XYZ: dynamic microservice composition for data-exchange systems, lessons learned

Index Terms—Microservices, Distributed Computing, Data Exchange, Self-Adaptivity, MAPE-K

***Abstract*—Data exchange has become increasingly important in modern business and research. Consequently, many initiatives are being developed worldwide to facilitate open data exchange in secure distributed marketplaces. Ideally, each party maintains control over their data and implements access through legal contracts, in the form of programmable policy. Such policy would express where the data exchange takes place, and who has access to the data.**

Inspired by how traditional Operating Systems abstract the complexities of computer architectures into standardized core functions, this research focuses on abstracting different data exchange patterns into a unified set of core data exchange microservices that adhere to agreed-upon data exchange policies. XYZ implements a distributed data exchange platform and recreates real-life data exchange use cases. It is designed to be self-adaptive, utilizing extendable algorithms to generate dynamic microservice compositions and dynamically choose archetype patterns, influenced by policy, user input, or system events.

In our study, we highlight key insights from our experience with a dynamic microservice platform. Employing sidecars for communication abstraction, protocol buffers for strict interface definition, and ephemeral single-use jobs for improved security emerged as pivotal strategies. However, these approaches do introduce a trade-off between operational speed and especially system complexity.

I. Introduction

In public and private sectors alike, the greatest value of data lies in the ability to refine and exchange it [1]. The ability to access larger datasets can significantly improve the performance of machine learning models, while in domains such as medical science, the availability of more patient data holds immense potential. However, for data exchange to be successful, it must align with the desired level of privacy and security of each participating party [1].

To create a platform for data exchange, various initiatives around the world work on establishing 'Digital Data exchange Marketplaces' (DDM). One such initiative is the AMdEX project [2], which aims to develop a DDM prototype that enables participating organizations to maintain control over their data and enforce access and control through policy. One part of this policy will express **where** the data exchange takes place, and **who** has access to the data.

An important concept within DDMs is a set of common data-sharing patterns, known as archetypes [1, 3]. These archetypes determine which party provides data, which party does computations on this data, and where the results of that data-sharing exchange go as agreed upon in the policies.

In XYZ[2] we strive to establish a self-adaptive system capable of seamlessly incorporating policy, archetypes, as well as functional and extra-functional adaptations, without manual intervention. We compose microservices in different configurations to serve different application goals in a distributed system. We use the term *'microservice chains'* to describe various microservice configurations. A chain is a directed acyclic graph of microservices that form a valid data exchange scenario that adheres to policy agreements. These chains need to be executed, for this, we use the generic term 'jobs'. A data exchange request flows through this chain job deployed to the parties set out by the policy. We test these concepts on a real-life case study described in section V. The goal of our work can be summarized in the following research question:

Using different microservice architectural patterns, can we provide dynamically composed microservices, and dynamic archetypes for data-exchange systems?

Through XYZ we made the following contributions:

- C1 Combining data-exchange requests and archetypes into composable microservice chains
- C2 Architecture to bridge the gap from a formal policy evaluation result, to a web environment where jobs are executed.
- C3 Generic method to create single-use, self-cleaning microservice chain jobs
- C4 Generic communication method of transferring data through microservice chains
- C5 Loosely coupled messaging solution, with a specified interface for communication in XYZ
- C6 An open-source, reproducible system that allows experimentation with microservice chains and archetypes in a distributed environment

This paper is organized as follows. section II explains AMdEX, the data-exchange platform that is the base of this research, archetypes, and related data-exchange terminology. section III introduces critical architecture components and draws an abstract design. In section IV we explain in depth the microservice chain, different execution architectures, and self-adaptability using MAPE-K. section V explains the exact use case for which the demo is available. In section VI we reflect on lessons learned. section VII presents related work and we conclude in section VIII.

[1]https://www.theguardian.com/technology/2013/aug/23/tech-giants-data

[2]https://anonymous.4open.science/r/XYZ-BB34

II. Background

The main use case for this research is based on the Amsterdam Data Exchange (AMdEX). AMdEX is introduced below, along with frameworks created by AMdEX that assisted this research. Data-sharing terminologies are also introduced, laying the foundation for the design described in section III.

A. *Amsterdam Data Exchange (AMdEX)*

AMdEX[3] is a collaboration between several public and private organizations. The goal of the consortium is to develop a neutral, generic, independent infrastructure that ensures sovereignty in data exchange initiatives.

