query

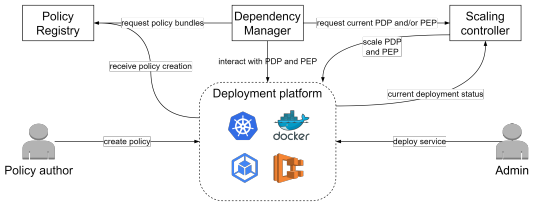

Fig. 4. Scalable PDP architecture

B. Dependency manager

The *dependency manager* is responsible for identifying and managing the dependencies between services and policies. For building and keeping an accurate dependency model, it uses the information from the registry component and extra metadata from the deployment platform.

To determine the optimal deployment strategy for services and policies, it is essential to analyze the relationships between these components. Understanding their dependencies provides insights into how services interact and how policies are applied, facilitating efficient deployment and management.

One of the earliest techniques for managing service relationships was proposed by Hasselmeyer in [15], where service dependencies were modeled as a directed graph. This work adopts a similar approach while extending the model to include policies as first-class components within the dependency graph.

Building upon the descriptions provided in the previous section and the rigorous dependency analysis outlined in [15], this work focuses on three primary types of dependencies:

- **Data Product-to-Data Product**: Dependencies between data products that interact directly.
- **Data Product-to-Policy**: Dependencies between a data product and the policies applied to it.
- **Policy-to-Policy**: Dependencies where one policy's outcome depends on the evaluation of another.

It is worth noting that the identification of the relationships now expressed assumes that it is also possible to evaluate the equivalence between policies. In fact, starting from the assumption that each data owner defines the policies for their data product, it could happen that rules that are not identical, but still equivalent, are defined for different data products. It is not the objective of this work to provide a mechanism for the evaluation of equivalence, leaving this task to existing works in the literature,

Further analysis can be derived from the work of [16], which introduced a graph-based structure to represent microservice dependencies and provided tools and techniques for extracting valuable insights. The authors demonstrated that understanding relationships within a microservice architecture through graph-based analysis enhances development, testing, and maintenance. Exploiting the similarity with data products, the same approach is adopted in this work.

The model proposed in [16] formalized dependencies as a directed graph referred to as the *Service Dependency Graph* (SDG). This graph forms the foundation for analyzing microservice relationships and detecting potential issues, such as cyclic dependencies or misconfigurations. Beyond [16], directed graphs—or, more specifically, Directed Acyclic Graphs (DAGs)—are widely used in various domains, including task scheduling [17].

An example of a resulting dependency graph applied to the

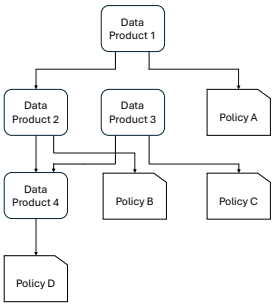

Fig. 5. Example of a dependency graph with services and policies

data mesh domain is reported in Figure 5.

The aforementioned dependency model enables the construction of a dependency graph represented as a directed graph. Both data products and policies can theoretically have circular dependencies, as noted in [16], which also proposed techniques for their detection. However, circular dependencies often lead to faulty or unresolvable deployments, as they create ambiguities that are difficult to address. Consequently, avoiding circular dependencies is a best practice in real-world applications.

For this reason, this work assumes that no circular dependencies exist at any point in the deployment. This assumption simplifies the dependency model, allowing the use of a DAG. A well-implemented DAG not only represents dependencies clearly but also validates the acyclic property during insertion, ensuring that cycles are identified and addressed promptly—either through error reporting or by rejecting the addition of conflicting components. The construction of the dependency DAG is an iterative process that evolves as new policies and data products are deployed. Each new entity is represented as a node in the graph, and its dependencies are registered as directed edges. This structure is particularly effective for understanding the relationships between components, enabling comprehensive reasoning about the policies and data products required for proper operation.

Leveraging the dependency DAG, it becomes possible to identify the *minimal deployment set* required for a specific data product. Starting from a target node and performing a depth-first traversal, the graph explores all connected nodes until no further dependencies are found. This traversal produces a list of required policies along with the associated set of data products. The resulting list of policies constitutes the *minimal policy bundle* necessary to support the targeted data product.

Considering the example situation depicted in Figure 5, the minimal deployment policy set for correctly handling *Data Product 1* is composed of *Policy A*, *Policy C* and *Policy D*. Yet, the minimal set of data products required by *Data product A* is composed of *Data product 3* and *Data Product 4*.