Research conducted by the Dutch economic ministry and the European Commission reveals the inadequacy of the current infrastructure for securely sharing data [4, 5]. The situation is worsened by the accumulation and control of vast amounts of data by several major tech companies, leaving consumers with little to no control over their data. The AMdEX project is part of a broader initiative aimed at addressing these challenges. A notable example concerns medical data; although hospitals and patients are willing to share patient data with researchers, the absence of appropriate legal frameworks and infrastructure hinders their ability to do so.

Several proof of concepts (PoCs) revealed significant challenges, such as high technical hurdles and legal complexities faced by data suppliers, scalability issues in the initial AMDEX use cases, constraints on data suppliers primarily to market entities potentially fostering monopolies and loss of data control, as well as substantial legal hurdles associated with cross-border data sharing.

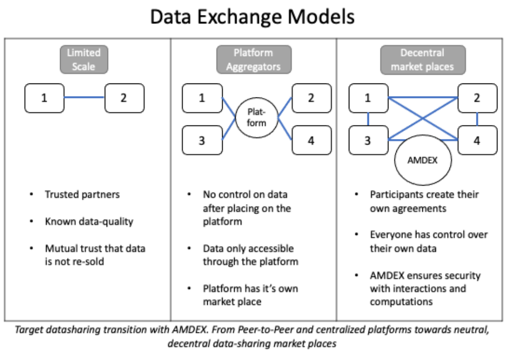
Data Exchange Models
Limited Scale
1
2
Platform Aggregators
1
2
3
4
Plat-form
Decentral market places
1
2
3
4
AMDEX
Trusted partners
Known data-quality
Mutual trust that data is not re-sold
No control on data after placing on the platform
Data only accessible through the platform
Platform has it's own market place
Participants create their own agreements
Everyone has control over their own data
AMDEX ensures security with interactions and computations
Target datasharing transition with AMDEX. From Peer-to-Peer and centralized platforms towards neutral, decentral data-sharing market places

Fig. 1. Data sharing architectures

The third column of Figure 1 shows the target of AMDEX, an *internet of data*, where multiple parties exchange data and algorithms in a secure, open, federated way.

B. *Data-sharing terminology*

Figure 2 illustrates several roles in a dataspace as defined in the AMdEX reference architecture [2]. A **dataspace** is an umbrella term describing several kinds of data exchange collaborations such as a Datatrust, Datacommon and Datamarket. Of these roles, this paper primarily focuses on the Asset Provider, Data Consumer, and Service Provider roles.

[3]https://amdex.eu/

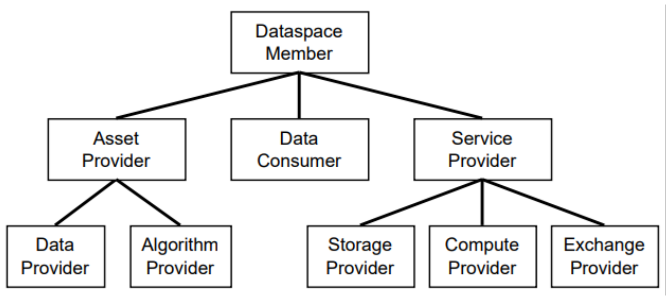

Fig. 2. Taxonomy of dataspace members [2]

Many case studies [1, 3] helped identify a set of typical patterns –**archetypes** in which data can be exchanged. Whether a party is allowed to access data, and which archetypes are chosen to facilitate the data exchange, is encoded in policies.

1) Archetypes: In general, the participating organizations in a data exchange can share two kinds of resources: software and data [1, 3]. For legal requirements and costs, it can also be important **where** the data exchange takes place. And lastly, **who** has agreements to access the data.

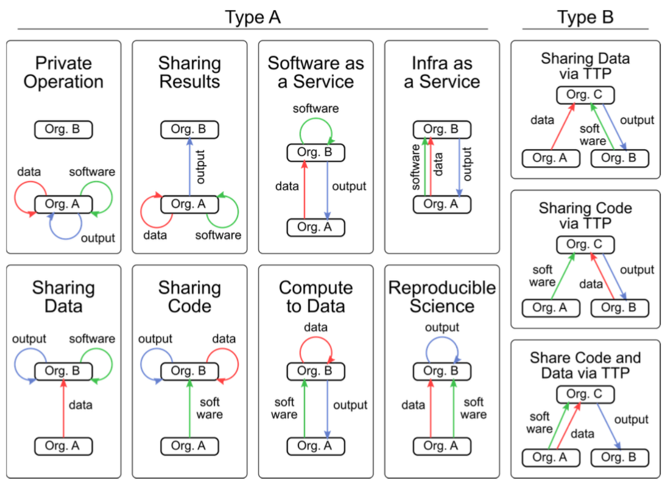

Fig. 3. Data-sharing archetypes [1]

Figure 3 shows an overview of a set of archetypes. The main use-case from this research focuses on the '**Data through a trusted third party** (TTP)' and '**Compute to data**' archetypes. More information about these particular archetypes is provided in section V

2) Policy: In the framework of the AMDEX project, the component tasked with ensuring compliance with policies is designated as a policy enforcer. Ongoing research [6] is focused on developing an automated policy enforcement framework. In this paper, we emulate the functions of the policy enforcer to simulate request validation and establish a foundational source of archetype information.