The minimal policy bundle is not only critical for resource-efficient deployment but also serves as a foundation for policy optimization and deduplication strategies. By ensuring that only the required policies are deployed alongside the target data product, it is possible to reduce redundancy, enhance system performance, and simplify the management of security configurations in a distributed environment.

The dependency manager plays a critical role in informing the controller about optimal deployment strategies and scaling decisions. This component may be integrated into a monolithic architecture or distributed across multiple data product to improve fault tolerance and scalability.

C. Scaling controller

The *scaling controller* is the active component of the scalability strategy, responsible for continuously monitoring system metrics and triggering scaling actions when predefined conditions are met. The specific scaling strategies will be discussed in Section V. Anyway, the fundamental responsibility of this component is to act as an always-active monitoring and control unit.

The term ”controller” aligns with its conventional usage in deployment strategies: a component that maintains a desired state for the system, actively monitors the current state, and initiates corrective actions when a discrepancy is detected. The controller ensures that the actual system state converges toward the desired state by dynamically adjusting resource allocations.

The controller can leverage platform-specific metrics or directly collect data from services and observation points using monitoring tools. While the details of the metrics are outside the scope of this document, the controller must be capable of interpreting the provided metrics and deriving meaningful scaling decisions. This procedure includes recognizing when a metric surpasses a predefined threshold, indicating the need for scaling up or down. The controller can use either heuristics or algorithms to make the best scaling decisions based on the type of work being done and the limits of the system.

Aside from these scaling-specific components, the underlying architecture also consists of domain-specific components such as application containers, custom service deployments, and policy infrastructure. Depending on the deployment platform, various deployment techniques may be utilized, including service containers running on Docker or Kubernetes, as well as virtual machines hosting standalone server applications that operate within or outside the platform.

The policy evaluation infrastructure is highly implementation-specific but typically consists of at least a PDP, which is a policy evaluation engine, alongside mechanisms for publishing and distributing policies to the evaluation engine. In the context of this work, the OPA infrastructure is employed, utilizing the OPA engine as the primary evaluation component.

The scaling controller component operates across both domain-specific and generic components, collecting usage metrics and dynamically replicating one or more components to achieve the desired policy and performance objectives. By intelligently managing service and policy scaling, the proposed architecture enhances efficiency, resilience, and adaptability in distributed environments.

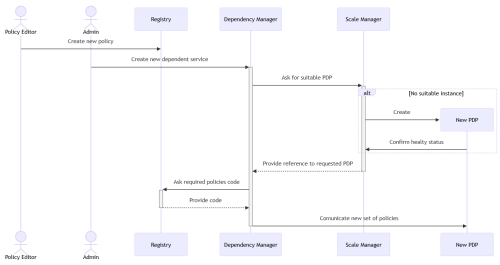

Fig. 6. Sequence diagram of initial deployment

V. Scaling Strategy

Given the importance, this section discusses some details related to the scaling mechanism implemented by the scaling controller.

A. Deployment Process

Initially, the scaling controller is in an empty state. Upon deployment of the first data product and its policies, both the scaling controller and dependency manager become active. The controller deploys a single PEP for all traffic, resembling a centralized model. The associated PDP contains only policies for this initial data product, optimizing decision times.

As new data products are added, the dependency manager updates its internal representation. The scaling controller modifies the single PDP's policy set via a rolling update, ensuring no downtime. A key principle is maintaining a *minimal policy bundle*: the PDP only evaluates policies essential for its assigned data products, excluding unnecessary ones to optimize performance and resource use.

B. Handling High Traffic Scenarios

When traffic exceeds predefined thresholds, the system adapts by analyzing metrics and dependencies to identify overloaded PDPs, triggering a *scaling deployment plan*. This plan determines the best strategy, such as separating independent policies into different PDPs or partitioning policies.

The scaling controller executes:

- Deployment of a new PDP instance with the policy set for the newly scaled data product.
- Updating the existing PDP by removing policies now managed by the new PDP, preserving the minimal policy bundle.
- If enabled, scaling and replicating the overloaded data product.

This dynamic scaling replicates only necessary components, maintaining performance without excessive resource use. When traffic normalizes, the controller can revert to the previous configuration. For multiple overloaded data products, the system provisions parallel deployments tailored to specific demands. This adaptive approach creates a scalable, efficient, and resilient policy enforcement framework. The behavior is described in Algorithm 1.

Algorithm 1 Scaling Controller - High Traffic Handling

Require: Traffic metrics, Service metrics, Dependency graph, Scaling configuration, Current PDP deployment
Ensure: Updated PDP deployment, Scaled services

```
if Traffic exceeds configured threshold then
    Analyze service metrics and dependency relationships
    Identify overloaded PDPs and services
    Generate a scaling deployment plan based on configuration:
    if Scaling strategy is policy separation then
        for each overloaded PDP do
            Identify policy sets requiring separation
            Deploy new PDP instance with separated policy set
            Update existing PDP to remove separated policies
        end for
    else if Scaling strategy is service replication then
        for each overloaded service do
            if Service replication enabled then
                Scale and replicate the service
            end if
        end for
    end if
    Apply the scaling deployment plan
    Update current PDP deployment state
end if
if Traffic returns to normal then
    Revert to previous deployment configuration
    Consolidate resources
    Update current PDP deployment state
end if
if Multiple services exceed thresholds then
    Generate and apply parallel deployment plans
    Provision additional parallel deployments
    Tailor deployments to specific workload demands
end if
return Updated PDP deployment, Scaled services
```

C. Scaling controller configuration

The scaling controller supports various configurable techniques for optimal scaling deployment plans.

One performance-optimized strategy scales a data product and its entire dependency pipeline (identified by the Dependency Manager), potentially replicating the busiest data product. Policies for all pipeline components are included in the new bundle. This offers optimal performance by dedicating resources to frequently accessed components but risks inefficiency if unneeded products are scaled or policy evaluation isn't the bottleneck.

A conventional strategy selectively replicates only the overloaded data product. Alternatively, if metrics indicate policy evaluation is the bottleneck, only the PDP and PEP for affected data products are scaled, automating traditional scaling techniques.

A domain-specific technique, useful in multi-tenant environments, tags policies and data products with group identifiers. The controller treats each group independently, deploying distinct PDP/PEP instances and applying group-specific metrics

and strategies. This offers fine-grained control, tenant-aware scaling, and optimized resource allocation.

These configured techniques and metric thresholds define the system's desired state, guiding the controller's operations. By comparing real-time conditions to this desired state, the controller ensures optimal performance and adapts to workload changes.

VI. Implementation Details

The implementation architecture is based on the *Kubernetes Operator*[3], a cluster-state management system that reconciles the current state of Kubernetes resources with the desired one. The controller observes resource states, identifies discrepancies, and acts to reconcile them. [18] demonstrates that applying a control system to cloud-native applications efficiently manages scaling for stateful applications like OPA PDP.

Operators create and install custom resource definitions (CRDs) in the Kubernetes API server, enabling interaction with the cluster to get the current state of custom resource (CR) instances. Each resource can represent Kubernetes components or store information, such as Rego file content. Resources may be associated with a dedicated controller that manages their status by registering ownership and receiving updates on changes. The operator also utilizes Prometheus[4] for monitoring.

Changes to CR instances by a user or another controller initiate a reconciliation process to return the resource to its expected state. The operator also interacts with Prometheus to detect and react to high-load situations, as depicted in Figure 7.

The operators define three custom resources:

- *policy*: Stores Rego policy content. The `policy` CRD (`policies.opas.polimi.it/v1alpha1`) allows loading policy files via text or OCI image. The `opaengine` controller manages policy download and provision to OPA.
- *dependency*: Informs the controller of a service's policy usage. The `dependency` CRD (`dependencies.opas.polimi.it/v1alpha1`) represents service-policy or policy-policy dependencies, addressing two of the three types described in Section III
- *opa engine*: Represents a PDP service and deployment, largely managed internally. The `opaengine` CRD (`opaengines.opas.polimi.it/v1alpha1`) signifies an OPA engine instance, owning a deployment and Kubernetes service. Its specification includes an expected policy list, configured by the dependency service, which the `opaengine` reconciler keeps synchronized with deployed policies.

Each resource has custom Kubernetes RBAC permissions restricting actions that could disrupt operator flow. For example, a user can delete an OPA engine resource but not its underlying container, which would break the deployment.

[3]The code is available at https://github.com/bramba2000/opa-scaler

[4]https://prometheus.io/

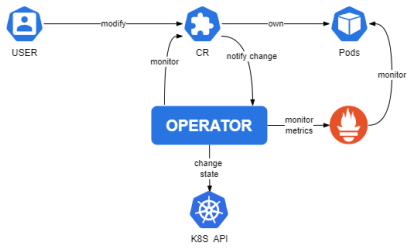

Fig. 7. Overview of the proposed implementation of Kubernetes operator

This implementation was rigorously tested for controller behavior and validation rules, and qualitatively validated for performance. All test scenarios showed enhanced response times and uninterrupted authorization service, attributed to the controller's rolling update strategy.