III. Design of XYZ

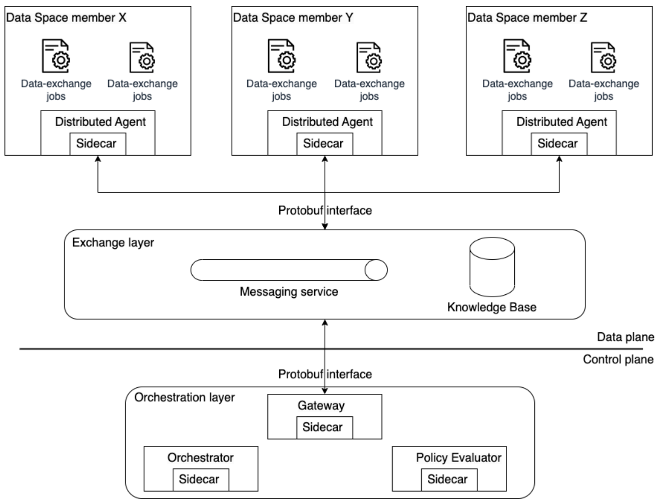

Fig. 4. XYZ abstract design

A data-exchange system is a distributed computing space with a variable amount of actors and stakeholders [2]. In this data-exchange system, we introduce two levels of adaptability: first, dynamic composition of microservices to complete different data-exchange requests; second, dynamic archetype selection to achieve extra-functional goals. Figure 4 shows our design based on the following architectural patterns.:

- A distributed agent-based pattern controlled by a central orchestrator
- A sidecar microservice pattern [7] to abstract communication patterns
- Short-lived single-use jobs to facilitate data-exchange services
- A MAPE-K approach for creating control loops to make choices in Archetypes, using distributed coordination principles [8]
- Google Protobuffers [4] for a language-and platform-agnostic, strictly typed messaging interface
- API gateway pattern to dynamically route request

IV. Implementation

We delve into the architecture underlying two critical processes: the dynamic creation of microservice compositions and the selection of archetypes. Additionally, we examine the underlying platform, communication patterns, interfaces, and gateway and sidecar patterns.

A. Platform

Every node in XYZ runs Kubernetes for executing microservices, chosen for its widespread industry acceptance which makes it a good candidate for running an open-source platform to experiment with microservice chains.

RabbitMQ is used for AMQ-based asynchronous messaging processing, and ETCD [9] is the distributed data store and

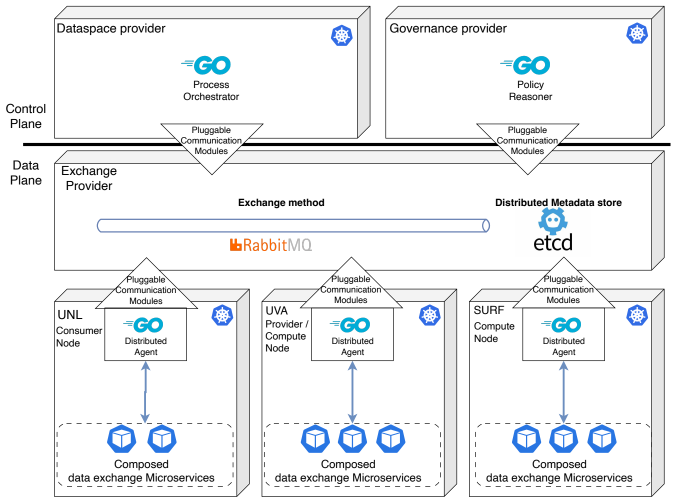

Fig. 5. XYZ implementation architecture

doubles as a mechanism for coordination primitives in the distributed system [8].

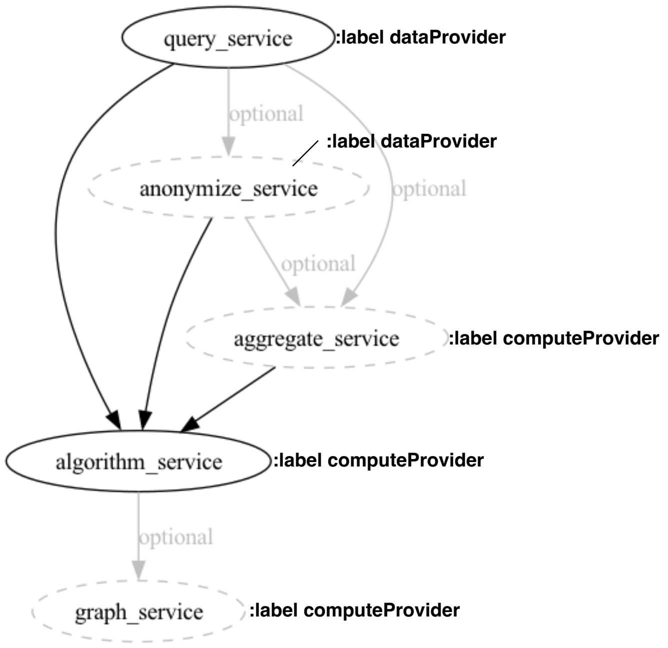

Fig. 6. SQL data request microservice chain

B. Microservice composition - microservice chains

A data exchange microservice chain has a minimum amount of *required* services to process a request. A chain might have *optional* microservices that add extra functionality. Figure 6 shows a valid microservice chain for the SQL data request.

Generating a chain for this request follows the following steps. All microservices are labeled and stored in the central knowledge base. On receiving a request for data exchange, the process orchestrator checks all policy requirements and creates

[4]https://protobuf.dev/overview/

a '*composition request*' for each distributed agent, giving each agent a *role* in the process.

After getting authorization from the orchestrator, an actual data exchange request can be sent. On receiving the request the distributed agents check if there is an available composition request, and now can generate a microservice chain using a topological sorting algorithm based on user input, available jobs, and the allocated role in that job.

C. Microservice chain jobs

The microservice chains are transformed into **single-use, ephemeral jobs**. Ephemeral jobs are created on demand, executed, and subsequently removed. This has advantages in data isolation, and security due to limited exposure and reduced attack surface. However, execution time will be higher due to always needing to spawn new microservices.

D. Job Data exchange - gRPC

The job is created in a single Kubernetes pod (in a Kubernetes job). This pod can contain several microservices to process a data exchange request. We have developed a generic way of using gRPC to pass along data as an argument to a following microservice. This method takes advantage of the following facts:

- gRPC is language agnostic by default
- Data stays in memory, it never writes to disk
- A Kubernetes job can be created with multiple containers, inside a single Pod. Pods communicate on 'localhost' level, or 'interprocess communication' (IPC). This makes the network latency of transferring data from service to service within a Pod negligible
- gRPC supports 'Client streaming RPC', which means that large datasets can be transferred in a stream of messages to preserve internal memory usage. This does require more complex message handling code

E. Sidecar pattern

XYZ makes heavy use of the sidecar pattern [7]. The sidecars within XYZ have two functions. First, to abstract communication logic from the exchange layer. In this implementation a RabbitMQ messaging implementation is used, the sidecar enables swapping this exchange layer for other messaging implementations without affecting the core services.

Secondly the sidecars function as the entry and exit points of a running microservice chain job. Any microservice entering the system will need to comply with this interface. After running a data exchange through the microservice chain, it will end up in the sidecar to route the message to the next node/destination, as seen in Figure 7.

F. Interface

A request can be dynamically processed through an *n* number of microservices. To let microservices handle any type of data, and allow the system to properly route messages we defined the interface in listing 1.