VII. Concluding Remarks

This paper presented an approach for efficient and scalable management of policies associated with data products in a mesh data environment. The proposed solution is based on the possibility of replicating PDPs with the increase in the number of policies or the number of policy evaluation requests, the latter aspect dependent on the increase in the number of requests to access a data product. Through an extension of a Kubernetes operator, the platform after acquiring monitoring information can decide to deploy new replicas of a PDP and associate them with a subset of the data products present. In this way we have a partition of the policies to evaluate. By leveraging the concept of minimal policy bundled, the system is able to ensure that this partition ensures consistency between policies.

Possible extensions of this work are multiple. If at the moment the policies considered are generic policies without any reference to particular aspects (e.g., security, performance), it is possible to study how the presence of policies that refer to different domains can be exploited for a more efficient partition. Furthermore, since at the moment the initial point of work involves a deployment with no data product, it is plausible to think of situations in which there is a first deployment with an already defined number of data product. Finally, other possible developments may concern the definition of scaling strategies that aim to minimize not only the response time but also the resources used. This is to be able to hypothesize a use in a mixed cloud/edge system that sees in the edge the presence of devices with constraints on the availability of computational resources.

Acknowledgment

This work has been funded by the European Union (TEADAL, 101070186). Views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Z. Dehghani, *Data Mesh*. O'Reilly Media, Inc, Mar 2022.

[2] V. K. Butte and S. Butte, "Enterprise data strategy: A decentralized data mesh approach," in *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*, 2022, pp. 62–66.

[3] A. Goedegebuure, I. Kumara, S. Driessen, W.-J. Van Den Heuvel, G. Monsieur, D. A. Tamburri, and D. D. Nucci, "Data mesh: a systematic gray literature review," *ACM Computing Surveys*, vol. 57, no. 1, pp. 1–36, 2024.

[4] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," Tech. Rep., 2001.

[5] A. Wider, S. Verma, and A. Akhtar, "Decentralized data governance as part of a data mesh platform: Concepts and approaches," in *2023 IEEE International Conference on Web Services (ICWS)*, 2023, pp. 746–754.

[6] Z. Dehghani, "How to move beyond a monolithic data lake to a distributed data mesh," May 2019. [Online]. Available: https://martinfowler.com/articles/data-monolith-to-mesh.html

[7] Data Spaces Support Centre (DSSC), "Core concepts," https://dssc.eu/space/Glossary/176554052/2.+Core+Concepts.

[8] CEN-CELEC, "Trusted data transaction," https://www.cencenelec.eu/media/CEN-CENELEC/CWAs/RI/2024/cwa18125_2024.pdf, 2024.

[9] R. Chandramouli, "Implementation of devsecops for a microservices-based application with service mesh," March 2022. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-204C

[10] J. Yang, Y. Tang, and A. Beheshti, *Design Methodology for Service-Based Data Product Sharing and Trading*. Cham: Springer International Publishing, 2021, pp. 221–235. [Online]. Available: https://doi.org/10.1007/978-3-030-73203-5_17

[11] A. Wider, K. Jarmul, and A. Akhtar, "Towards automating federated data governance," in *2024 IEEE International Conference on Web Services (ICWS)*, 2024, pp. 10–19.

[12] Q. Yaseen, Y. Jararweh, B. Panda, and Q. Althebyan, "An insider threat aware access control for cloud relational databases," *Cluster Computing*, vol. 20, pp. 2669–2685, 2017.

[13] G. Deep, J. Sidhu, and R. Mohana, "Distributed pep–pdp architecture for cloud databases," *Wireless Personal Communications*, vol. 128, no. 3, pp. 1733–1761, 2023.

[14] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[15] P. Hasselmeyer, "Managing dynamic service dependencies," in *12th International Workshop on Distributed Systems: Operations & Management*, 2001. [Online]. Available: http://proceedings.utwente.nl/13/

[16] S.-P. Ma, C.-Y. Fan, Y. Chuang, W.-T. Lee, S.-J. Lee, and N.-L. Hsueh, "Using service dependency graph to analyze and test microservices," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 02, 2018, pp. 81–86.

[17] A. S. of Mechanical Engineering, Ed., *Managing Dependencies for Collaborative Design*, ser. International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, vol. Volume 4: 12th International Conference on Design Theory and Methodology, 09 2000. [Online]. Available: https://doi.org/10.1115/DETC2000/DTM-14552

[18] F. Y. Chemashkin and P. D. Drobintsev, "Kubernetes operators as a control system for cloud-native applications," Tech. rep., Peter the Great St. Petersburg Polytechnic University, Tech. Rep., 2021.