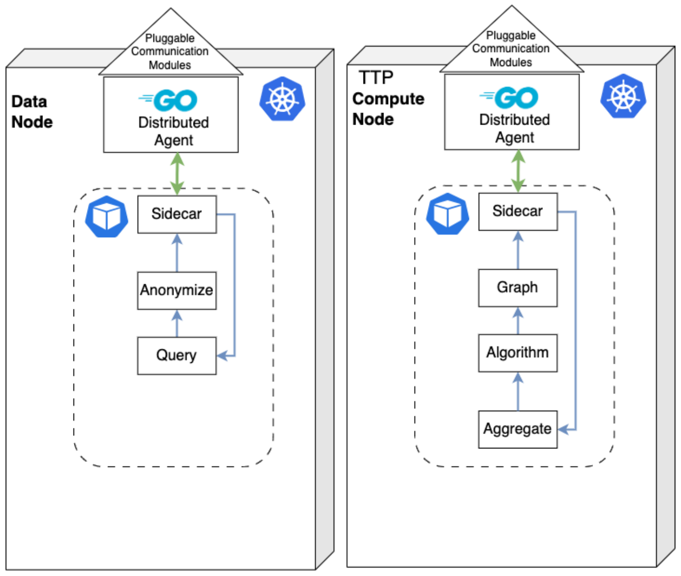

Fig. 7. Job running on two nodes

Listing 1. Microservice Communication

```
message RequestMetadata {
  string correlation_id = 1;
  string destination_queue = 2;
  string job_name = 3;
  string return_address = 4;
  string job_id = 5;
}

message MicroserviceCommunication {
  string type = 1;
  string request_type = 2;
  google.protobuf.Struct data = 3;
  map<string, string> metadata = 4;
  google.protobuf.Any original_request = 5;
  RequestMetadata request_metadata = 6;
  map<string, bytes> traces = 7;
  bytes result = 8;
  repeated string routing_data = 9;
}
```

The data of a request is stored in a generic 'google.protobuf.Struct' type, metadata, the original request, and other specific data can also be stored in this message.

G. Self-adaptability and MAPE-K

MAPE-K [10, 11] is a common pattern to add self-adaptibility and make decisions about the system based on a *knowledge base, etcd* in this case.

We identify a control loop in an interplay between the distributed agents and the orchestrator. The microservice chain enables dynamic microservice composition, and does this based on the information in the knowledge base. The orchestrator can adapt the jobs in this knowledge base based on information in the legal policies, or other environment input. This allows it to change an archetype *per request*, or dynamically add or remove required microservices.

H. Gateway pattern

Due to changing archetype patterns an endpoint or route of a request may change at any time. Furthermore, exposing the endpoints of Data Space members to other members might not be feasible or desirable. Due to his constraints, a central gateway service plays a role in dynamically routing requests based on job information in the central knowledge base.

V. CASE STUDY: UNL

To illustrate the practical implications of this research, a proof-of-concept was developed based on the Universities of the Netherlands (UNL) use case from the AMdEX digital data-exchange project [12], focusing specifically on Scenarios 2 and 3 outlined in the AMdEX use cases [2]. This scenario involves a data analyst analyzing SQL datasets sourced from various universities across the Netherlands. It operates under the assumption that all universities connected to the system share identical data schemas, such as database tables. Furthermore, all data utilized within this use case was synthesized to replicate real-life scenarios accurately.

The scenario necessitates meeting several key requirements. Firstly, it should enable the selection of a different archetype for each request, accommodating changes in contract agreements, user requests for data aggregation requiring third-party involvement, or shifts in data processing to third parties due to nearing capacity at data nodes. XYZ is expected to autonomously determine the appropriate archetype for each request. Additionally, all decisions made must be subject to validation by a policy evaluator. Moreover, the system must ensure programming language agnosticism for broader compatibility and flexibility.

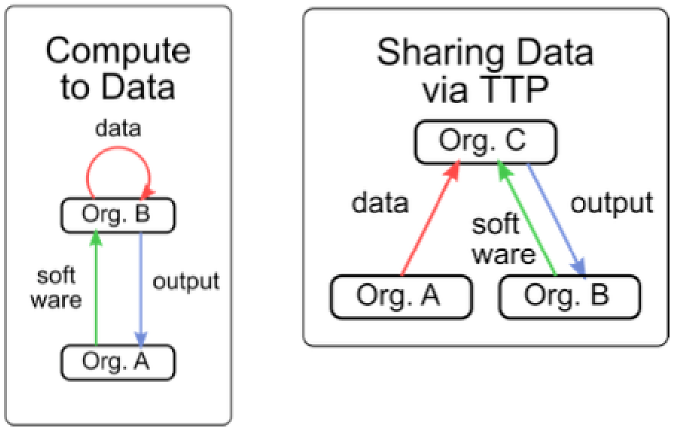

Fig. 8. Archetypes based on work by Shakeri *et al.* [1]

In particular, this use case involves a data analyst investigating wage gaps within the universities included in the system. The data analyst sends out an SQL query and an algorithm, on which XYZ checks the policy, generating a microservice chain, and lets the universities deploy the services, returning the results without requiring the data analyst to interact with any data. This scenario is executed using two archetypes, namely '**ComputeToData**' and '**Data via Trusted third party (DataThroughTtp)**' [3], seen in Figure 8.

To realize the intended objectives of this scenario, five microservices were specifically created in addition to the services discussed in section IV. These services include:' a Query service facilitating data retrieval from databases, an Anonymize service dedicated to anonymizing sensitive data, an Aggregate service consolidating data from multiple queries, an Algorithm service executing data analysis according to specified algorithms, and ultimately, a Graph service that optionally transforms results into graphical representations.

The distributed architecture to create this PoC is depicted in Figure 5. It recognizes a **control plane** and **data plane**. In the control plane, the process orchestrator makes archetypes, and job decisions in conformance with a policy evaluator. In the data plane, distributed agents are responsible for scheduling the services, and updating the current system state in the *distributed datastore*.

VI. DISCUSSION

We discuss the observations and knowledge acquired from designing and implementing XYZ, along with the results from the case study. This section concludes with threats to the validity and future work of this research.

A. Architecture - Lessons Learned

1) Sidecar Pattern: Throughout the design and implementation phases of XYZ, numerous observations surfaced. The first observation relates to the Sidecar pattern, which XYZ utilizes as explained in subsection IV-E. However, during development, we considered removing the sidecar from the architecture by compiling a messaging (RabbitMQ) library into each data-sharing microservices. This would decrease the amount of data transfers, latency, and would remove the need for the current gRPC data transfer methodology.

However, this approach comes with its own set of challenges:

- **Tight Coupling**: Direct integration would result in a strong dependency on the RabbitMQ platform, rendering transitioning to another messaging solution more challenging.
- **Update Overheads**: Any change or update to the messaging library would necessitate modifications, recompilation, and redeployment of all data-exchange microservices.
- **Maintenance Issues**: Microservices developed in various programming languages would use distinct libraries to connect to a single RabbitMQ instance, potentially leading to increased maintenance complexities.

2) Protocol Buffers: Another observation relates to the utilization of Google's Protocol Buffer messages. The integration of protobuf offered a standardized and language-independent method for facilitating communication between services. Although this proved pivotal in the design and implementation of XYZ, it does however slightly heighten the complexity of the codebase. This complexity originated from the requisite generation, typing, and incorporation of additional files within the codebase.

3) API Gateway: In the latter stages of XYZ development, we introduced an API gateway. Recognized for its utility, it effectively segregated the public-facing API from the private calls managed by the orchestrator. The creation of the API gateway had the potential to enhance the separation of concerns and elevate the overall code quality.

B. Ephemeral jobs

Setting up ephemeral jobs proved intricate mostly due to to timing issues, each service needs to be ready in time to send and receive messages. Yet they guaranteed enhanced stability and a streamlined infrastructure after each job cycle. Given that data is transient and never stored, this approach bolstered security measures without compromising traceability.

1) Cold starts: A 1.7-second average startup loss is long in the modern web application world. However, in this research, we cannot qualify whether this would be unacceptable for a digital data-exchange marketplace solution. One could argue that since the dataspace removes a large amount of effort in getting access to otherwise near-inaccessible datasets, these few extra seconds in latency are trivial.

If future user requirements however seek to improve this cold start latency, there are ways to mitigate this. Using a *persistent job* (the counter-part to ephemeral jobs) architecture could be used to avoid cold starts altogether. The trade-off is unnecessary resource consumption and possible security risks.

For the *ephemeral job* architecture, an alternative solution would be not to allow the user to add extra options dynamically, like the 'graph' option. This would allow the agents to always have a job ready to reduce or avoid cold starts. This comes at the cost of flexibility, user experience, resource consumption, and increased exposure time which is a possible security risk.

2) Data transfers: We devised a customized data transfer approach for ephemeral jobs using gRPC to circumvent the frequent creation and removal of single-use queues in the messaging platform. This methodology offered several advantages: data was transferred at the Inter Process Communication level, ensuring language agnosticism and loose coupling. However, it entailed some drawbacks, including a complex implementation process and the necessity for a small gRPC library per language utilized in the data exchange microservices.

C. Further insights

This design of XYZ provides a clear separation of concerns, creating three main parts of the system. The protocol files define an interface between these components making it explicit what parts of the system can expect from each other.

One part is the interaction between the orchestrator and the policy enforcer framework. The interface could help the external development of the policy enforcer framework due to the explicitness of the expected results.

The second part is the combination of the orchestrator and the distributed agents, in which the orchestrator chooses archetypes; vice-versa, the agents provide information on that system's state to guide these choices.

Thirdly, the data-exchange microservices are a separate component.

D. Threats to validity

1) Specific sample use case: In this research, we took two archetypes and the UNL use case as the basis for the XYZ implementation. We believe the implementation of the language to generate microservice chains, including the way archetypes are defined, is general enough to extend to more use cases and archetypes. This, however, is not proven and requires a more exact analysis of different use cases, request types, and archetypes.

2) Generic microservices: We made a generic method of passing data from microservice to microservice and showed with a small example how this data can be manipulated. However, we fell short of realizing the full chain as described in the use case. With our current implementation we cannot prove that this method of passing data is user-friendly, or generic enough to scale up toward more complex data-sharing scenarios.

3) Authorization and data pods: The generic architecture hinges on the idea that we can implement web tokens, that can be invalidated, to access data pods on data stewards. This has not been tested, or researched much for XYZ, although we believe that the techniques and technology do exist to make this happen.

4) Message size: Streaming data would be a possible method of working with larger datasets. How this would be exactly processed by microservices expecting, for example, SQL data, has not been properly researched in this paper.

5) HTTP: For communication between the microservices, whether it be for persistent or ephemeral jobs, we solely focused on an AMQ or gRPC approach in this thesis. We believe in this scenario it makes sense to have stable queueing in the form of AMQ, and in ephemeral jobs to have the direct IPC communication of gRPC. However, we never fully investigated whether HTTP should be completely discarded as an option. We believing putting some extra effort into listing out the advantages and disadvantages of using HTTP would be beneficial.

E. Future work

There is a whole lot of future research we can think of to continue the work on XYZ. A few possibilities spring to mind however as more essential to show that the general architecture of XYZ is relevant.

1) Data pods and authorization: A logical next step would be to implement a data pod solution and show how a user can get authentication to a dataset in that specific data pod, and nowhere else. This entire flow could be essential in the dataspace, irrespective of how the final application is designed.

2) Expand archetypes and scenarios: To see if the current microservice chain generation holds, it is essential to study more use cases. Another interesting use case would be a federated learning model, where the results of learning algorithms on different parties aggregate. And archetypes that are more flexible in who gets the results.

3) Archetype selection algorithms: XYZ has shown a framework and method on how active, and future jobs can dynamically pick archetypes. Especially algorithms concerning 'green-IT' and resource consumption require more extensive research.

VII. Related Work

This section highlights work that is tangential to this research, particularly relating to Digital Data-exchange Marketplaces and Self-adaptive Microservice systems.

A. Digital Data-exchange Marketplaces

Although not the first, Shakeri *et al.* [1] clearly describes a set of archetypes and proposes a high-level architecture for a container-based Digital data-exchange marketplace.

Zhang *et al.* [3] model customer requests to archetypes and complement these with possible metrics to use for evaluation. The authors list as possible future work: 'How to map applications into such policy-driven infrastructures?'.

Both papers are major sources of inspiration for XYZ.

B. Self-adaptive Microservice systems

Research on self-adaptive microservices tends to be focused on architecture and modeling [10, 13–15]. Mendonça *et al.* acknowledge the limited scope of current works addressing the challenges of developing self-adaptive microservice applications [16–18]. They strive to bridge the gap between state-of-the-art research and practice, particularly emphasizing the future of self-adaptive microservices. Their work proposes a microservices service-mesh as the path to self-adaptability, exemplified by KubowMesh, a modern microservice approach built on Kubernetes and the Istio service mesh [16, 17] based on the self-adaptability patterns of Rainbow [19].

While Mendonça *et al.* provide a practical approach, other papers contribute key ideas. The concept of distributed agents orchestrating microservices in a heterogeneous system draws from the computing-fleet reference architecture, which proposes a three-layered architecture for microservice orchestration on edge computing devices [13]. Many papers on self-adaptive (microservice) systems predominantly focus on 'Quality of Service' attributes such as costs, seamless upgrades, resiliency, and scaling. For instance, a comprehensive literature review on QoS-driven service composition under uncertainty provides a taxonomy, comparisons, and analysis of state-of-the-art approaches [14]. Additionally, a multi-level self-adaptation architecture and domain-specific language (DSL) proposal addresses seamless service upgrades [15].

The work of Arcaini *et al.* [10] formalizes the modeling of MAPE-K systems in distributed systems, discussing aspects of validation and correctness [10]. Boyapati *et al.* [11] implement a lightweight MAPE-K loop to control dynamic rate limiting in a Docker compose setup.

VIII. Conclusion

Witness to the growing public and private sectors' interest in data exchange, recent literature introduces several approaches to how DDM architecture could be created.

XYZ makes this practical. It can be used to experiment with security, different archetypes, and how to interact with a policy enforcer. It allows for each part of the system to operate distinctly, just like an actual data steward might want to do things differently from another.

We defined microservice chains as a directed acyclic graph, created a custom method of passing data between services in short-lived ephemeral jobs, designed two microservice architectures, and incorporated two data-sharing archetypes. The power of XYZ is that it can dynamically choose archetypes and architectures. Finally, we performed initial experiments to gain insight into bottlenecks and the advantages and disadvantages of different microservice architectures.

We also reflect on the choices made in designing XYZ. Microservices, which comprise loosely coupled date-exchange services allow for separate parties to develop the services, while XYZ abstracts away the complexity. Latency is increased due to the microservices and the sidecar pattern, however, in our view, the flexibility benefits this brings outweigh this issue.

This practical application of policy, archetypes, and microservices informs future work toward the wide adoption of data exchanges.

References

[1] S. Shakeri, L. Veen, and P. Grosso, "Evaluation of container overlays for secure data sharing," in *2020 IEEE 45th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, 2020, pp. 99–108. DOI: 10.1109/LCNSymposium50271.2020.9363266.

[2] L. T. van Binsbergen, M. Oost-Rosengren, H. Schreijer, F. Dijkstra, and T. van Dijk, *Amdex reference architecture – version 1.0.0*, version 1.0.0, Feb. 2024. DOI: 10.5281/zenodo.10565916. [Online]. Available: https://doi.org/10.5281/zenodo.10565916.

[3] L. Zhang, R. Cushing, L. Gommans, C. de Laat, and P. Grosso, "Modeling of collaboration archetypes in digital market places," *IEEE Access*, vol. 7, pp. 102 689–102 700, 2019.

[4] E. Commision, *Communication from the commission to the european parliament, the council, the european economic and social committee and the committee of the regions a european strategy for data*, 2020. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:52020DC0066.

[5] *Generiek afsprakenstelsel voor datadeelinitiatieven als basis van de digitale economie*, 2018. [Online]. Available: https : / / dokumen . tips / documents / generiek - afsprakenstelsel - voor - datadeelinitiatieven - als - datadelenmkbpdf.html?page=1.

[6] L. T. van Binsbergen, L.-C. Liu, R. van Doesburg, and T. M. V. engers, “Eflint: A domain-specific language for executable norm specifications,” *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2020.

[7] B. Burns, *Designing distributed systems: patterns and paradigms for scalable, reliable services.* ” O’Reilly Media, Inc.”, 2018.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. C. Reed, “Zookeeper: Wait-free coordination for internet-scale systems,” in *USENIX Annual Technical Conference*, 2010.

[9] RedHat. “Introduction to kubernetes architecture.” (), [Online]. Available: https://www.redhat.com/en/topics/containers/kubernetes-architecture.

[10] P. Arcaini, E. Riccobene, and P. Scandurra, “Modeling and analyzing mape-k feedback loops for self-adaptation,” in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015, pp. 13–23. DOI: 10.1109/SEAMS.2015.10.

[11] S. R. Boyapati and C. Szabo, “Self-adaptation in microservice architectures: A case study,” in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2022, pp. 42–51. DOI: 10.1109/ICECCS54210.2022.00014.

[12] M. Steketee. (2023), [Online]. Available: https://amdex.eu/news/sharing-research-data-under-ones-own-conditions/.

[13] D. Roman, H. Song, K. Loupos, T. Krousarlis, A. Soylu, and A. F. Skarmeta, “The computing fleet: Managing microservices-based applications on the computing continuum,” in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*, 2022, pp. 40–44. DOI: 10.1109/ICSA-C54293.2022.00015.

[14] M. Razian, M. Fathian, R. Bahsoon, A. N. Toosi, and R. Buyya, “Service composition in dynamic environments: A systematic review and future directions,” *Journal of Systems and Software*, vol. 188, p. 111 290, 2022.

[15] S. Zhang, M. Zhang, L. Ni, and P. Liu, “A multi-level self-adaptation approach for microservice systems,” *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 498–502, 2019.

[16] N. C. Mendonça, D. Garlan, B. Schmerl, and J. Cámara, “Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices,” in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ser. ECSA ’18, Madrid, Spain: Association for Computing Machinery, 2018, ISBN: 9781450364836. DOI: 10.1145/3241403.3241423. [Online]. Available: https://doi.org/10.1145/3241403.3241423.

[17] N. das Chagas Mendonça and C. M. Aderaldo, “Towards first-class architectural connectors: The case for self-adaptive service meshes,” *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, 2021.

[18] N. das Chagas Mendonça, P. Jamshidi, D. Garlan, and C. Pahl, “Developing self-adaptive microservice systems: Challenges and directions,” *IEEE Software*, vol. 38, pp. 70–79, 2019.

[19] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 276–277, 2004.